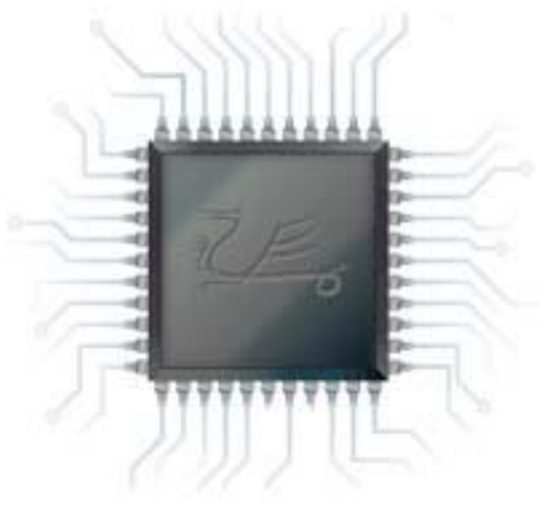
Co-simulating with OSVVM



Simon Southwell

June 2023

(updated 3rd November 2025)

Preface

The contents of this document were originally posted on the OSVVM website as a set of blogs between February and June 2023. The original articles can be found here and the text refers to the features released with OSVVM 2023.01 or newer.

Simon Southwell
Cambridge, UK
June 2023

Contents

Part 1: Co-simulation Introduction

Introduction

I have written previously about co-simulation in my series of LinkedIn articles about using the programming interfaces of logic simulator tools in a fairly generic way, summarising the possibilities and referencing, in outline, some real-world use cases. Since then, I have been collaborating to bring some of these co-simulation features to OSVVM, an open-source VHDL verification methodology consisting of a set of libraries and packages for easing and improving the verification of logic IP. In this article I want to take a look at some of the new features and how these might be used to extend the possibilities for verification with OSVVM to include developing and testing with software, interfacing to external C/C++ models, generating tests in C/C++ and more.

Co-simulation Definitions

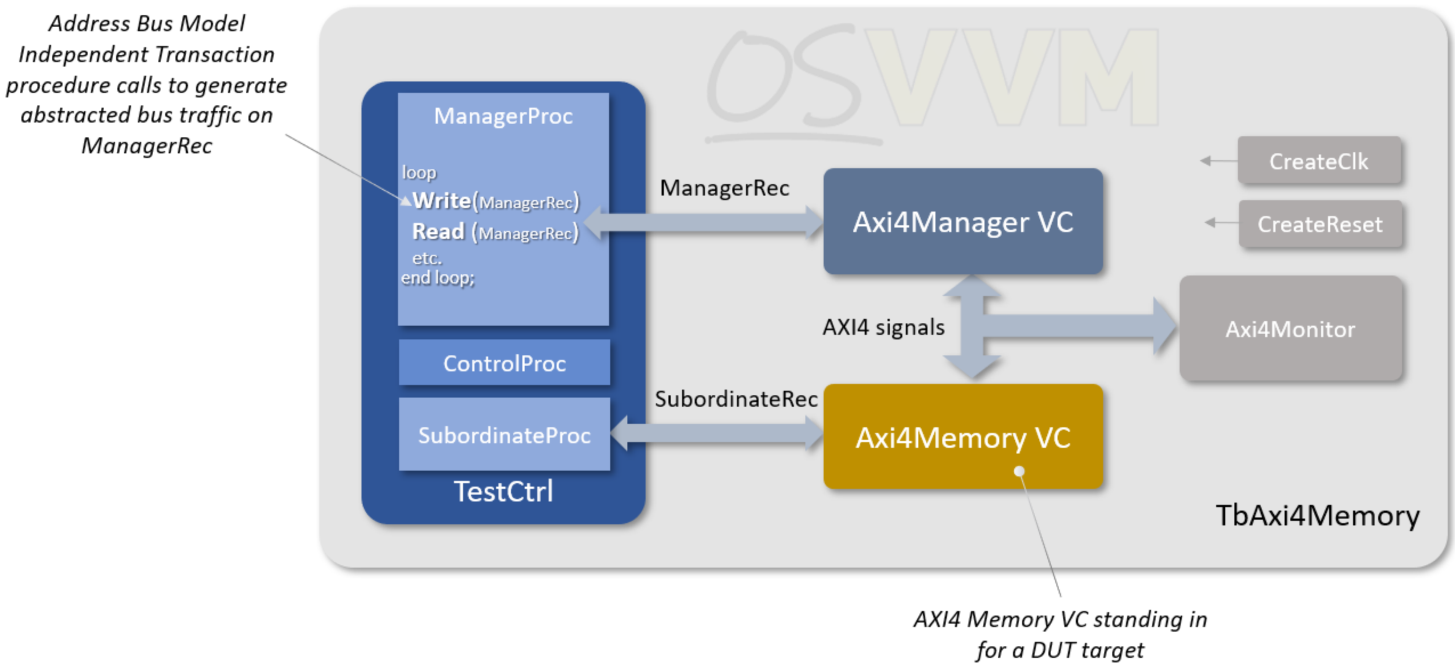
Before we get going, I want to define what is meant by co-simulation in this context as it can mean different things to different people. At a base level it can just mean running logic simulations via secondary languages such as Python (e.g., cocotb). These can be very low level allowing the driving of individual signals via the secondary language to generate test vectors. At the other end I have seen co-simulation refer to running FPGA emulated logic hardware with the embedded software to provide a pre-silicon, system level, platform of integrated testing.

OSVVM is set of VHDL libraries and procedures to allow methodical testing of logic IP on a logic simulator to easily meet test and coverage goals etc. In this context, then, OSVVM co-simulation is also about running software in a logic simulation environment. The software side, as we shall see, is at the bus transaction level, utilising pre-existing OSVVM features, and can be anything from writing transaction test vectors in C++ to modelling a whole SoC system and its embedded software with a mixture of C++ models and real logic IP.

OSVVM Address Bus Transaction Level Modelling

There is not enough space to go through all of OSVVM’s generous set of functionality here but, relevant to co-simulation, it includes Address Bus Model Independent Transaction features, whereby VHDL can issue abstracted bus read and write requests (either word or burst) via a set of API procedures that are then translated to specific bus protocols, such as AXI, via verification components (VCs)—so called Transaction Level Modelling (TLM). The advantage of this is that tests that run for one protocol can be run for another protocol by just using a different VC to map the abstracted transactions to the particular protocol signalling.

A test bench within the OSVVM repository’s [AXI4 sub-module]() demonstrates how this works. The particular example we are going to reference is a test case located in `AXI4/Axi4/TestCases/TbAxi4_RandomReadWrite.vhd`, running on the test bench `AXI4/Axi4/testbench/TbAxi4Memory.vhd`. The block diagram below outlines this arrangement:



Here we see a manager process generating abstracted transactions via a record, `ManagerRec`, containing all the fundamental transaction information (e.g., address data, width, burst size etc.). In this example it does this with random calls to the OSVVM provided transaction procedures (such as `Write()`, `Read()`, `ReadCheck()` etc.) in a loop, terminating after a set number of transactions have been generated. The responses to the transactions, such as read data or error status, are returned within the same `ManagerRec` record back to the manager process. In this sense, the manager process here is acting like a processor core, as if issuing reads and writes on load and store instructions.

The `ManagerRec` is connected to an `Axi4Manager` verification component which converts the abstract transactions to specific AXI4 protocol manager signalling. A subordinate is connected to the AXI4 bus—in this example it is another OSVVM VC, being the `Axi4Memory`. Of course, in a real test environment it would be the device under test (DUT) that would be connected, which could be a single subordinate peripheral logic IP or a whole sub-system, with AXI interconnect and several IP blocks. The use of the `Axi4Memory` VC in this example is just to demonstrate and test the OSVVM components and infrastructure but serves to have a target that read and write transactions may be directed towards. As such this `Axi4Memory` VC has a `SubordinateRec` output (of the same type as `ManagerRec`) which is directed at a subordinate process and allows cross checking between issued manager transactions and received subordinate transactions.

This, then, is a brief summary of one possibility for the use of the Address Bus Model Independent Transaction features of OSVVM. Much more detail can be found in the OSVVM documentation.

Adding Co-simulation to OSVVM

Using the basic outline from the previous section we would like to add co-simulation capability to OSVVM with minimal disruption to the transaction level modelling features that are already in place. In essence we still want to generate address bus transactions, but from software rather than VHDL. None-the-less there must be a connection point within VHDL and, ideally, the procedures provided for the transactions are replaced with a simple procedure that connects to the C/C++ software domain, hiding the details of this from the user. As such it would have, as for the address bus transaction procedures, a manager record argument for generating the abstracted bus transactions and receiving the responses. It would also need to have an interrupt input (more later) and some status outputs to flag errors within the software and to indicate when the software has completed. The definition for the procedure provided in OSVVM is shown below:

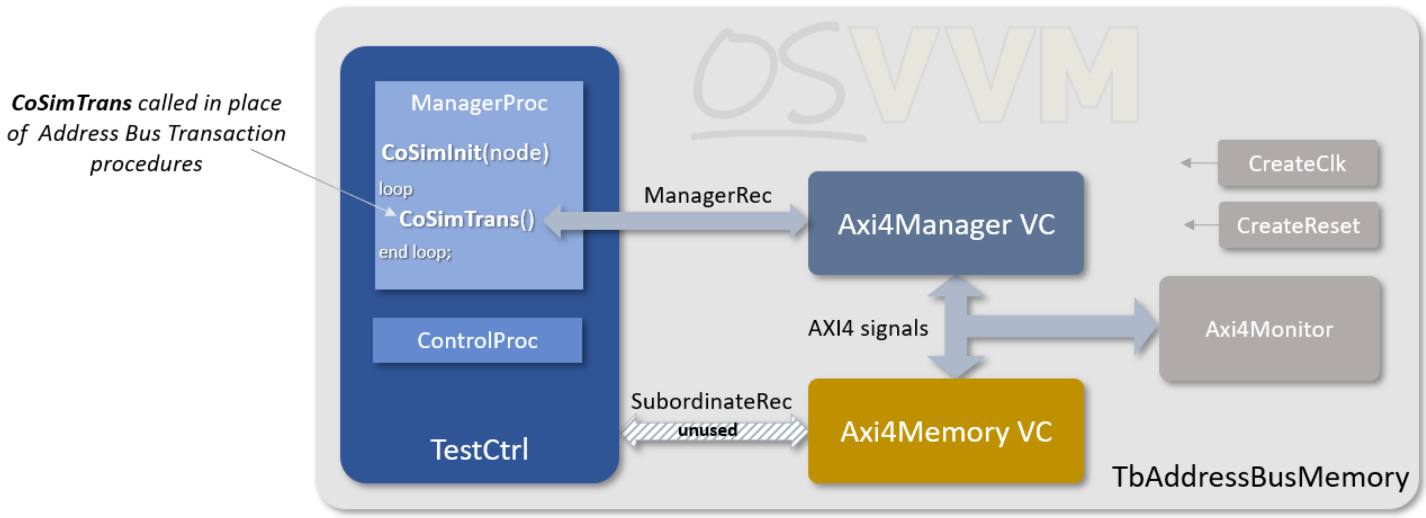
```
procedure CoSimTrans (
  signal    ManagerRec        : inout  AddressBusRecType ;
  variable Done               : inout  integer ;
  variable Error              : inout  integer ;
  variable IntReq             : in     integer := 0 ;
  variable NodeNum            : in     integer := 0
);
```

The arguments include those as discussed above, with a `ManagerRec` for transactions, an `IntReq` to receive interrupt state, an `Error` output, and a `Done` flag. The only additional argument is the node number (`NodeNum`). This allows multiple calls to the procedure from other processes for generating transactions on different abstracted busses. Each use of the procedure in a different process will use a unique node number. This is analogous to having multiple processor cores with their own address bus generating reads and write transactions which can be connected, say, to an interconnect with multiple subordinate interfaces.

There is also an initialisation procedure which is used to initialise a co-simulation interface, starting up the software for that node, and must be called before any calls to `CoSimTrans` of the same node number.

```
procedure CoSimInit (variable NodeNum : in integer);
```

Taking the example that we discussed before, we can now modify this to use the new procedure.



Now, instead of calling `Read` and `Write` procedures in the loop, `CoSimTrans` is called instead. The rest of the test environment remains unchanged. The `SubordinateRec` is not used as the transactions now originate in software which does its own checks on data and a DUT that would sit in place of the `Axi4Memory` in a real test setup would also not have this. The above example outlines the test environment in the `CoSim` sub-module of the OSVVM repository and can be found in the file `CoSim/TestCases/TbAb_CoSim.vhd` which uses the test bench defined in the file `CoSim/testbench/TbAxi4/TbAddressBusMemory.vhd`.

And that's all there is to it from the VHDL side. But from this, as we shall see, we can do such things as run a program on a processor modelled in software right out of the `Axi4Memory` VC (standing in for DUT) or connect to an external program via a TCP/IP socket or just write tests in C++ to generate transactions, as well as much more. These are just some of the possibilities and the use of the co-simulation features are not limited to the arrangement discussed above or to the examples we will look at below—which is also true for the rest of the OSVVM features, and the full range of OSVVM documentation should be inspected to understand all the possible use cases.

Having looked at the VHDL side of things, let’s see what it looks like from the C++ software viewpoint and later we will look at how some of this is achieved by lifting the lid on the underlying co-simulation code.

The Software View

With `CoSimTrans` looping in a process we can now generate transaction from a C++ program. In a similar manner to `main()` being the entry point in a C program or `WinMain()` being the entry point in a graphical Windows program, the entry point for an OSVVM co-simulation program is `VUserMain`*n*`()` when *n* is the node number we used for the `CoSimInit` and `CoSimTrans` calls. Thus, if using the default value for `NodeNum` of 0, the software entry point is `VUserMain0()`. From here (or any other called function, class method or sub-program) we gain access to transactions via the provided `OsvvmCosim` class (defined in `CoSim/include/OsvvmCosim.h`). This is a wrapper class that sits on top of the lower-level co-simulation code and holds no state of its own except its node number, set at construction. This means it is safe to have as many instantiations of the class for a given node as needed without causing conflicts. When the class is constructed the node number is given as an argument to tie it to that particular node in VHDL. A test name can, optionally, be given at construction as well—usually only once for a given node if multiple instances of the class for the same node—and this identifies the particular software being run which may share the same VHDL test bench with multiple software tests.

The class provides methods to generate read and write transactions. These are summarised below:

```
uint<m>_t transWrite     (uint<n>_t addr, uint<m>_t  data);
void      transRead      (uint<n>_t addr, uint<m>_t *data);
void      transBurstWrite(uint<n>_t addr, uint8_t   *data, int bytesize);
void      transBurstRead (uint<n>_t addr, uint8_t   *data, int bytesize);
```

The methods are overloaded such that the size of the address type and the size of the data type determine the type of transfer. So, the `uint<n>_t` for the address can be `uint32_t` or `uint64_t` for 32-bit and 64-bit architectures. Similarly, the `uint<m>_t` type for the data can be `uint8_t`, `uint16_t`, `uint32_t` or `uint64_t` (if a 64-bit architecture) to allow byte, half-word, word and double-word transfers. For the burst methods, the address can be 32- or 64-bit, as before, with data transferred in or out of a buffer, passed in as a pointer, and the size of the burst via the `bytesize` parameter. With these methods the program can now generate transactions in the testbench, just as if using the OSVVM Address Bus Model Independent Transaction procedures in VHDL.

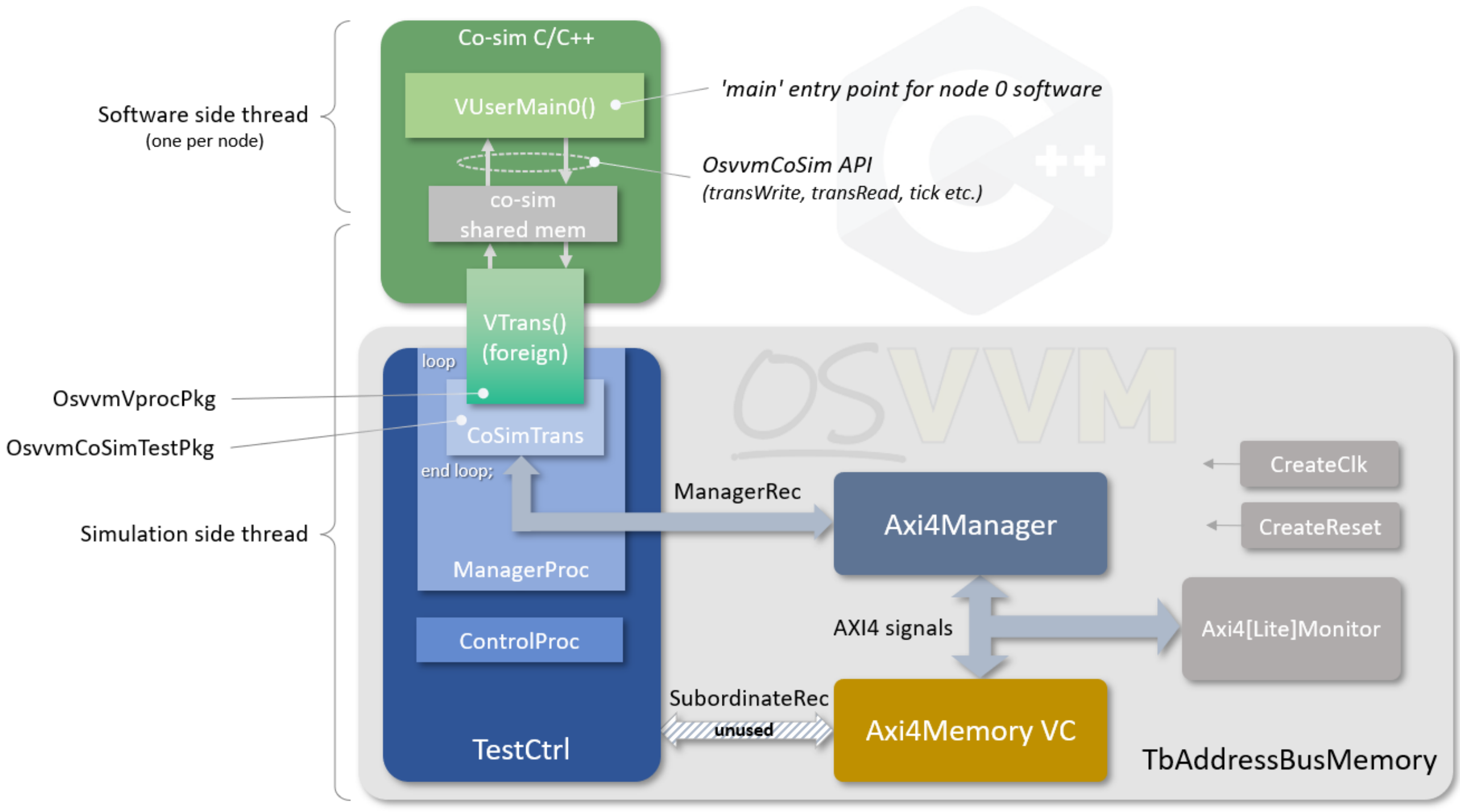
The way the underlying co-simulation code works, the simulation will not advance time and be blocked, once `CoSimTrans` is called, until a call to one of the transaction methods is made in software. The simulation will then advance for as many cycles as required to complete the transaction. Of course, the simulation code might do other things between calls to `CoSimTrans` that advances time, and the user software will be blocked in its call to the transaction method whilst this is going on. There may be times when one might want to advance simulation time without generating a transaction. The `OsvvmCosim` class provides a 'tick' method:

```
  void tick (int ticks, bool done, bool error);
```

The term 'tick' here is loosely defined as basically a single call to `CoSimTrans`. This uses the OSVVM `WaitForClock` procedure on the `ManagerRec` parameter, and so will tick for the specified number of clock cycles associated with the transaction interface. Thus, the tick method allows for passing control back to the simulation for a set number of cycles when it is known that the software does not need to generate new transactions for some period. Notice, also, that there are 'done' and 'error' arguments. This is a means to affect the equivalent outputs on `CoSimTrans` to flag that the software is finished and indicate any error status.

The class also provides a method for registering an interrupt callback function, but we shall deal with interrupts in separate sections below.

Having looked at the VHDL side and now the software side, we have a system for effectively writing tests in C++ in place of calls to the VHDL procedures. Effectively we have made some of the Address Bus Model Independent Transaction procedures available in C++. This is useful in itself, but if that was all that could be done the use would be restrictive. At present only the blocking transaction functionality is available, though there are plans to add the other non-blocking, asynchronous and checking functionality as well, allowing a full range of tests to be written in C++. The diagram below summarises the situation with both the VHDL and C++ domains and an indication of the underlying code structure.



With this simple setup, beyond simply writing tests in C++, we can now do much more in terms of modelling SoC systems or connecting to external programs whilst running logic simulations with IP we are developing and wish to test. In the limit, the software used to drive the IP could also be run and co-developed with the logic IP using co-simulation, long before silicon becomes available. In the next couple of sections of this article I want to look at some examples that are available in the OSVVM repository of just such arrangements, but the usage is not limited to just these and it would be possible to hook up to other complex system models using these techniques. We will look at interrupts once we get to running a processor model and interrupting this.

Co-simulating with a Processor Model

In the discussions up until now, any code we write would be writing software to generate transactions for test vectors. Just such a program is in the `CoSim` sub-module repository in `CoSim/tests/usercode_size/VUserMain0.cpp` (there's also a burst equivalent in `usercode_burst`). In this section I want to look at using a C++ RISC-V instruction set simulator (ISS) model to hook-up to the co-simulation API so that it can access the logic simulated AXI address bus. This sits on top of the environment described in the sections above and requires no additional changes to the infrastructure already outlined.

At its most basic a single core processor has a clock and a reset, along with a manager type bus interface (I'm ignoring Harvard architecture for now) and one or more interrupt inputs. This is very similar to the `CoSimTrans` procedure's `ManagerRec` and `IntReq` parameters. The model being used is my riscV 32-bit RISC-V RV32GC ISS. This model pre-dates the OSVVM co-simulation features and has not been modified for OSVVM and I want to highlight how easy this was to integrate on top of the software API. The ISS allows for registering callback functions—one for memory accesses and one for interrupts. These features are used to generate read and write transactions and to allow program interrupts. Below is some abbreviated sample main code.

```
// Co-sim user code entry point for node 0
extern "C" void VUserMain0()
{
    OsvvmCosim cosim(node);

    rv32i_cfg_s cfg; // ISS config structure
    bool error = false;

    rv32* pCpu = new rv32(); // ISS object

    // Register memory access callback
    pCpu->register_ext_mem_callback(memcosim);

    // Register an interrupt callback with the CPU model
    pCpu->register_int_callback(interrupt);

    // Register an interrupt callback with the cosim software
    cosim.regInterruptCB(cosim_int_callback);

    // Load a program and run the ISS model
    if (!pCpu->read_elf("test.exe")) {
        pCpu->run(cfg);
        error = check_exit_status(pCpu);
    }
    else
        error = true;

    // Clean up
    delete pCpu;

    // Flag to sim that the test is finished
    cosim.tick(10, true, error);
    SLEEPFOREVER;
}
```

Here an `OsvvmCosim` object is created for node 0 (`cosim`). The ISS has a defined configuration structure of type `rv32i_cfg_s` and we will assume this is filled in appropriately. The ISS class itself is of type `rv32` and a pointer, `pCpu`, is set to a new instance of the model. Two callback functions are then registered for memory accesses (memcosim) and interrupts (interrupt)—more in a bit—before loading a program (pCpu->read_elf) and running the model (pCpu->run). That's more or less it for the main program, with the interfacing to the co-simulation done via the callbacks. The memory access callback might look something like the following abbreviated code fragment:

```
int memcosim (const uint32_t byte_addr, uint32_t &data,
              const int      type,      const rv32i_time_t time)
{
  OsvvmCosim cosim(node);
  int        cycle_count = 5;
  uint8_t    rdata8;
  uint16_t   rdata16;
  uint32_t   rdata32;

  switch (type)
  {
    case MEM_WR_ACCESS_BYTE : cosim.transWrite(byte_addr,  (uint8_t)data); break;
    case MEM_WR_ACCESS_HWORD: cosim.transWrite(byte_addr, (uint16_t)data); break;
    case MEM_WR_ACCESS_WORD:  cosim.transWrite(byte_addr, (uint32_t)data); break;
    case MEM_WR_ACCESS_INSTR: cosim.transWrite(byte_addr, (uint32_t)data); break;
    case MEM_RD_ACCESS_BYTE:  cosim.transRead(byte_addr, &rdata8);  data=rdata8;  break;
    case MEM_RD_ACCESS_HWORD: cosim.transRead(byte_addr, &rdata16); data=rdata16; break;
    case MEM_RD_ACCESS_WORD:  cosim.transRead(byte_addr, &rdata32); data=rdata32; break;
    case MEM_RD_ACCESS_INSTR: cosim.transRead(byte_addr, &rdata32); data=rdata32; break;
    default: cycle_count = RV32I_EXT_MEM_NOT_PROCESSED; break
  }
  return cycle_count;
}
```

When the ISS calls the callback function it provides an address, a reference to a data word, an access type, and a current 'time' (which we will gloss over here as we don't really use it). Here we simply have a switch statement on the access types and call the appropriate OsvvmCosim method with the appropriate data type. Note that there are *instruction* write and read types as well as the normal data access types. This allows for loading of code (instruction writes) and reading of program instructions (instruction reads) via the co-simulation interface, as well as the normal load and store of data.

The interrupt callback is even simpler:

```
uint32_t interrupt (const rv32i_time_t time, rv32i_time_t *wakeup_time)
{
    *wakeup_time = 0;
    return IntReq ? 1 : 0;
}
```

The ISS passes in the current time and a pointer for returning a 'wake-up' time—i.e., the time for the next scheduled call to the interrupt callback. Again, time modelling is of no interest and we will always set this to 0 to have no delay in calling. The callback returns 1 if an active interrupt is set, else it returns 0—the RISC-V architecture only defines a single external interrupt, with multiple interrupts handled by an external processor level interrupt controller (PLIC). `IntReq` is an external variable that is updated to reflect the current state of the `IntReq` parameter of the `CoSimTrans` VHDL procedure. In order to get hold of that we need to register another callback—this time with the co-simulation software. The `OsvvmCosim` class has a `regInterruptCB` method for registering a callback function which will be called whenever the `CoSimTrans IntReq` parameter changes. For our example, the code might look something like the following:

```
int cosim_int_callback(int int_vec)
{
    IntReq = int_vec & 1;
    return 0;
}
```

When the co-simulation software calls this it simply updates the `IntReq` local static variable with the low bit of the passed in interrupt vector state, making it available to the ISS via its `interrupt` callback. There are better ways of doing this than using a locally static variable, but things have been kept simple for this example.

Note that the co-simulation callback function is only meant to update local state to allow the main user code to process it. It can't be used to make additional calls to transaction methods of an `OsvvmCosim` instance. This is because the callback is instigated from the simulation with the main program blocked on its last call to a transaction method. To call a new transaction method would be to do so in the middle of a pending transaction. The main code is responsible for modelling the actual interrupt actions, just as is done by the ISS model in this case.

So, having hooked up the ISS model to the co-simulation interface, what can we do with this. Well, the ISS can run cross-compiled RISC-V programs which we can load and run on the model. In the OSVVM provided example in `CoSim/tests/iss` a precompiled program called `test.exe` is used, alongside the `VUserMain0.cpp` program. This is actually a 3rd party piece of software. The RISC-V International organization provide a suite of instruction unit tests and the test code is one of these from the `rv32ui` set for testing byte store instructions.

With the user code configuring the model to do run time disassembly and to dump the register state on termination the following shows a fragment of the output of the simulation:

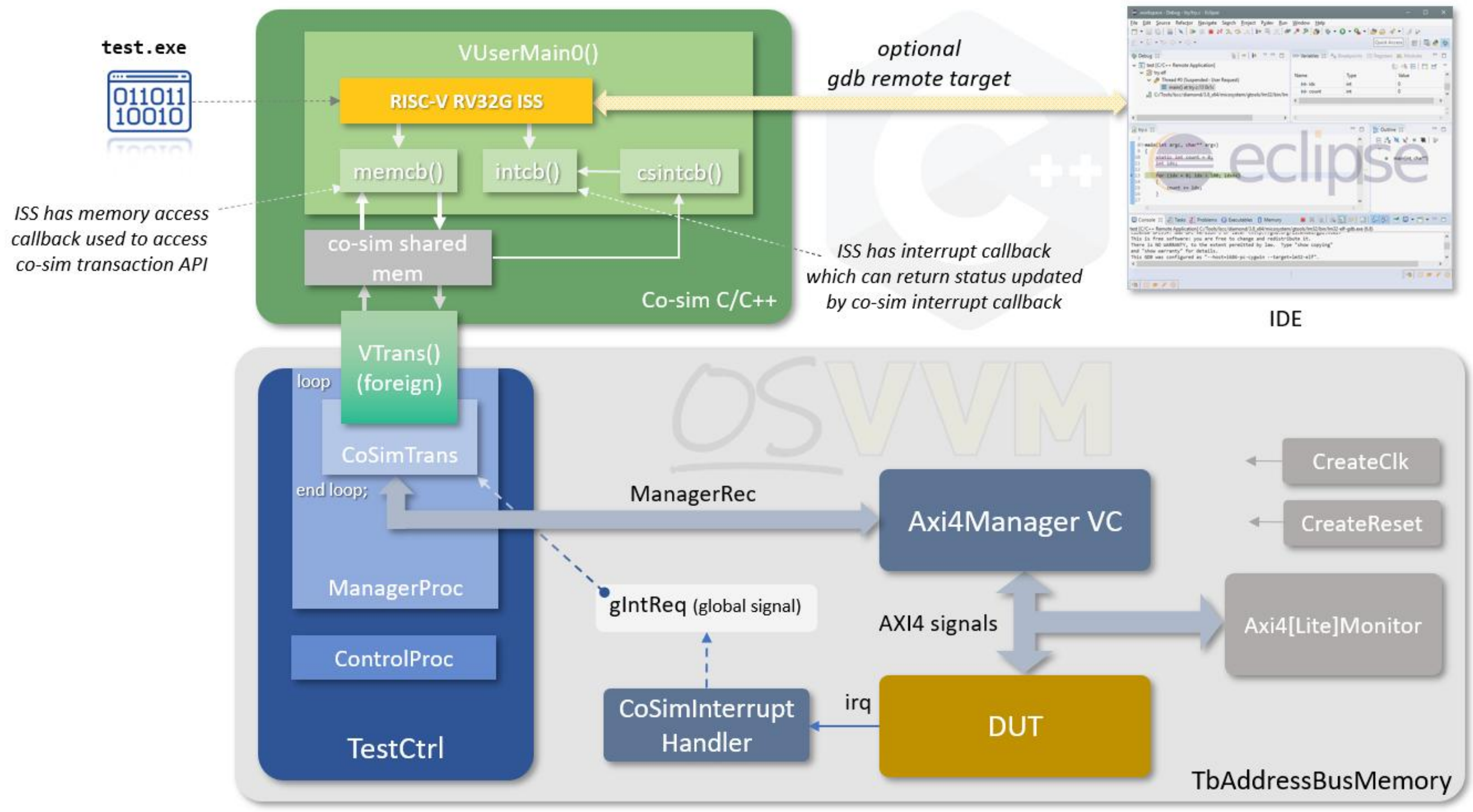
```
00000564: 0x00a581a3    sb       a0, 3(a1)
%% Log   INFO      in manager_1,   Write Data.  WData: EFUUUUUU  WStrb: 1000  Operation# 451 at 21030 ns
%% Log   INFO      in manager_1,   Write Address.  AWAddr: 000015C3  AWProt: 000  Operation# 451 at 21030 ns
%% Log   INFO      in memory_1,   Memory Write.  AWAddr: 000015C3  AWProt: 000  WData: EFUUUUUU  WStrb: 1000  Operation# 450 at 21040 ns
%% Log   INFO      in manager_1,   Read Address.  ARAddr: 00000568  ARProt: 000  Operation# 532 at 21040 ns
%% Log   PASSED    in manager_1: WriteResponse Scoreboard,      Received: 0   Item Number: 451 at 21050 ns
%% Log   INFO      in memory_1,   Memory Read.  ARAddr: 00000568  ARProt: 000  RData: 02301063  Operation# 531 at 21050 ns
%% Log   PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 532 at 21060 ns
%% Log   INFO      in manager_1,   Read Data: 02301063  Read Address: 00000568  Prot: 0 at 21060 ns
00000568: 0x02301063    bne      zero, gp, 32
    *
%% Log   INFO      in manager_1,   Read Address.  ARAddr: 00000588  ARProt: 000  Operation# 533 at 21060 ns
%% Log   INFO      in memory_1,   Memory Read.  ARAddr: 00000588  ARProt: 000  RData: 0FF0000F  Operation# 532 at 21070 ns
%% Log   PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 533 at 21080 ns
%% Log   INFO      in manager_1,   Read Data: 0FF0000F  Read Address: 00000588  Prot: 0 at 21080 ns
00000588: 0x0ff0000f    fence    15, 15
%% Log   INFO      in manager_1,   Read Address.  ARAddr: 0000058C  ARProt: 000  Operation# 534 at 21080 ns
%% Log   INFO      in memory_1,   Memory Read.  ARAddr: 0000058C  ARProt: 000  RData: 00100193  Operation# 533 at 21090 ns
%% Log   PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 534 at 21100 ns
%% Log   INFO      in manager_1,   Read Data: 00100193  Read Address: 0000058C  Prot: 0 at 21100 ns
0000058c: 0x00100193    addi     gp, zero, 1
%% Log   INFO      in manager_1,   Read Address.  ARAddr: 00000590  ARProt: 000  Operation# 535 at 21100 ns
%% Log   INFO      in memory_1,   Memory Read.  ARAddr: 00000590  ARProt: 000  RData: 05D00893  Operation# 534 at 21110 ns
%% Log   PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 535 at 21120 ns
%% Log   INFO      in manager_1,   Read Data: 05D00893  Read Address: 00000590  Prot: 0 at 21120 ns
00000590: 0x05d00893    addi     a7, zero, 93
%% Log   INFO      in manager_1,   Read Address.  ARAddr: 00000594  ARProt: 000  Operation# 536 at 21120 ns
%% Log   INFO      in memory_1,   Memory Read.  ARAddr: 00000594  ARProt: 000  RData: 00000513  Operation# 535 at 21130 ns
%% Log   PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 536 at 21140 ns
%% Log   INFO      in manager_1,   Read Data: 00000513  Read Address: 00000594  Prot: 0 at 21140 ns
00000594: 0x00000513    addi     a0, zero, 0
%% Log   INFO      in manager_1,   Read Address.  ARAddr: 00000598  ARProt: 000  Operation# 537 at 21140 ns
%% Log   INFO      in memory_1,   Memory Read.  ARAddr: 00000598  ARProt: 000  RData: 00000073  Operation# 536 at 21150 ns
%% Log   PASSED    in manager_1: ReadResponse Scoreboard,      Received: 0   Item Number: 537 at 21160 ns
%% Log   INFO      in manager_1,   Read Data: 00000073  Read Address: 00000598  Prot: 0 at 21160 ns
00000598: 0x00000073    ecall
    *
PASS: exit code = 0x00000000 running test.exe

Register state:

  zero = 0x00000000   ra = 0x00000001   sp = 0x000015c0   gp = 0x00000001
    tp = 0x00000002   t0 = 0x00000002   t1 = 0x00000000   t2 = 0x00000001
    s0 = 0x00000000   s1 = 0x00000000   a0 = 0x00000000   a1 = 0x000015c0
    a2 = 0x00000000   a3 = 0x00000000   a4 = 0x00000001   a5 = 0x00000000
    a6 = 0x00000000   a7 = 0x0000005d   s2 = 0x00000000   s3 = 0x00000000
    s4 = 0x00000000   s5 = 0x00000000   s6 = 0x00000000   s7 = 0x00000000
    s8 = 0x00000000   s9 = 0x00000000  s10 = 0x00000000  s11 = 0x00000000
    t3 = 0x00000000   t4 = 0x00000000   t5 = 0x00000000   t6 = 0x00000000

%% DONE   PASSED   CoSim_iss  Passed: 988  Affirmations Checked: 988  at 21300 ns
simulation stopped @21300ns
Simulation Finish time 12:49:10, Elasped time: 0:00:13
Build Start time  12:48:51 GMT Mon Jan 23 2023
Build Finish time 12:49:10, Elasped time: 0:00:19
Build: Scripts_RunCoSimIssTests PASSED,  Passed: 1,  Failed: 0,  Skipped: 0,  Analyze Errors: 0,  Simulate Errors: 0
```

Here we see ISS disassembly output (coloured blue for clarity) alongside the logging from the OSVVM transactions and can see how the reading of instructions and store of bytes corresponds to the transactions in the OSVVM simulation. So the RISC-V software that's running on the ISS model is actually running out of the OSVVM `Axi4Memory` VC in the logic simulation and doing loads and stores to this memory as well. The ISS also has remote gdb debugging capabilities and so can be connected to an IDE and RISC-V software debugged as for any other program, with stepping, breakpoints, variable inspection etc., all from the logic simulation. The diagram below summarises the situation described here:

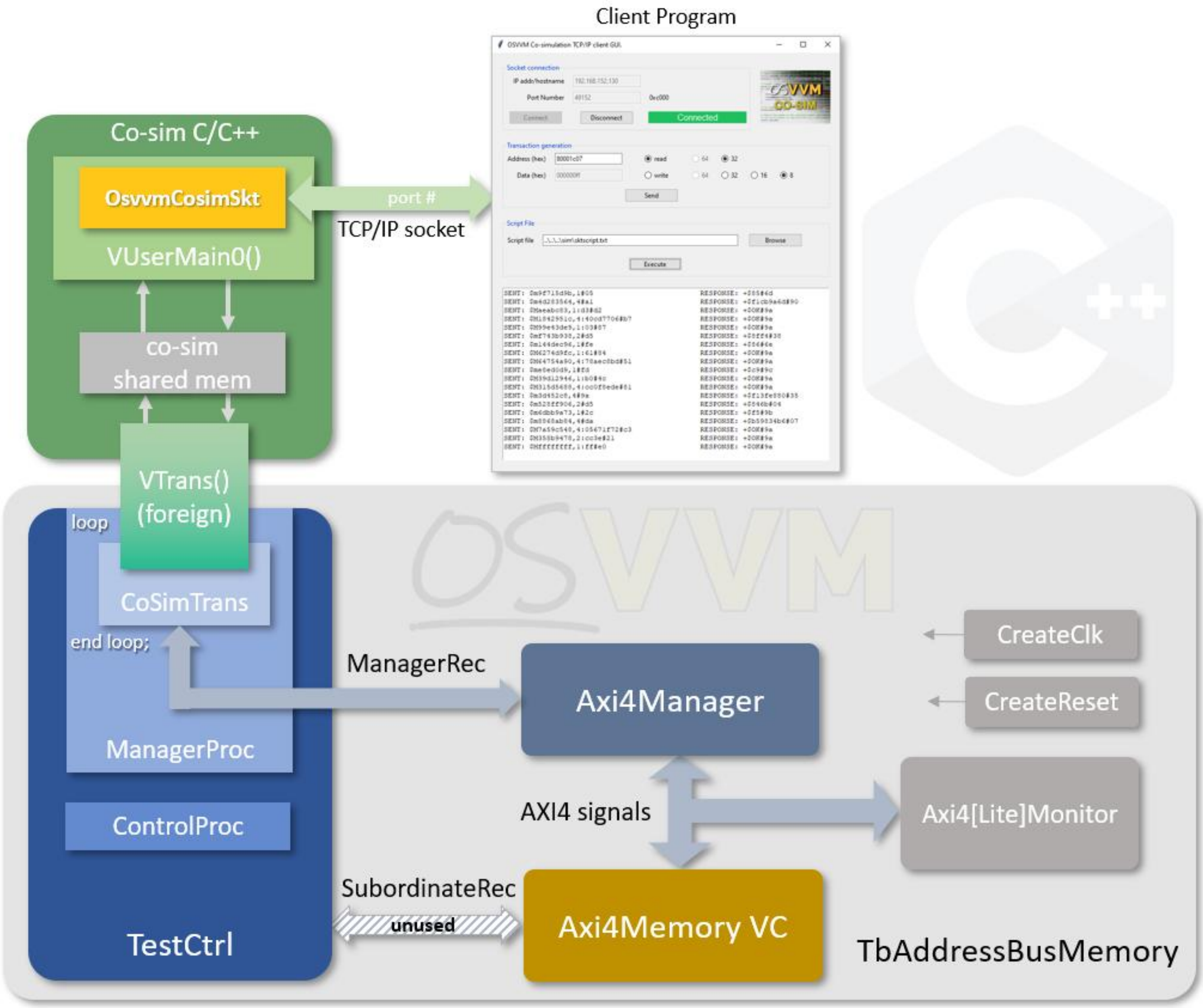


With any interrupt request status being passed into the `CoSimTrans` procedure calls the processor software can be interrupted (assuming its enabled). An example test for invoking an interrupt exists in the directory `CoSim/tests/interruptIss` using the `TbAddressBusMemory.vhd` test bench in `CoSim/testbench/`. Here the test can write to a set location in memory to invoke an interrupt request, allowing the RISC-V code to generate its own interrupt and check that it was indeed interrupted.

Hopefully it is not a giant conceptual leap to imagine a test environment where the `Axi4Memory` VC is replaced by some IP with memory mapped registers that can be accessed by software on the processor model to exercise its functionality, and that the code might even be the intended device driver for that block.

Connection to an External Program

The previous ISS example made the assumption that the system or processor model can be called from the `VUserMain0` code (or a sub-program of it). This might not be the case if the model we wish to use is a stand-alone executable such as a virtual machine (e.g., QEMU). Since we have a free hand in what code is written under the `VUserMain0` top level this can be anything we like, including inter-process communication software such as a TCP/IP socket, though any IPC method can be utilised. A simple socket class is provided in OSVVM as an example, located in `CoSim/code/OsvvmCosimSkt.cpp`. This allows remote connection via a TCP port and accepts communication that loosely follows the gdb remote protocol for reading and writing to memory. Sending read and write commands over the socket instigates transaction reads and writes on OSVVM, returning any read data responses as responses to the commands. The protocol does not yet support bursts or interrupts but could be extended to do so. It is meant as a demonstration of external connection rather than be a fully functional solution, as the protocols and requirements will vary wildly between real-world use cases. This use case, then might look something like the following.



```
SENT: $m9f715d9b,1#05                    RESPONSE: +$55#6d
SENT: $m4d283564,4#a1                    RESPONSE: +$f1cb9a6d#90
SENT: $Maeabc83,1:d3#d2                  RESPONSE: +$OK#9a
SENT: $M1842951c,4:40cd7706#b7           RESPONSE: +$OK#9a
SENT: $M99e63de9,1:03#87                 RESPONSE: +$OK#9a
SENT: $mf743b938,2#d5                    RESPONSE: +$ff4#38
SENT: $m144dec96,1#fe                    RESPONSE: +$66#6e
SENT: $M6274d9fc,1:61#84                 RESPONSE: +$OK#9a
SENT: $M64754a90,4:70aec8bd#51           RESPONSE: +$OK#9a
SENT: $me0ed0d9,1#f8                     RESPONSE: +$c9#9c
SENT: $M39d12946,1:b0#4c                 RESPONSE: +$OK#9a
SENT: $M315d5688,4:cc0f8ede#81           RESPONSE: +$OK#9a
SENT: $m3d452c8,4#9a                     RESPONSE: +$f13fe880#35
SENT: $m528ff906,2#d5                    RESPONSE: +$546b#04
SENT: $m6dbb9a73,1#2c                    RESPONSE: +$f5#9b
SENT: $m8868ab84,4#da                    RESPONSE: +$b59834b6#07
SENT: $M7a59c548,4:05671f72#c3           RESPONSE: +$OK#9a
SENT: $M358b9478,2:cc3e#21               RESPONSE: +$OK#9a
SENT: $Mffffffff,1:ff#e0                 RESPONSE: +$OK#9a
```

This arrangement uses a python program as a stand in for an external program with a socket interface. The test code is in `CoSim/tests/socket` with the python GUI program in `CoSim/Scripts/client_gui.py`. Essentially, though, this is the same arrangement as for any of the previously discussed examples—that is, some software makes calls to the C++ co-simulation API to generate transactions in the simulation that get translated from abstract accesses to bus specific protocols, such as AXI4. The same test could run on a different memory mapped bus protocols (e.g. Avalon) without modification. Whether the software is running a processor system modelled in C++ or being controlled via a TCP/IP socket connection to an external program, the simulation traffic is the same in nature. Also, because the traffic is generated in OSVVM, even if originating from software, all the advantages of OSVVM are available, with all the metrics, logs, and results that that brings.

Under the Hood

There's no space to go into full detail here about how the co-simulation software interfaces with a logic simulator such that a software program can appear to free run whilst interacting with a logic simulation that also appears to free run, but it will be instructive to give an overview to help understand what's going on and inform on efficient usage. This is really for those interested in the inner workings or those intending to interface with more complex software models, so feel free to skip this section if you would just be a user of the features.

To get to the end of the story first, it's all done with threads and mechanisms put in place to ensure that, actually, the threads aren't running concurrently but only one is ever unblocked at a time.

The `VUserMain`*n* entry points (as many as there are nodes) are each running in a separate thread from the simulator (which is the 'main' thread, if you like). These were set up and instigated when `CoSimInit` was called in VHDL for that node. These all need to be coordinated to exchange information between the C/C++ and logic simulation domains. When the simulation calls the `CoSimTrans` procedure for a given node it will block on that call. The corresponding `VUserMain` thread will free run until it makes a call to an `OsvvmCosim` transaction or tick method for the same node. The transaction information is loaded to a 'send' structure (one for each node) and the simulation call is unblocked, with the `OsvvmCosim` method now becoming blocked. The simulation side software now accesses the transaction structure and returns to `CoSimTrans` which generates the transaction in the simulation, advancing time, and at some point, makes another call to the `CoSimTrans` procedure again with any response information. The response is copied to a 'response' structure and the originally called `OsvvmCosim` method is unblocked and can access this and return the data and/or status. The `CoSimTrans` call is blocked once more, waiting for the next time a transaction or tick method is called from software (for that node). This happens for as many nodes that are used in the simulation and gives the illusion that software and simulation are all free running programs.

The details aren't important, but this is all done using node specific send and response structures as shared memory between the user threads and the simulation thread and synchronised with semaphores. The sequence described above ensures that only one thread is ever unblocked at any one time making the access of shared data perfectly safe. This is why, with the interrupt callback, it is perfectly safe to update state in a `VUserMain` program without risk of race conditions, as the callback is called from the simulation thread and thus the `VUserMain` thread will be blocked and cannot access the updated data until it is unblocked when the simulation calls `CoSimTrans` once more. It is important to note that between calls to `OsvvmCosim` transaction or tick methods simulation time is not advancing and the software is effectively running at infinite speed (as far as the logic simulation is concerned). This is not usually a concern, but if the simulation does need to advance between certain points in software execution the `OsvvmCosim` tick method can advance time without generating new transactions, and if the calls to `CosimTrans` are set up so that it is called at the clock rate whilst ticking, then time can be advanced accurately to the resolution of the clock rate.

Having explained the use of threads in the co-simulation workings and how they are, in fact, not free running, I want to mention that the software does allow multi-threaded programs to run and access the `OsvvmCosim` methods, and effectively share a given node's transaction interface. At the heart of the co-simulation software is an 'exchange' function which synchronises the threads and copies the send and response data in and out of the shared structures. This function is protected by mutexes on a per-node basis. Therefore, any threads using the same node will have atomic exchange of data to instigate a transaction (or tick) without fear of another thread stomping over this exchange.

There are probably a dozen ways the same functionality could be achieved using more modern libraries and methods. The PLI side of the software must be C (at least have C linkage), and so this simulation side code was chosen to be written in C. The origins of this technology also go back 20 years, and so some methods have probably been superseded since then. At any rate, this is all hidden from the user behind the VHDL `CoSimTrans` procedure and `OsvvmCosim` class methods, but for anyone interested in looking at the source code to see how it works, this gives a flavour of what you'll find.

Before we finish this section, with all this talk of threads, wouldn't it be easier if we could just call the simulation as if it were a function call and then we wouldn't need to muck about with threads. Most simulators that I have used do not provide any such feature, as the simulator is a free-standing executable rather than a library that can be linked to a user program. One exception though is GHDL. This provides a method for calling GHDL via a ghdl_main() function. GHDL comes in various flavours and the ‘mcode’ version does not support this. However, the others do and the OSVVM repository (under the CoSim sub-module) provides a demonstration test in CoSim/tests/ghdl_main. Here the user VHDL code, compiled into by GHDL into object files, are gathered into a library which can be linked with user code along with some GHDL libraries, into an executable. From the user code, ghdl_main() can be called to start the simulation. Unfortunately, the ghdl_main() call is blocking until the simulation terminates, so calling the OsvvmCosim methods isn’t possible from the same thread—so the call is made in a separate thread! There’s no getting away from it, I’m afraid. None-the-less, this is still useful as this can be linked with code that extends or forms part of another modelling system, such as QEMU, which is, itself, a stand-alone executable. It is a pity that other simulators don’t have this feature. This method is not strictly how the use of ghdl_main() is documented, as it is still expected that user code will be provided as a shared object to the simulation executable for dynamic loading. The OSVVM test compilation simply does the final link to allow the user code to be part of the executable that GHDL generates.

Conclusions

We have looked at the new co-simulation features of OSVVM and how they can be used to extend verification into the C++ domain with, ultimately, the ability to run software interacting on a system model with logic design on a simulator via the OSVVM TLM features. We looked at a couple of examples of using a RISC-V processor model and of connecting to an external program to drive transaction generation. The features are not limited to these examples and one can extend to C++/logic co-simulation to any level of sophistication. We also had an overview of how the co-simulation software works.

The emphasis on the OSVVM co-simulation features is to expose parts of the transaction layer modelling features of OSVVM in the C++ domain, such that a simple class gives access to those features from a user written C++ program. From that point it has been demonstrated that this opens up having logic simulation as an extension of software models of any complexity and of executing embedded software driving logic IP in one, pre-silicon, environment. This is seen as complementary to other techniques, such as FPGA emulation, but gives the ability of early co-development of logic and driver software, as well as a host of other possibilities.

What’s Next

The current release of OSVVM gives access to a sub-set (though a useful one) of the possible Address Bus Model Independent Transactions procedures. These are enough to generate transactions from bytes to burst accesses as atomic actions. It is planned to add support for the split transactions as well as for the checking flavours of reads etc. This will make it easier to write complete tests in the C++ domain as would be available from VHDL.

OSVVM also has model independent streaming functionality as well as address bus, and it is also planned to add support for this in co-simulation to give greater flexibility in system modelling with the co-simulation features. So watch out for these updates coming soon.

Part 2: Modelling Interrupts with Co-Simulation

Introduction

In this part of the document 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 the first part of this document 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:

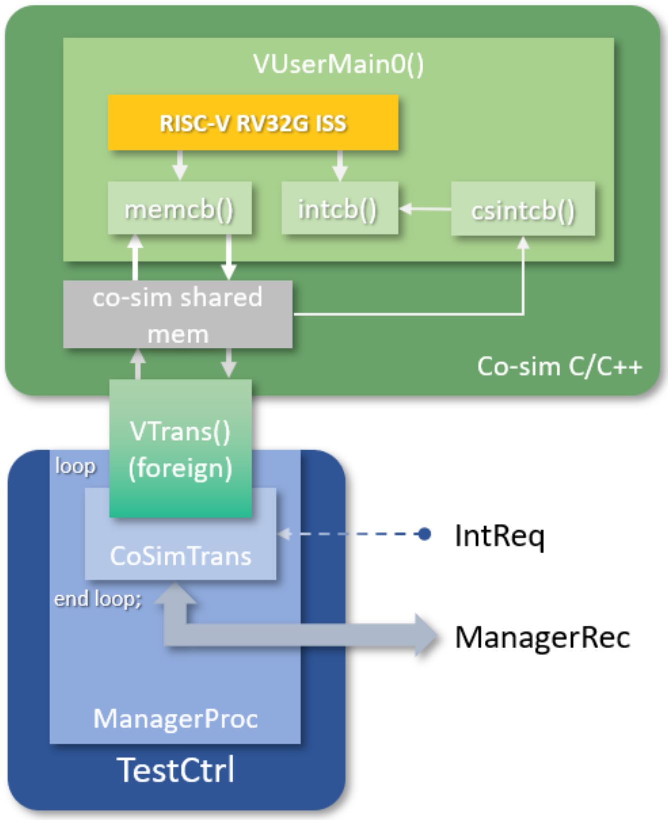
```
int csintcb (int int_req);

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 first part of this document 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.

Part 3: Using Multiple Transaction Nodes in Co-Simulation

Introduction

The OSVVM co-simulation environment has the ability to generate transactions from C++ to drive various virtual components (VCs), such as AxiBus, AxiStream, Ethernet, Uart, dual port RAM etc. But did you know that you can drive multiple VCs from within the same test bench, all driven by C++ software? To do this we have the concept of 'nodes'.

The OSVVM co-simulation documentation does document the concept of a node, and a unique node number for each co-simulation instantiation, though in a reference manual manner. In this article, though, I want to discuss in more detail what the nodes are, what they are used for, how to instantiate them in a test bench, and how the software for each node, in a multi-node system, can communicate and synchronise with each other and even be multi-threaded. We'll close up by taking a look at some ideas on how these facilities can be used to model embedded software architectures.

What is a Co-simulation Node?

In short, a co-simulation node is a source of transactions that has a unique node number. The OSVVM library provides a set of VHDL co-simulation procedures that can be called from a process to drive an OSVVM abstracted bus independent transaction interface signal. It's these interface signals that drive the verification components (VCs) which convert the high level, abstracted, transaction information into specific protocol signalling, such as AXI4 or Ethernet. As of OSVVM release 2023.05 the co-simulation VHDL procedures available are as listed below:

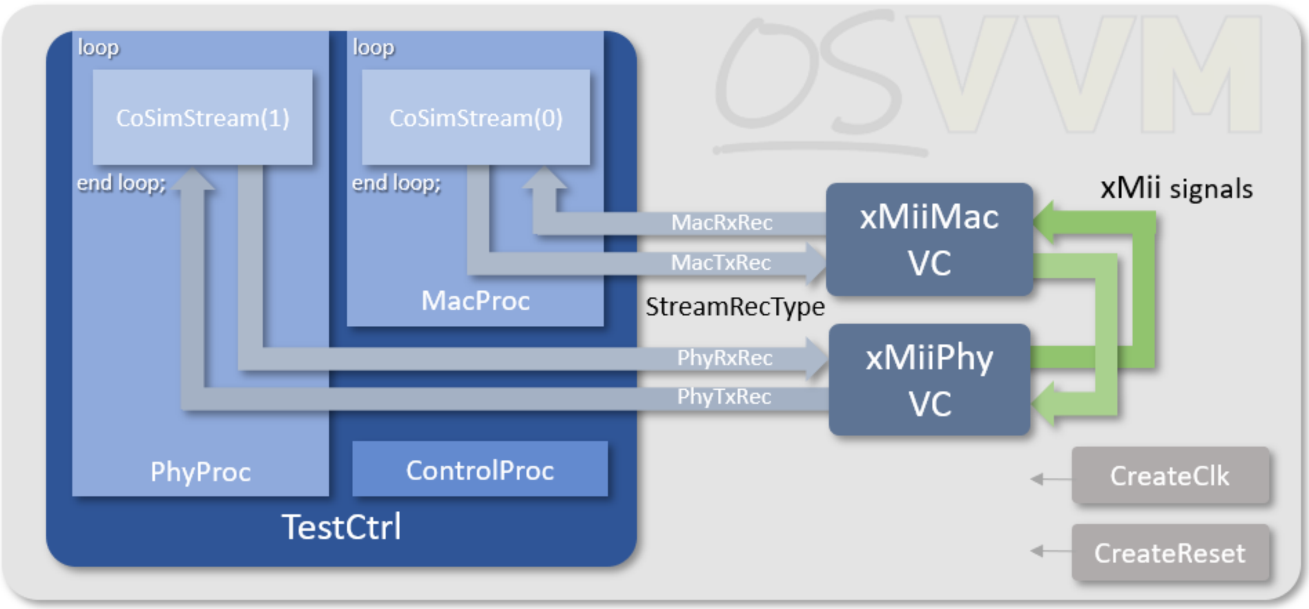
- `CoSimTrans:` manager transactor driving an `AddressBusRecType` signal
- `CoSimResp:` subordinate responder driving an `AddressBusRecType` signal
- `CoSimStream:` streamer driving `StreamRecType` signals

Each call to one of these procedures will generate either a new transaction or allow simulation time to tick by (or some other utility functions, which aren't relevant to this discussion).

One or more of these procedures can be called from within its own process, usually in a loop, to drive multiple bus model independent transaction signals attached to multiple VCs. Each co-simulation procedure used will have a unique node number (more later), and all can be the same procedure type, or a mix of any of the types, as appropriate to the test environment being constructed.

What can Multiple Nodes be Used For?

A couple of scenarios can illustrate how a multiple node environment could be used. For example, one might have two `CoSimTrans` co-simulation procedures driving two AXI4 VCs if, say, testing an AXI4 interconnect as the DUT, where the two `CoSimTrans` procedures are generating transactions, acting as if they are two processor cores. In another scenario, there might be one `CoSimTrans` and another `CoSimStream`, where the DUT is an Ethernet MAC, attached to an AXI bus VC to access its memory mapped registers, whilst the other is a `CoSimStream` connecting to an `EthernetStream` VC to receive and respond to the IP's traffic. You get the idea—there is no restriction on the mix of types. There is a slight restriction in that the default maximum number of nodes is 64, but even this can be overridden at compile time if necessary. The diagram below shows a multi-node set up for one of the OSVVM co-simulation tests—in this case for driving ethernet VCs.



For reference, this example is found in the files in `CoSim/testbench/TbEthernet`, and `TestCtrl` in `CoSim/testbench/TestCases_Ethernet`. The `TestCtrl` architecture, where the `CoSimStream` procedures are called, is defined in `TbStandAlone.vhd`.

Node Number

The node number is critical in delineating the various calls to OSVVM co-simulation procedures. All of the OSVVM procedures take a `NodeNum` argument, and this connects the calls to that procedure with the user software that is driving the generation of transactions. Each procedure that drives a given model independent signal (or signals, if the connected VC requires them) must have a unique node number, and must also be consistent between calls to that procedure—no dynamically changing the `NodeNum` between calls. This ensures that the user software running for a given node is the only source for transactions to a particular VC.

Initialisation

The transaction procedures connect user software for a given node number to a particular program. If a transaction procedure uses a node number of 0, then the user is expected to supply a C function (or one with C linkage, if compiled in C++) that is called `VUserMain0()`. Likewise, a node number of 1 requires a function called `VUserMain1()`, and so on for each node number used. You can think of these as the `main()` program running on a 'virtual processor' core, with a separate core for each node. Access to generating transactions is provided with one of the API classes, such as `OsvvmTrans` or `OsvvmStream` (more later). At construction, these are assigned the matching node number to link it to the VHDL co-simulation procedure.

These user programs, however, must be executed to start generating transactions, and so an OSVVM co-simulation procedure, `VInit(NodeNum)`, is provided to do just that. It takes a single node number argument, does some internal initialisation, and will start the user code for that node running. It *must* be called before any call to a co-simulation transaction procedure is called that uses that same node number, or the simulation is likely to hang. In addition, if it is called with a node number that does not have a matching `VUserMain`*`N`* program, then an error is likely to occur.

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.

A set of API classes are provided to match the type of transactions being used by the co-simulation VHDL procedures to drive a VC. The current classes supported as of release 2023.05 are:

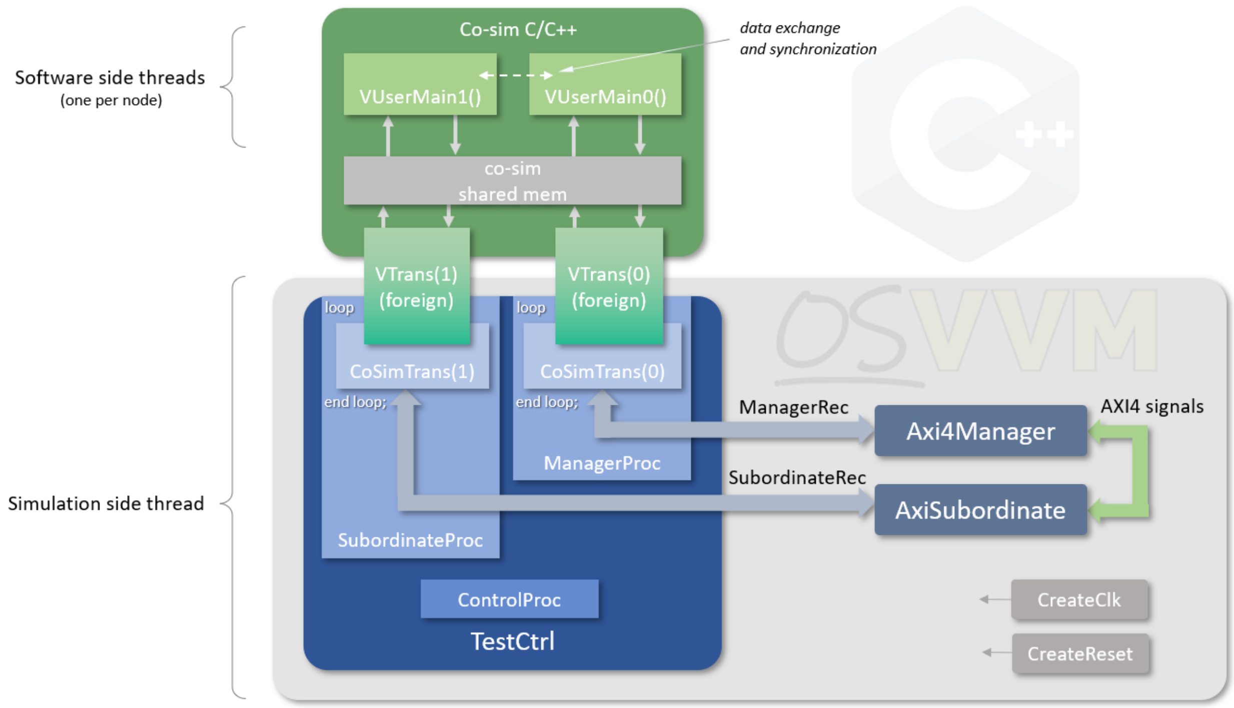
- `OsvvmTrans:` manager transactor API class
    - `OsvvmTransInt:` manager transactor API class with nested interrupts
- `OsvvmResp:` Subordinate responder API class
- `OsvvmStream:` Stream API class

Use of these classes is documented elsewhere (see here and in the first part of this document), so I won't go into details but, suffice it to say, the user code generates transactions with calls to the methods provided by these classes, as well as means to advance simulation time without generating a transaction.

So, we have multiple nodes, each running a user program, as if running on its own processor. How?

Threads Under the Hood

Relevant to our discussion, it is worth having a look at what's going on under the hood that allows us to have multiple running user programs. Without going into a lot of detail, each user program is running in a separate thread—and these threads are separate from the simulator executable's 'thread', which is just its main program flow. (I'm sure it will have multiple threads within its execution but, as an executable, it has a single 'main' entry point, just like our co-simulation user programs.) The OSVVM co-simulation takes care of managing the exchange of information between the user programs and the simulation and keeping this all synchronised. The diagram below summarises the setup for one of the tests in the OSVVM co-simulation test suite:



For reference the files are in `CoSim/testbench/TbAxi4_Responder`, with the `TestCtrl` architecture file in `CoSim/testbench/TestCases/TbAb_Responder.vhd`.

Here we see two processes calling `CoSimTrans` procedures with a node number of 0 for the manager and a node number 1 for the subordinate. Each calls an underlying foreign procedure (`VTrans`) with the node number to connect to the C/C++ domain.

The OSVVM co-simulation software manages the exchange of this information via some shared structures in memory (one set for each node). As shown, each of the `VUserMain`*`N`* functions are running in a separate thread, which are separate from the simulator program main program thread.

Communication Between User Programs

Now it happens that the synchronisation mechanisms that have been put in place in the OSVVM co-simulation software (not coincidently) ensure only one of the user threads or the simulation are running at any given time, whilst the others are blocked. As well as ensuring safe communication between the user program threads and the simulation, it also has the benefit in that it allows us to have safe communication and synchronisation between the user programs as well, and without the need to use thread synchronisation methods in the user code. There are some things we must take care of, but these are just programming requirements, and not 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 co-simulation user code threads—accept now that has already been done by the co-simulation software. Therefore, any data can be constructed for reading by another user program safely. Not until a new call to a transaction API class method is made by the user program that's generating the data for use by another user program, will that user thread be de-scheduled, and the target program have a chance to be activated and read the 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. An example use of such an exchange comes from the example of the diagram above, where there is a transactor and responder thread. Here it is useful to send information of the data it used to generate a transaction directly to the responder thread which can then check the validity of the data it actually received.

Synchronisation

As well as exchanging data between user programs, it is useful if they can be synchronised to co-ordinate that data exchange. For instance, taking the example from above, the responder software might want to control when the transactor software sends the next set of data so that a send-receive-check scenario is established for each individual test point. Without any synchronization, the transactor thread will simply run through its program, sending transactions at arbitrary times relative to the responder.

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 variable whenever it wants the transactor to advance to its next transaction generation. The transactor 'waits' on the barrier variable to increment before advancing. This is the situation used in the test program in `CoSim/tests/responder.cpp` that runs on the test environment as shown in the diagram above.

So, what do we mean by 'wait'. As mentioned in the co-simulation documentation, a user program cannot loop internally on some state that it relies on 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 if simulation time is not advanced. In our example for transactor/responder, the transactor 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. This is what the test code does via a local function:

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

static void waitOnBarrier(void) {
    OsvvmCosim  cosim(node);

    while (barrier <= last_barrier)
        cosim.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 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, as we've seen, each 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 it is a single thread. Is this a limitation required by the co-simulation environment?

Actually, no. 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 co-simulation software layer. At the heart of the co-simulation 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 mutexes so that it becomes an atomic operation. There is a mutex 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 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.

Accessing Different Nodes from a Single Program

We've already seen that we can drive different transaction interfaces using separate nodes running separate user programs, but it is still possible to have multiple threads from a single program that accesses different transaction interfaces, each on a unique node. This more closely matches a muti-threaded software environment where there is a single main program that spawns multiple threads.

In order to drive a second node a `CoSimInit` call must still be made in VHDL for this additional node (just as for the simpler cases), and this will require an equivalent `VUserMain`*N* function. Now, though, the user function has very little to do and just needs to raise a flag that it has been called (see the discussion on barriers above) and then it can exit. A scenario might be that from within two separate VHDL processes, `CoSimInit` is called for each node, just as for ordinary cases. Let's say that we are using node numbers 0 and 1, and that the code running for node 0 is to be the main code. When VUserMain0 is called, it will do its initialisations and then start a new thread (with, for example, a function called `RxDriver`) for node 1's transaction interface. This new thread must now wait on a barrier. At some point (which could be before or after `VUserMain0` is called), `VUserMain1` is called. All this must do now is increment the barrier and then it can return. The `RxDriver` code running in the new thread is released and can then generate transactions for that node's transaction interface using an API class with a matching node number. Meanwhile, once `VUserMain0` has spawned a node 1 thread, it is free to generate transactions on the node 0 interface, perhaps by calling a sub-program `TxDriver`. Note that the two driver functions, unlike having separate node programs, are free running threads, so both the `RxDriver` code (in our example) and the `TxDriver` code, if they need to share data, must follow all the thread safe precautions, and the co-simulation software layer will not take care of this in this scenario. Therefore, this setup is seen as an advanced configuration for allowing modelling of, say, real-time multiple threaded embedded software, and to be used by those who are confident in its safe implementation.

An example use for this scenario might be driving a stream bus model independent interface connected to an AxiStream or Ethernet VC. The `CoSimStream` co-simulation VHDL procedure allows for connection to both a transmitter and receiver `StreamRecType` signal, and this is ideal for the previous case where we had a single node driving Tx and Rx with different threads within the node's user program. With this new configuration, though, the `CoSimStream` procedure can be called (in a loop) from separate processes, with different node numbers, where one of the interfaces (either the Tx or Rx) is connected to a dummy signal. Now, the `TxDriver` thread can drive a TX interface of one node, and the `RxDriver` thread can drive the RX interface of the other node. Again, this can be achieved with separate user programs, called from, say, `VUserMain0` and `VUserMain1`, but this configuration maps to a single multi-threaded program to allow easier mapping to real-time embedded software architecture with simple setup and initialisation as described. The usual API access class for the software is via the `OsvvmCosimStream` class, which provides both Tx and Rx methods. However, if now separating these functions, then one of two supplied derived classes can be use as appropriate (either `OsvvmCosimStreamTx` or `OsvvmCosimStreamRx`) which limit the available methods to only the relevant functionality to prevent accidental calling of the inappropriate methods which would have undefined results.

Using a Separate Node for Interrupts

OSVVM co-simulation provides for modelling interrupts (see the second part of this document on interrupts for details), with the `CoSimTrans` procedure having an `IntReq` input argument, and the C++ API of the `OsvvmCosim` class providing a `regInterruptCB` method which allows a user program to register an interrupt callback function. This is called whenever the `IntReq` argument of a call to `CoSimTrans` changes state, passing in the new `IntReq` value to the user's callback function. In the interrupts blog it was mentioned about interrupt resolution and the need for not 'ticking' for extended periods of time (some of which is mitigated by the use of the `OsvvmCosimInt` class, described in the blog). The interrupts are not lost, but the latency before the code can handle the change in interrupt state might be artificially extended if care isn't taken.

One way around this is to have a separate `CoSimTrans` in its own process with a unique node number, wiring off the transaction signal and control outputs, and simply using the `IntReq` and `NodeNum` inputs. However, a convenience procedure is also provided, called `CoSimIrq`, with just the two `IntReq` and `NodeNum` inputs. Since this is not connected to transaction signal it has no concept of time and calls to a tick method in the C++ API do not influence the advancement of the simulation, and so would be used in a loop (just as for `CoSimTrans` or any of the co-simulation procedure calls), but with time advanced externally within that operational loop. This is done so that just when the procedure is called is completely up to the implementation, and also so that it *cannot* be blocked with a 'tick' call with a long delay. This could be by simply having a "`WaitForClock(ModelIndependentRec, 1)`" call in the loop, to using a "`wait on IntReq`" to only call `CoSimIrq` when the interrupt signal changes or, indeed, whatever convenient method is required, so long as time is advancing between calls to the procedure.

Whether using `CoSimTrans` or `CosimIrq`, the associated user program, `VUserMain`*`N`* need only register an interrupt callback function and then `SLEEPFOREVER` (a macro provided via the `OsvvmTrans` class header). The callback function, when activated, can then update interrupt state, and make it available to other user programs and/or threads. A simplified code snippet of this arrangement is shown below.

```
static uint32_t irq        = 0;
static int      irqBarrier = 0;

// Event processing
extern "C" void VUserMain0()
{
    OsvvmCosim mgr(0);

    // Event loop
    while (true)
    {
        waitOnIrqBarrier();
        processIrq(intReq, mgr);
    }
}

// Interrupt callback function
int procIrq(int int_vec)
{
    irq = int_vec;
    irqBarrier++;
}

// Interrupt node program
extern "C" void VUserMain1()
{
    OsvvmCosim int(1);

    int.regInterruptCb(procIrq);

    SLEEPFOREVER;
}
```

In this example the call to `processIrq` hides a lot of detail. At its simplest it would make calls to methods that generate transactions to service the interrupts, but it might also be within a multi-threaded implementation, as discussed earlier, posting event messages to different threads to handle the specific classes of events and be the source of the generation of transactions. This architecture would make the event loop non-blocking and so will log all incoming events, including edge triggered interrupts of a single cycle. The software can then set a pending bit for the interrupt, which is cleared when it gets serviced. The interrupt node program might take care of this, providing means to clear the pending state, but a better way could be in a software model of an interrupt controller if co-simulating, for example, within a software model of an SoC system, with the controller having configurable registers to mark interrupt relevant inputs as edge triggered and thus register, and hold, pending status internally.

With this kind of arrangement, the interrupt state within the software will be updated regardless of whatever activity is happening on any transaction interface—even within the middle of an active transaction.

Modelling an Event Based Software Architecture

Using the separated interrupt handing node, an event based software architecture can be modelled that has an event loop. The interrupt node software is constructed as described above to simply receive the changes in interrupt state and make it available. 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 associated with a transaction generating node, such as a `CoSimTrans` based loop for address bus model independent transaction generation, 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. 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 transaction calls to handle the interrupt. This main program loop, then, is the event loop servicing the incoming events on the interrupt lines.

Conclusions

In this article we had a look at the use of multiple nodes to drive more than one source of model independent transactions, looking at how to add and use this to the OSVVM VHDL co-simulation environment, and what this means in terms of user code.

We saw that the user code for each node run as separate threads, but that the co-simulation layer co-ordinates this in such a manner that data exchange is safe between the user programs, without further thread requirements on that program, and that even synchronisation between the user code is possible with emulating a simple barrier variable.

We also looked at how the co-simulation environment can even allow a node's user program to be multi-threaded, if so required, and accesses to transaction generation from multiple threads, via the APIs, is thread safe. Becoming more advanced, we looked at driving multiple nodes from a single user program, with its own multiple threads. We finished up by looking at having a separate interrupt node, and how this might be used to model an event loop. Thus, the OSVVM co-simulation environment provides the facilities for any level of user code sophistication, from simply writing linear, single threaded, test code to drive a DUT, to co-simulating with multi-threaded, event based, embedded software models.

Part 4: Using External Libraries

Introduction

The OSVVM Co-simulation features allow user supplied C or C++ code to drive model independent transaction (MIT) signals in a VHDL logic simulation via a supplied API, as I've discussed in a previous blog. When combined with one of the supplied Verification Components (VCs) or with a custom VC, the user software can generate protocol specific signalling to drive a design under test RTL implementation. It is also possible to have multiple user programs driving separate interfaces for as much complexity as needed. The user software is not running as separate processes with some inter-process communication links but is loaded by the logic simulator and run in threads as part of the simulator's normal threading environment making it very fast to execute.

The user programs that can be run in the simulator can be pure test code and mirror what might be done in VHDL, but any kind of program can be written and the "OSVVM's Co-simulation Framework" document gives two examples. The first of these details interfacing a RISC-V instruction set simulator so that RISC-V programs can be run and be debugged whilst driving memory load and store transactions into the logic simulation to access memory and memory mapped RTL components' registers etc. Another example is implementing a TCP socket server to interface with externally running programs to generate transactions.

The OSVVM co-simulation API maps the model independent transaction VHDL procedure calls to equivalent C or C++ calls, to give the same rich environment for driving the signalling as for VHDL test code, but also allows the ability to use any C or C++ library that may be useful to do so, giving access to all the computing power any normal C or C++ program access on the host machine. In this blog I want to discuss how external code and libraries can be linked to the user supplied test code so that any arbitrary program can be run with OSVVM.

How the Normal User Code is Run

Before looking at linking to external code, it is worth reviewing how straightforward user test code is compiled and run within the OSVVM co-simulation environment. I'll assume a constructed VHDL test environment already exists that makes calls to co-simulation VHDL procedures—`CoSimInit` (in all cases; one for each MIT being driven, each with their own unique "node" number and at least one from `CoSimTrans`, `CoSimResp` or `CoSimStream`. For each "node" (supplied as an argument to the VHDL procedures) as a minimum, the user must supply a "main" entry point in the form of VUserMain<*n*>, where <*n*> matches the node number. These must also have "C" linkage. From that point on any normal C or C++ code (as appropriate) can be written and use the supplied API. So, an example template for some user C++ code for a node of 0 might be:

```
#include "OsvvmCosim.h"

extern "C" void VUserMain0(int node)
{
    /////////////////////////////////////////////
    // User code and calls to other functions
    /////////////////////////////////////////////

    // If ever got this far then sleep forever
    SLEEPFOREVER;
}
```

The SLEEPFOREVER call at the end just ensures the logic simulation can continue even though this code does nothing.

To compile this "program" let's assume it is in a file called VUserMain.cpp in a directory called tests/basic. From within an OSVVM TCL script environment, the code can be compiled with the MkVproc procedure:

```
ChangeWorkingDirectory ./tests
MkVproc basic
```

The MkVproc procedure takes a single argument specifying the directory where the source code is to be found. Since the script already changed to the tests directory, this argument is just basic. All the C and C++ code in the specified directory will be compiled to object files, with the directory also added to the include search path. Therefore, any amount of C or C++ source code, with any organisation, can be added to the directory to be compiled.

Once all the code is compiled to object files, these are all linked into a static library, libvuser.a, and then linked into a shared object (or dynamic linked library in Windows speak) VUser.so. The script will also compile the OSVVM co-simulation software into a libvproc.a library which is linked into a shared object, VProc.so. When the simulation is run it loads the VProc.so library to interface with the logic, and the VProc.so code loads the VUser.so library to start running the user code. As a side note, the reason there are these two separate shared objects rather than one big one, is that the compilation of the OSVVM co-simulation code is simulator and operating system dependent, with various flags and definitions required, depending on the environment. With a separate user library, its compilation is independent of the OS and simulator used, simplifying its compilation as a separate step.

Source Code

If third party code, that requires to be included in the user code, is in source-code form, then it could be added to the compilation directory and will automatically be compiled with the VUserMain code. If this is not practical, then the external source code can be compiled separately into object files which, after a `MkVproc` compilation can be appended to the `libvuser.a` static library. The compilation of `VUser.so` would then need to be re-done to include the new object files. E.g.

```
ar q libvuser.a <objfile1> <objfile1> ...
make -f <path to OsvvmLibraries>/CoSim/makefile SIM="<SimName>" VUser.so
```

Here the `makefile` in the `CoSim` directory is being used directly, rather than called from `MkVproc`. If not using ModelSim, then the `SIM` make variable needs to be set to an appropriate other simulator name (selected from GHDL, NVC, RivieraPRO or QuestaSim) to do an x64 link rather than an x86 32-bit link. A separate make file, `makefile.avhdl`, is supplied for Active-HDL, but it's used in the same way.

Linking Prebuilt Static Libraries

To specify a static library to compile along with the user code, then a second `MkVproc` argument can give the library core name. E.g.

```
ChangeWorkingDirectory ./tests
MkVproc iss rv32
```

This compiles the user code in `./iss` and links the RISC-V ISS library from `CoSim/lib`, with an include path for `CoSim/include`. The supplied library name will be expanded depending on OS and simulator type, with a lib prefix added and a suffix of `[win[32|64]][<blank>|x64]`. For example, if using GHDL, it's expanded to `librv32x64.a`—there are 32-bit versions for ModelSim and variants for both Linux and Windows under MSYS2. This scripting, though, relies on libraries and required include files placed or linked into the `lib/` and `include/` directories of the `CoSim/` directory.

Using makefile for External Libraries

If more complex compiling and linkage is required then the `makefile` (or `makefile.avhdl`, if using Active-HDL) in `CoSim/` can be used or copied and modified to add include search paths and link to external libraries, both static or dynamic as required. Normally, the call to the make file compiles everything in one step. If we break this into three steps, then there's a chance to add customised flags to compile in external libraries. Assuming nothing has been compiled so far, then the `VProc.so` shared library must be compiled first:

```
make -f <path to OsvvmLibraries>/CoSim/makefile \
    SIM="<SimName>" \
    PCIEDIR=<path to OsvvmLibraries>/PCIe \
    SRCDIR=<path to OsvvmLibraries>/CoSim/code \
    VProc.so
```

This is standard, so no customisations are required, but the paths to the relevant OsvvmLibraries sub-directories are needed as shown—either relative to the compilation directory or absolute paths. Note, again, that `SIM` must be set for the appropriate simulator. The next step is to compile the user static library `libvuser.a`:

```
make -f <path to OsvvmLibraries>/CoSim/makefile \
    SIM="<SimName>" \
    SRCDIR=<path to OsvvmLibraries>/CoSim/code \
    USRCDIR=<path to VUserMain test code directory> \
    USRFLAGS="<my flags>" \
    libvuser.a
```

`SRCDIR` needs specifying again to pick up the headers for the API. The `USRCDIR` is specified to compile all user test code, which must include the `VUserMain` entry C functions. In addition, `USRFLAGS` can add any gcc or g++ flags specific to the user code, including for headers of libraries to be linked or paths to libraries etc. The last step is to link the code for `VUser.so`:

```
make -f <path to OsvvmLibraries>/CoSim/makefile> \
    SIM="<SimName>" \
    USRFLAGS="<my flags>" \
    VUser.so
```

In this last step `USRFLAGS` is now used to add any library paths and library references in order to link to the desired external pre-built libraries. We now have `VProc.so` and `VUser.so` and we can skip the `MkVproc` step in the test script and simply run the simulation which will load the shared libraries and execute the programs using their external libraries.