OSVVM's Co-simulation Framework

User Guide for Release 2023.01

By

Simon Southwell

simon.southwell@gmail.com

Table of Contents

1 Overview

The co-simulation features of OSVVM are a compliment to the Address Bus Model Independent Transactions features, as outlined in [1]. It is highly recommended that this document is read before this one, as it will be assumed that the reader is familiar with the transactions associated with that feature.

A typical address bus transaction test bench might look something like the diagram below for an AXI4 environment.

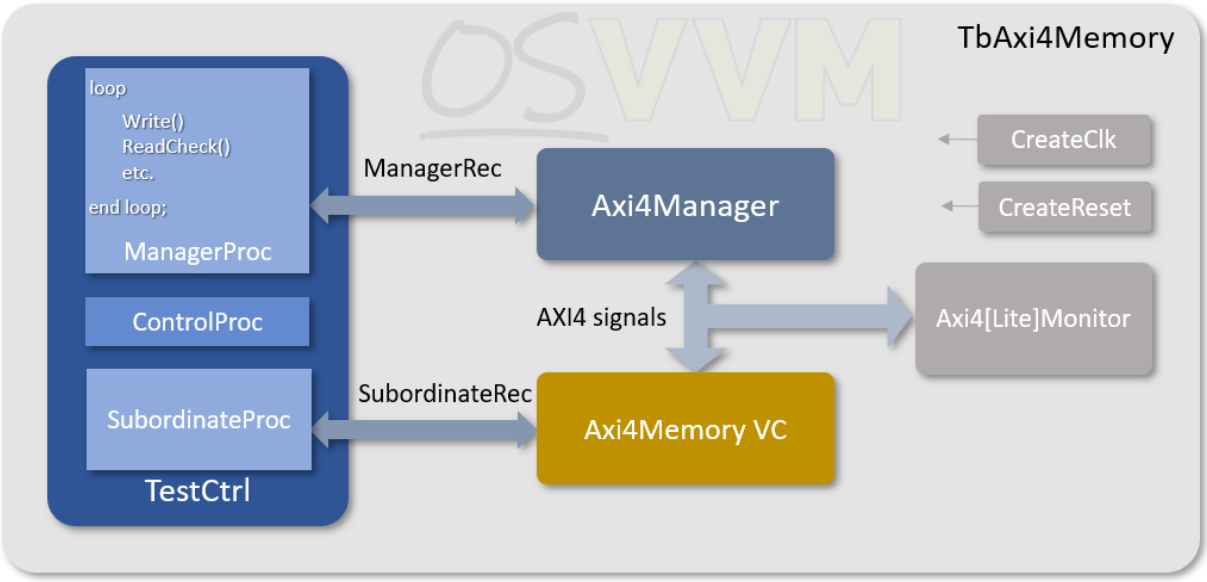


Figure 1: Address Bus Transaction Test Environment

In this example the test has a manger process using calls to to the address bus procedures, within an operation loop, to generate transaction via the `ManagerRec` which the `Axi4Manager` VC component translates into specific AXI4 signalling. In this case the target is another OSVVM VC, which is a memory component but would normally be the DUT component or sub-system. A subordinate process inspects arriving traffic and performs checks. In some scenarios the subordinate process generates the responses but, in this case, the `Axi4Memory` VC generates these internally. The advantage of this system is that the transactions are abstracted away from the bus specific model with re-usable components. If the `Axi4Manager` VC is replaced with another address bus VC, then the manager code remains unchanged and can drive the new system

The aim of the co-simulation features is to extend this architecture to allow the generation of the transactions to be performed within a C/C++ domain. It extends the calls to the address bus procedures to calls to methods within a C++ class. This opens up new possibilities not availabe in a pure VHDL domains. Some of these are listed below:

- Writing of tests in C++
  - Test development environment extended to software engineers
- Running logic concurrently with C++ models
  - Instruction set simulators or cycle accurate processor models

  - System models with a mix of software models and logic IP
- Connection to an external program via TCP/IP sockets
  - Where model might not be callable directly from the C++ code

With co-simulation of processor and system models, the possibility of developing software alongside the logic IP exists before silicon is available, but using both that actual hard and soft components together and not facsimiles, reducing risk.

The co-simulation features are added at the point of generation of `ManagerRec` transactions with calls to provide co-simulation procedures to abtract away all details of the co-simulation infrastructure. The diagram below shows the same environment above, only using the co-simulation features:

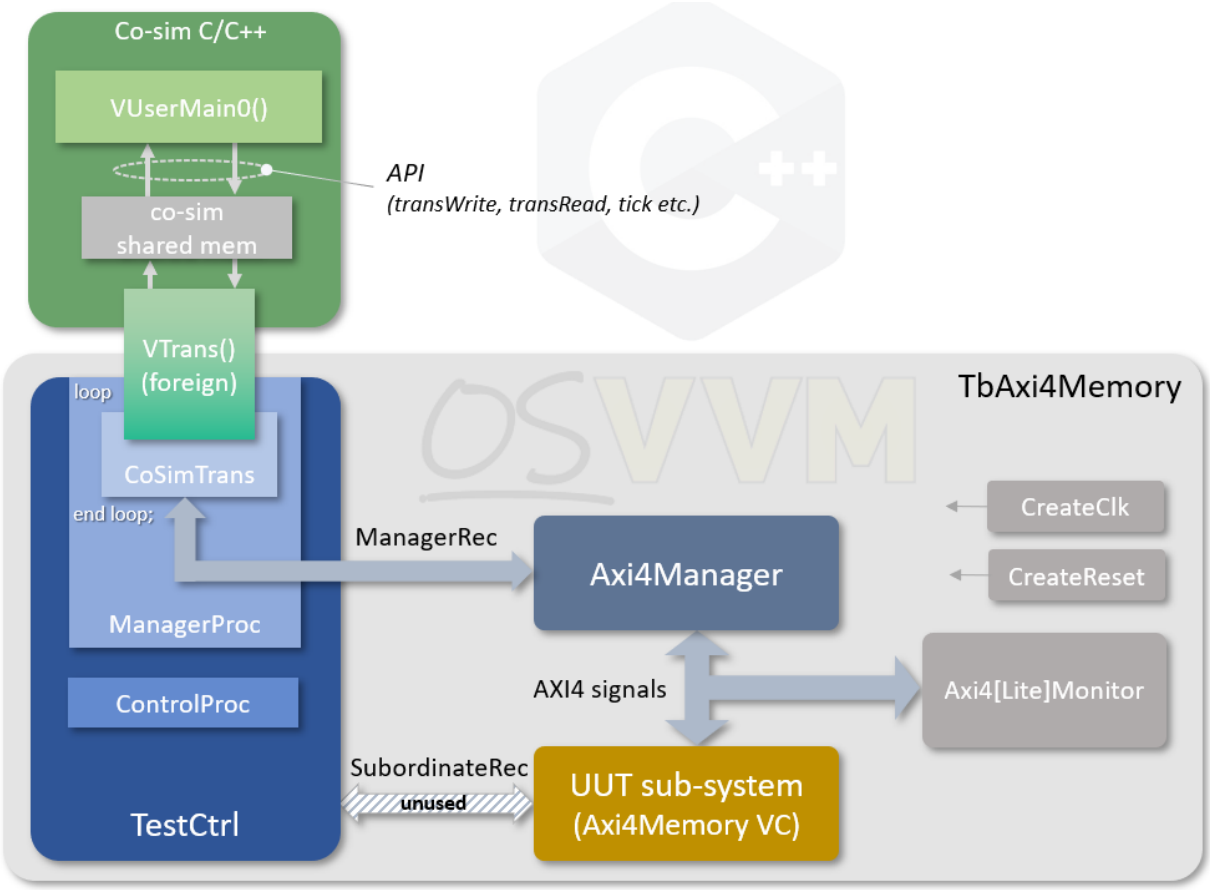


Figure 2: Co-simulation Environment

In place of calls directly to the address bus transaction procedures, a `CoSimTrans` procedure is provided to replace these and drive `ManagerRec` instead. This hides deatils away for calls to the the underlying co-simulation code, with calls to foreign procedures that communicate with user program. The code, at startup will call a function (`VUserMain0` for 'node 0', for example) that the user provides with all their code and sub-code called from there (much like `main()`). The user code now has access to the transaction API provided by the co-simulation code, and can generate read and write transactions in the OSVVM domain. The complexity of the user code to model other system features is now unlimited, and

can direct address bus accesses towards the simulation, and components modelled there via the API as appropriate.

The main features, then, of the co-simulation C++ code are as follows:

- Ability to instigate single word (and sub-word) read and write transactions
  - Split transactions not yet supported
- Ability to generate burst read and write transactions
  - Split transactions not yet supported
- Multiple nodes to drive separate `AddressBusRecType` transaction signals from multiple processes
- Ability for user code to register a callback function, called when interrupt state changes in the simulation.
- Ability to advance simulation time (without a transaction)

In the rest of this document will be detailed the usage of the co-simulation features to drive address bus transactions from the VHDL/OSVVM side and the C++ side.

2 Prerequisites for Windows

To run the co-simulation features on Windows requires the use of MSYS2 and the gcc toolchains for 64- and 32-bit compilation, as well as `make` and `python3`. The list below provides a link to the MSYS2 installer. Once installed, the required gcc modules can be installed as shown.

- msys2
- mingw-w64 32- and 64-bit gcc toolchains, with required non-included libraries
- From an MSYS2 MINGW64 shell:
  - `pacman -S mingw-w64-x86_64-gcc`
  - `pacman -S mingw-w64-i686-gcc`
  - `pacman -S mingw-w64-x86_64-dlfcn`
  - `pacman -S mingw-w64-i686-dlfcn`
- Utility functions
- From an MSYS2 MINGW64 shell:
  - `pacman -S make`
  - `pacman -S python3`
  - `pacman -S rlwrap (for GHDL and NVC)`
  - `pacman -S mingw-w64-x86_64-tclib (for GHDL and NVC)`

Note that **ModelSim** requires running from the MSYS2 MINGW32 shell, as it is a 32-bit executable and the co-simulation code must be compiled for 32-bits, whereas all the other simulators require using the MSYS32 MINGW64 shell for 64-bit compilation.

If **GHDL** is required to be used, the `mingw-w64-x86_64-ghdl-llvm` package must be installed with `pacman` and should be run from the MSYS2 MINGW64 shell. If must also be ensured that `tclsh` is available (`pacman -S tcl`).

3 Running the Co-simulation Examples

See the steps in the OSVVM Overview guide for running the OSVVM demos. Do the steps shown in Figure 8 in your simulator. `StartUp.tcl` needs to be sourced each time you start the simulator. See the Script User Guide for additional details for Aldec's ActiveHDL, GHDL, Synopsys VCS, and Cadence Xcelium.

```
cd sim
source ../OsvvmLibraries/Scripts/StartUp.tcl
build  ../OsvvmLibraries/OsvvmLibraries.pro
build  ../OsvvmLibraries/AXI/RunAllCosimTests.pro
```

Figure 3. Compiling and Running Co-simulation tests

4 Co-simulation Supported Platforms

Below is a list of simulators and the tested support for them for the co-simulation features of OSVVM.

Vendor	Simulator	Linux	Windows
Siemens	ModelSim	✓	✓
Siemens	Questa	✓	✓
Open Source	GHDL	✓	✓
Open Source	NVC	✓	✓
Aldec	ActiveHDL	N/A	×
Synopsys	VCS-MX	×	×
Cadence	Incisive	×	×
Cadence	Xcelium	×	×
AMD	Vivado	×	×

Table 1: Supported Simulators

5 OSVVM Co-simulation Address Transaction Procedure

To add co-simulation to the VHDL test logic in OSVVM procedures are provided in `OsvvmTestCoSimPkg`. The procedures stand in for address bus transaction generation logic such as that demonstrated, for example, in the `AXI4/Axi4/TestCases/TbAxi4_MemoryReadWrite.vhd` test. The co-simulation procedures use the standard OSVVM test environments driving the `AddressBusRecType` records, usually from a manager process, to instigate bus reads and writes on, for instance, AXI4 or Axi4Lite

busses, as dictated by the test bench used. The instigators of the transactions are then from C++ programs compiled and run with the simulation instead of direct calls to the OSVVM Address Bus Transaction procedures [1]. The co-simulation demonstration programs use the `TbAxi4Memory.vhd` test benches fro Axi4 and Axi4Lite.

Before any calls to co-simulated code can be made, the code must be initialised for *all* nodes to be used. Separate programs can be run concurrently if needed (though not usual), and each must have a unique node ID. This is done using the `CoSimInit` procedure, as shown below.

```
------------------------------------------------------------------
procedure CoSimInit (
------------------------------------------------------------------
  variable NodeNum          : in     integer := 0
) ;
```

The procedure would normally be called from within the `ControlProc` of the test case, or possibly from within the `MangerProc` if only a single node used, before any calls made to other co-simulation procedures.

To generate address bus transactions from the co-simulation software repeated calls to a procedure `CoSimTrans` must be made from a process (in a loop for example) that drive an `AddressBusRecType` signal. This would normally be from a test case's `MangerProc`, but multiple driving processes are allowed and each can make calls to `CoSimTrans`, so long as each uses a unique mode ID and that node has been initialised with a call to `CoSimInit` (see, for example, `Common/TbInterrupt/TestCases/TbAxi4_InterruptCosim1.vhd`). The `CoSimTrans` procedure is defined below:

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

The main argument to the procedure is the `ManagerRec` that is used to send a transaction request to the address bus transaction VC, such as an AXI4 manager VC. The `Ticks`, `Done`, and `Error` arguments indicate status from the co-simulation software. These must be connected to local variables in the calling process in order for state to be preserved between calls. The calling procedure can choose to use these signals or not, but that inidcate state that is useful in controlling the simulation from state that the software has indicated.

The `Ticks` argument would normally be 0 when generating streams of transactions. The software, however, can ask that transactions are not generated but that simulation time passes for a number of

clock ticks. When this is active the `Ticks` value is non-zero and is decremented at each call to the `CoSimTrans` procedure. The procedure itself manages waiting on a clock tick but the calling process can use this information. It should *not* alter the value however.

The `Done` argument is an indication that the co-simulation software has completed its run, when non-zero, and will no longer generate output. The calling process may use this to halt testing, but the status is an indication that the software is complete, rather than a request to halt. In systems with multiple nodes, a test may wish only to stop when all noes have indicated that they are done. It is possible that the software requested some 'tick' idle cycles when flagging 'done' and the calling process should wait until at least `Ticks` is 0 and `Done` in non-zero before considering a co-simulation node to have completed.

The co-simulation software can also have internal errors and self-check failures. It communicates this by setting the `Error` argument to a non-zero value. The calling process can use this to assert a failure and even stop at the first instance this error state.

The `IntReq` input argument is used to communicate interrupt status to the co-simulation software. This boolean, when true, will call a function in the co-simulation software if one has been registered as an interrupt callback function (see the next section). The callback function is called for every call to `CoSimTrans` that the input is true, so if the source of the interrupt in the test bench is a level driven interrupt, the calling process should only set the `IntReq` true when the state changes and the callback to set/clear interrupt state to the software on alternate calls. An edge driven interrupt would need the `IntReq` true just for cycles on its active edge. For environments that do not use the interrupt callback features, the argument can be left off and the default false value used.

The `NodeNum` argument is the node ID. When only one co-simulation process present, this can be left off and the node defaults to 0. Where multiple processes are generating transactions each must use a unique `NodeNum`.

This, then, is all that is required from the VHDL/OSVVM environment side to be able to generate transactions for both word/sub-word and burst address bus transactions.

6 OSVVM C++ API

The user side C++ code has access to an API in order to generate address bus transactions, allow simulation time to advance and to generate status. It can also get interrupt status from the simulation if present. For each node used in the OSVVM test bench, there is expected to be a user written function named `VUserMain`*n*, where *n* is the number matching the node ID used (e.g. `VUserMain0`). This is like the `main()` of a C/C++ program for the node, or perhaps more like `WinMain()` for Windows non-console programs. An example definition is shown below:

```
-----------------------------------------------------------------
extern "C" void VUserMain0()
-----------------------------------------------------------------
{
// User code here
}
```

The source code for the user code would normally be gathered into a single sub-directory for a given test. This might include just a single source file (e.g. `VUserMain0.cpp`) but can have sub-files to any complexity and would also include the code for all active nodes. All the source code then has access to the co-simulation API.

6.1 OsvvmCosim Class Constructor

The API is defined in the `OsvvmCosim.h` header in `CoSim/code`. This is a C++ wrapper class to abstract away the lower level calls to the co-simulation infrastructure software. A constructor defines which node the object is attached to.

```
-----------------------------------------------------------------
OsvvmCosim (
-----------------------------------------------------------------
  int nodeIn = 0
);
```

Any user file can contruct the API object. This class is a wrapper and does not hold state (except the node number) and so sub-programs can create their own API objects (with the correct node for the top level program) to gain access to the API methods.

6.2 Advancing Time

Advancing time without generating transactions is done with the `tick()` method as defined below.

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

The method is called with a `ticks` parameter to indicate how many calls to the `CoSimTrans` VHDL procedure should be made without generating a transaction. If the calling process does not add any additional clock waits in the loop calling `CoSimTrans`, then the ticks will be clock cycles. The optional `done` and `error` inputs send status to the equivalent arguments on `CoSimTrans` as explained in the previous section.

The transaction methods are the methods that are used to generate the address bus activity.

6.3 Transaction methods

The available transaction methods are shown below

```
------------------------------------------------------------------
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);

------------------------------------------------------------------
void transBurstWrite (
------------------------------------------------------------------
  const uint<nn>_t addr,
        uint8_t    *data,
  const int        bytesize,
  const int        prot = 0);

------------------------------------------------------------------
void  transBurstRead  (
------------------------------------------------------------------
const uint<nn>_t addr,
      uint8_t    *data,
      const int  bytesize,
      const int  prot = 0);
```

These transaction methods are overloaded for varous sizes of address and data , using the `stdint.h` definitions, where `uint<nn>_t` indicates different uint sizes. For address parameters this is `uint32_t` or `uint64_t`. For the data paramters of `transRead` and `transWrite` this is one of `uint8_t`, `uint16_t`, `uint32_t` or `uint64_t` (if address type is also `uint64_t`). Note that using `uint64_t` can only be done if the target test bench is configured for a 64-bit bus architecture.

The single word methods, `transRead` and `transWrite`, take an address, data and optional `prot` (protection) arguments, with the `transRead` having a pointer to the variable for returning the read data. The transWrite returns a value which is a future-proofing for returning read data in the same transaction as a write, but is not yet supported. The `transRead` method does not return a value and a pointer is used to return the data as this is how the method selects the size of the read transaction (byte, half-word etc.) based on the argument's size.

The burst transactions use byte buffers to send and return data. The calling code provides buffer space to a maximum of 4Kbytes preloaded with byte data for writes, or updated with byte data on reads. The

bytesize parameter indicates the transfer size. The co-simulation code will not go beyond 4Kbytes when reading or writing to the buffers, so it is recommended to use 4Kbyte buffers in all cases to avoid overrun.

6.4 Interrupt Callback

User code can register a callback function which will be called by the simulation every time the `IntReq` input to the `CoSimTrans` procedure is true in the OSVVM test process. To register an interrupt callback function the following method is used:

```
------------------------------------------------------------------
void regInterruptCB (
------------------------------------------------------------------
        pVUserInt_t func,
  const int         level=1);
```

The type of callback function, as defined by `pVUserInt_t`, is `int <myfunc>(void)`. Only level 1 (of 255) are currently employed but, for future proofing, an optional `level` argument (defaulting to 1) is provided.

7 Compiling Co-simulation Code

To compile the co-simulation files, both OSVVM and C++, some additional steps are required over a pure OSVVM environment. Some Tcl procedures are available to compile the C++ code which are gathered into `CoSim/Scripts/MakeVproc.tcl`. Before compiling co-simulation code, this script must be included. Ultimately, these scripts call the `makefile` in the `CoSim` directory, setting parameters as appropriate. This build the co-simulation code into a shared object called `VProc.so`, and the user code is compiled to `VUser.so`.

All the provided OSVVM foreign procedures must be analysed into the test library, and this can be done using `AnalyzeForeignProcs`. This has an optional argument to specify the path to the `OsvvmLibraries` directory, either as an absolute path or relative to the current working directory. It defaults to being in the Axi4 or Axi4Lite directories containing the test cases.

```
------------------------------------------------------------------
proc AnalyzeForignProcs {
------------------------------------------------------------------
  {top_level ../../../}
}
```

For normal OSVVM test case code, this might use `RunTest` to analyse and then run the test code. For a co-simulation test additional arguments must be given to compile the C++ code. The `MkVproc` TCL function is used and has three arguments

```
------------------------------------------------------------------
proc MkVproc {
------------------------------------------------------------------
  srcrootdir
  testname
  {libname ""}
}
```

The first argument is a path to where the directory where all the user source code is located, with the second argument being the path to a specific directory, relative to the `srcrootdir`, containing the user code for the particular test. An optional `libname` argument allows for libraries within `CoSim/lib` to be linked with the user code by tagging a `-l:<libname>` argument.

An example `.pro` script to compile a co-simulation test is shown below, where the script is located in `AXI4/Axi4/TestCases`:

```
ChangeWorkingDirectory   $::osvvm::OsvvmCoSimDirectory

MkVproc  $::osvvm::OsvvmCoSimDirectory tests/usercode_size
TestName CoSim_usercode_size
simulate TbAxi4_CoSim.vhd [generic TEST_NAME usercode_size]
```

Figure 4: Compiling Co-simulation Code

In this example, the test case, `TbAxi4_Cosim`, is run with arguments to build the software to run with it. The `MkVproc`call specified that the top level test directory is `CoSim` (using the global `$::osvvm::OsvvmCoSimDirectory`), and that the particular test code to be compiled is in `tests/usercode_size`, under the top level test directory. The output of the make process, instigated by `MkVproc`, will be in the directory from which the `.pro` script was called, which would normally be outside of the `OsvvmLibraries` directory. The co-simulation shared objects, `VProc.so` and `VUser.so`, along with the intermediate compiled files and static libraries are all output to the running directory. With a call to `MkVproc` a clean compile is done. That is so all the build files of any previous compilation are deleted first in order to ensure no clashes or pick up of old files in the event of a compilation error. A `MkVprocNoClean` procedure is provided to skip the cleaning step for diagnostic purposes but would not be used in normal operations.

8 Usage Examples

In this section will be described a set of usage cases for the co-simulation features. Each has a demonstration test case which may be referred as a reference for each of the case.

8.1 C++ test code for word and burst transactions

At its simplest, the co-simulation features allow address bus transaction tests to be written in C++. The user code provides a VUserMain0 function, instantiates an `OsvvmCosim` object and then has access to all the API features described above. The diagram in Figure 2 shows this test setup.

Demo tests for both word/sub-word and burst transactions exist, with the C++ source as:

- `CoSim/tests/usercode_size/VUserMain0.cpp`
- `CoSim/tests/usercode_burst/VUserMain0.cpp`

Both of these programs can be run on the TbAxiMemory.vhd test bench for both Axi4Lite and Axi4. The Axi4Lite and Axi4 test case VHDL for the tests are:

- `AXI4/Axi4Lite/testbench/TbAxi4_CoSim.vhd`
- `AXI4/Axi4/TestCases/TbAxi4_CoSim.vhd`

These VHDL test case files are common to most of the relevant tests as it is just different C++ programs that need to be run on them. A simple example of a program using the word/sub-word API calls is show below.

```
#include "OsvvmCosim.h"

extern "C" void VUserMain0() {
    bool                 error = false;
    uint32_t             wdata = 0;
    OsvvmCosim           cosim(0);

    for (int loop = 0; loop < 4; loop++)
    {
        uint32_t addr  = 0x10000000;
        uint32_t rdata;

        for (int idx = 0; idx < 4; idx++) {
            cosim.transWrite(addr, wdata + idx);
            addr  += 4;
        }

        addr = 0x10000000;

        for (int idx = 0; idx < 4; idx++) {
            cosim.transRead(addr, &rdata);
            if (rdata != (wdata + idx)) {
                error = true;
                break;
            }
            addr += 4;
        }
        wdata += 0x10;
    }

    // Flag to the simulation we're finished, after 10 more iterations
    cosim.tick(10, true, error);

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

Figure 5: Example Co-simulation Test Program

8.2 RISC-V instruction set simulator

The first case above gives full access to the API to write new test code to drive the address bus transactions. A more useful ability might be to run a complex C++ model and have this access logic components in the simulation. The test code in `CoSim/tests/iss` hooks up a RISC-V instruction set simulator. This is a pre-existing open-source model available on github which models a RISC-V processor to the RV32GC+Zicsr standard, with machine level privileges. The model can be called from an external function and provides a means to register a callback function to be called whenever the processor does a load or store access. This callback function can choose to process the access or hand it back to the ISS. The model can also be connected to a gdb debugger as a remote target over TCP/IP and thus an IDE such as Eclipse. The test environment now looks like the following

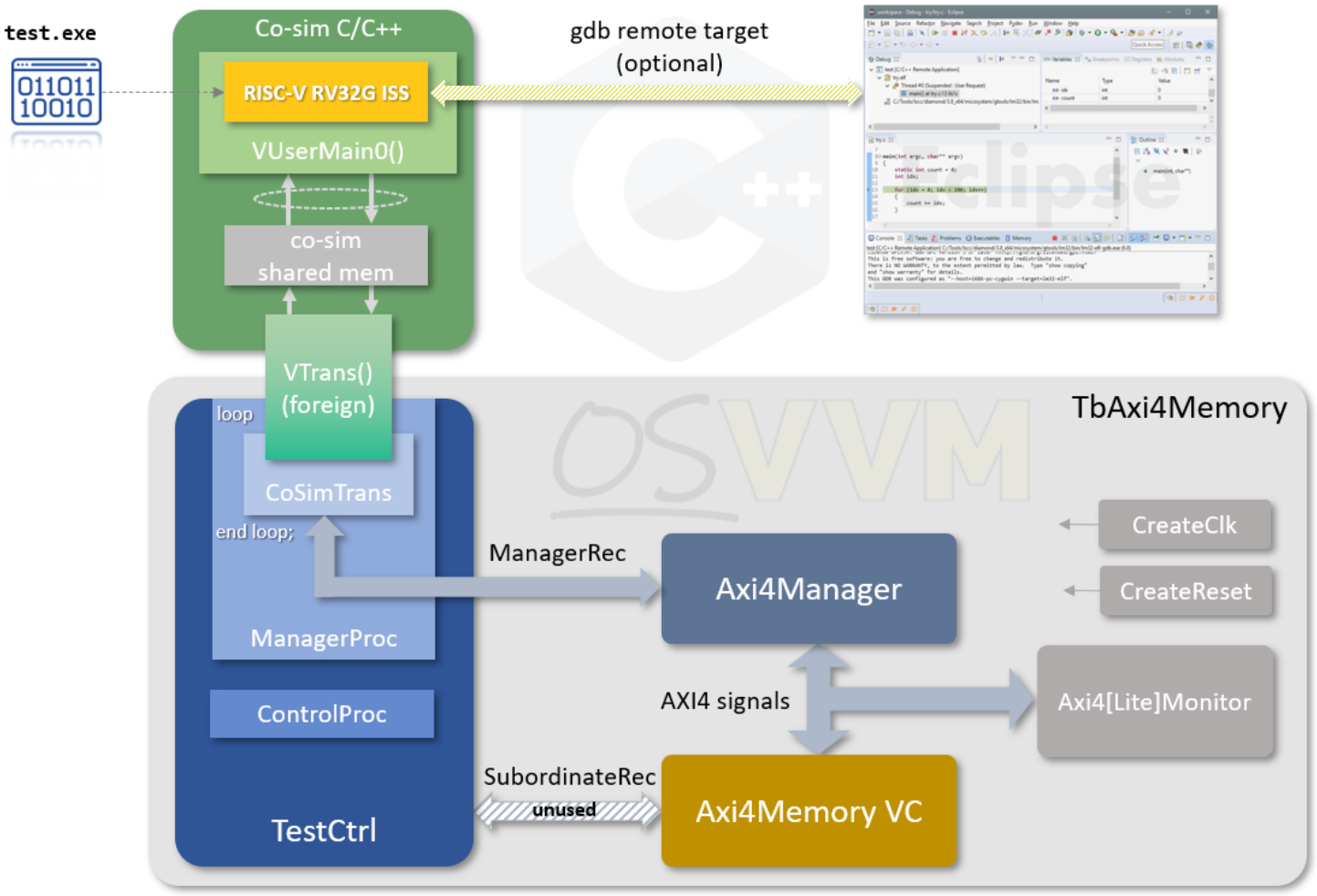


Figure 6: ISS model Co-Simulation Environment

In this test the driving of the transactions now come from within the model itself, with a pre-compiled RISC-V program (`test.exe`, in the test directory) that is one of RISC-V International's unit test—to test

the **sb** (store byte) instruction. The `VUserMain0` function is now just a means to instantiate the model, load a program (if not loaded over gdb) and run it. Binary libraries for the RISC-V ISS model are provided in `CoSim/lib` and the headers in `CoSim/include`.

A simplifed `VUserMain0` program is shown below.

```
#include "OsvvmCosim.h"
#include "rv32.h"
#include "rv32_cpu_gdb.h"

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);

    // 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);

    // Let the simulation free run
    SLEEPFOREVER;
}
```

Figure 7: Main Program with RISC-V ISS

The registered call back function (memcosim in the example above) is called each load and store from the model and has parameters for byte address data and type. A simplified version of the callback is shown below

```
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;
}
```

Figure 8: ISS Callback Function

This example serves to demonstrate the simplicity of connecting an existing C++ model environment to the OSVVM co-simulation features. The test bench uses the `TbAxi4Memory` VC as a target, with the program loaded and then run from that memory, but this just stands in for simulated DUT logic, either a peripheral or a sub-system, say. The callback might be enhanced to do some address decoding to only forward to OSVVM relevant accesses, and hand back those out of range for the rest of the model to handle.

The method described here relies on having a model that is callable from the `VUserMain0` program. The simulators, in general, are executables and are not callable from external programs. The OSVVM co-simulation code provides the means to have a 'free running' program that can call API methods to initiate activity in the logic simulation but, ultimately, this was all initiated from the simulator process. In order to have a program, separate from the simulator, drive the transactions another method is required.

8.3 Connecting to External Programs

If the modelling system to be hooked to OSVVM is, itself, an executable and can't be called from the co-simulation code, then a means is needed to communicate between the external model and the OSVVM co-simulation program. In `CoSim/code` is some source code that defines an `OsvvmCosimSkt` class which acts as a TCP/IP socket server. It defines a protocol for sending word or sub-word read or write

transactions, sending back responses to a connected client. The protocol is based on gdb's remote target protocol, but the class is defined in such a way as to allow a parsing and response methods to be overwritten in a derived class to alter or extend that protocol, so long as the basic structure is a start-of-packet (SOP) byte, data bytes, end-of-packet byte, followed by a fixed number of bytes (which can be 0). The current protocol is restricted to word and sub-word transactions and has no support for time advancement or interrupts, but this is easily added by extending the class to enhance the protocol and add burst transfers, time and interrupt support. It serves as a demonstration and, of course, a user can write their own socket code. The test environment now looks like that shown in the figure below:

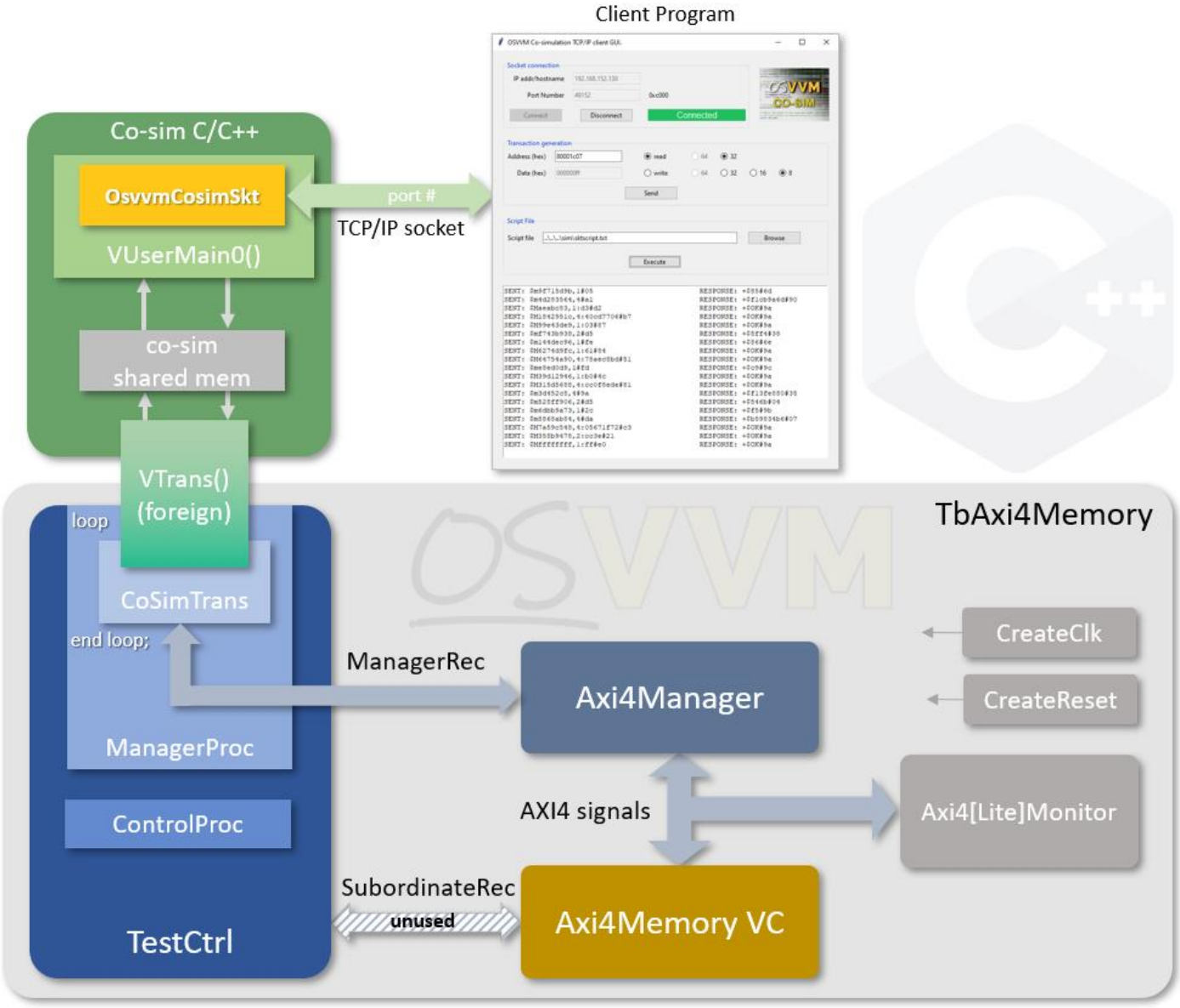


Figure 9: TCP/IP Socket Based Test Environment

The CoSim/tests/socket test code makes use of this class to open up a TCP/IP port and listen for an external connection. The VUserMain0 program is shown below

```
extern "C" void VUserMain0()
{
    OsvvmCosim    cosim(node);
    OsvvmCosimSkt skt(node);
    bool error = false;

    if (skt.ProcessPkts() != OsvvmCosimSkt::OSVVM_COSIM_OK)
    {
        fprintf(stderr, "***ERROR: socket exited with bad status\n");
        error = true;
    }
    else
    {
        printf("DONE\n");
    }

    // Flag to the simulation we're finished, after 10 more iterations
    cosim.tick(10, true, error);

    SLEEPFOREVER;
}
```

Figure 10: Use of OsvvmCosimSkt Class

Standing in for an external program, a python script is provided in CoSim/Scripts. A batch version, used in running the demo tests is called client_batch.py. A GUI version (client_gui.py) can be used to interact directly with the simulation. When the simulation is started the simulation will wait on connection to a port (by default 49152—which is 0xC000). The GUI can then connect to this port and send commands and receive responses. The GUI is shown as the client in Figure 10 above. As well as sending and receiving data manually, the GUI can read a script, and the ISS demo test generates a script as it runs that can be used. There is also a sktscript.txt file in the CoSim/tests/socket directory, used in the demo tests.

8.4 InterruptHandler VC usage

In the 2022.11 release of OSVVM an Interrupt Handler VC component was added to allow for testing of the interrupt generation features of a DUT (see [2] for details). The test environment adds to the basic environment by having an additional InterruptProc process that can generate transactions in the same manner as a MangerProc process. The InterruptHandler arbitrates between the two sources to forward transactions to the Axi4Manager VC based on the state of the DUT's interrupt request line. To use the handler with co-simulation code, the multi-node capabilities are used, where the manager process uses node 0 (the case in the previous examples, and the interrupt process uses node 1. Each node will call its respective top level functions—VUserMain0 and VUserMain1—each running as separate programs. This arrangement is shown in the diagram below.

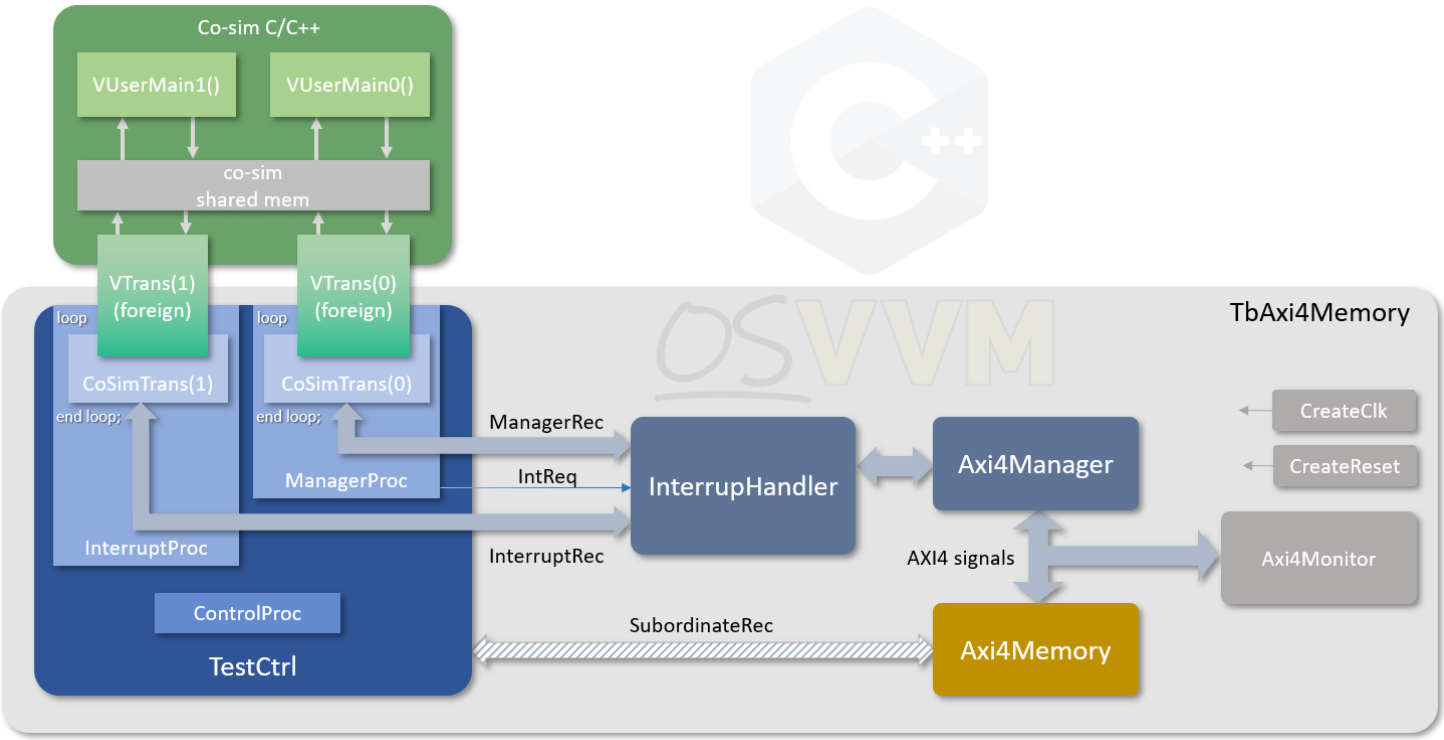


Figure 11: Co-simulation with InterruptHandler VC

A test example is given in the code in `CoSim/tests/interrupt` and runs on the `TbAxi4Memory.vhd` test bench in `Common/TbInterrupt/testbench`. The VHDL test case is `TbAxi4_InterruptCosim1.vhd` under `Common/TbInterrupt/TestCases`. The two `VUserMain` programs running simply emulate the functionality in the `TbAxi4_Interrupt1.vhd`, just using calls to the API to instigate the transactions.

8.5 Interrupt callback

The interrupt callback method of handling interrupts make use of a VHDL global signal, `gIntReq`, defined in the `InterruptHandlerComponentPkg`. The `InterruptHandler` reflects the state of the `IntReq` input and sets the global signal accordingly. The manager process of the test case can then access this signal to set the interrupt input of `CoSimTrans` as required. In the case of the `TbAxi4_InterruptCosim3.vhd` test case, the input to `CoSimTrans` is set to 1 whenever the `gIntReq` changes state. The test code in `CoSim/tests/interruptIss` registers an interrupt callback function with the co-simulation software and this function toggles a local static variable, `IntReq`, each time it is called. The test uses the RISC-V ISS model, and another function is registered as a callback from this model to inspect `IntReq` and return the status to the model. The associated RISC-V test program has the exception vectors set up and enables interrupts before running code to write to a given address to initiate an external interrupt, which the test bench monitors for and sets the `IntReq` output from the `TestCtrl_e` test bench component in the logic simulation. The RISC-V interrupt routine clears the interrupt (writes 0 to the same address that set the interrupt signal) and sets a RISC-V register to say it has been called, which the main test program then checks for, flagging a pass/fail condition.

This setup is shown in the diagram below:

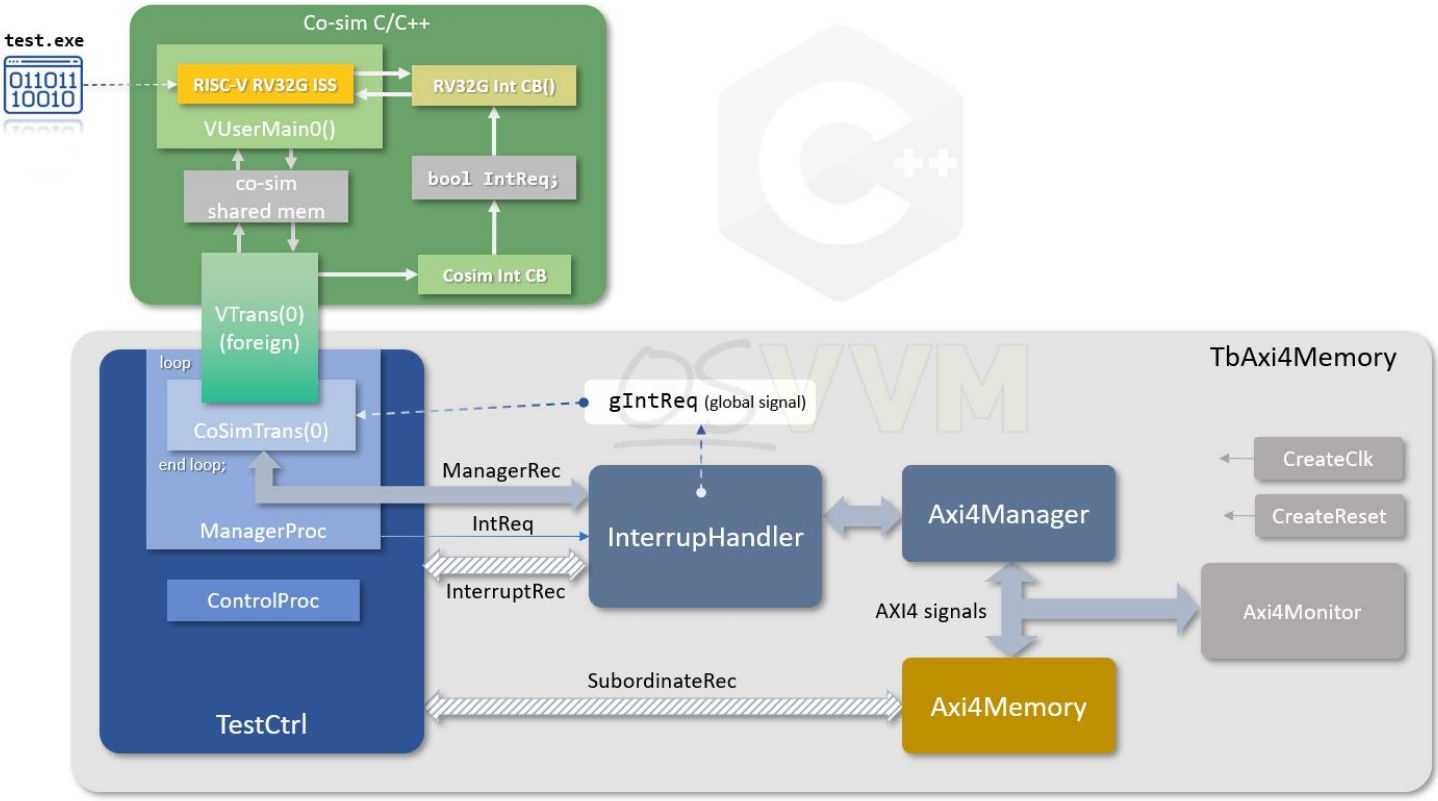


Figure 12: Interrupt Callback Environment

This test, then, demonstrates the connection between an interrupt signal in logic simulation causing an external interrupt initiating a jump to exception code, within a software processor system model.

8.6 Using GHDL callable Simulation

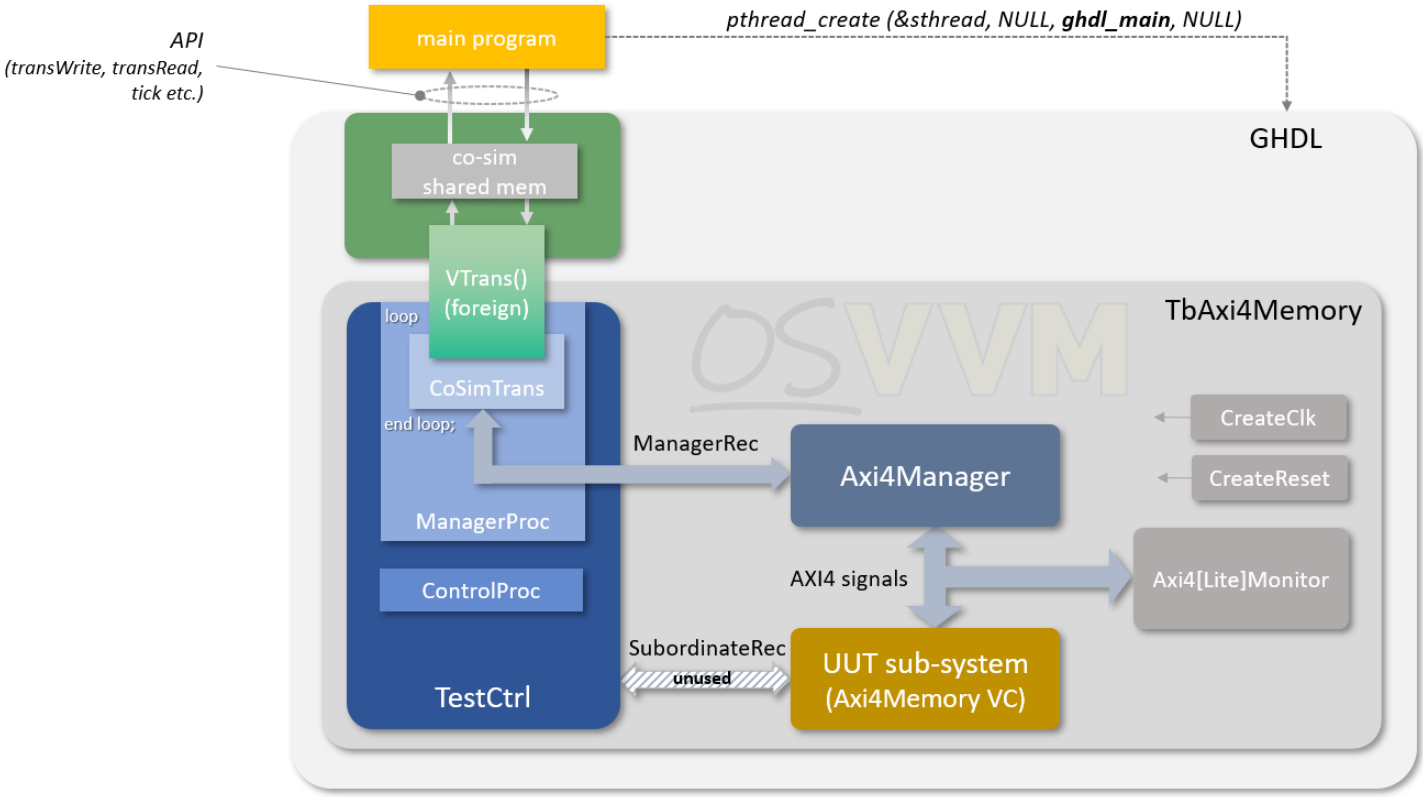
The default behaviour of the co-simulation code is to call one or more user programs, running in threads, with entry points at `VUserMain`*x* (one for each active node). The user test code is called from there and can be any code complexity, modelling whole systems if required. The assumption here is that the user's code, modelling their system, is callable from external code. This is true of the example with the RISC-V ISS, which is instantiated and run from a `VUserMain` program. This may not always be possible for pre-existing modelling systems, where the tool being used is an executable in its own right, and custom models are added to this environment and are called from there.

The supported GHDL simulator has a documented feature whereby the simulator can be called from an external program via the `ghdl_main` function. As documented, this is meant to be called from user code compiled as a shared object and loaded as such as for foreign

language routines. Again, since a user's model may not be in the form of a shared object, a different model is needed to make calling the simulation possible. GHDL's two steps for compiling code is to first analyse all the VHDL files and then, secondly, elaborate the top level. The GHDL tool must be compiled for a `gcc` backend (though LLVM might work, but not tried) and not mcode (see here). The first step results in a set of objects for each component in a respective library. For OSVVM, all these files are in subdirectories of `VHDL_LIBS`. The elaboration step results in an object in the compile directory with the form `e~<name of top level>.o`, for example `e~tbaxi_cosim.o`. It also results in an executable which is the simulation that can be run, but this is not yet suitable. These objects in `VHDL_LIBS` can all be gathered into a single static library, along with the relevant objects from GHDL for the normal VHDL libraries (e.g. `<root to GHDL libs>/std/08/*.o`) to be used in compiling with the main code.

The OSVVM co-simulation code itself must be compiled slightly differently so that it does not attempt to start the threads containing the `VUserMain`*x* code. This is done by compiling with `-DDISABLE_VUSERMAIN_THREAD` in the gcc/g++ flags. This will still create the usual `VProc.so` and `VUser.so` shared object files. It also prevents the user code from setting the `done` argument to `true` in any call to the `tick` method. A call to this method with `done` set to `true` will block forever as it will terminate the simulation. The co-simulation software expects a response for each API call from the simulator that is no longer forthcoming, as the simulation has terminated. When the code is run as a thread from the simulator, this is not an issue, as the simulator terminating is the end of the simulation and the program exits. When called from an external program, it must not hang on this call, and external means are required to call the `tick` method with `done` set to `true` (in code which hasn't been compiled with `-DDISABLE_VUSERMAIN_THREAD`).

The two co-simulation shared objects, along with the new static library of component object files and the elaboration object can now be compiled with the user's main program to replace the original executable with one that now has user code that can call the GHDL simulator and use the co-simulation API directly. However, the simulator must be started first via the call to `ghdl_main`. This is a blocking call and won't exit until the simulation has finished. Therefore this must be run in a separate thread. Once started, and the simulation has initialised the co-simulation code, the user code can then make calls to the co-simulation API, via an `OSvvmCosimClass`. The diagram below summarises this environment



The user code must take care of initialising the simulator, running `ghdl_main` in a thread, and waiting for it to be ready for API accesses. When the simulation finishes, the simulation must be stopped with a call to the `tick` method with the `done` argument set to `true`. As mentioned above, such a call to this method will block forever, and so this needs to be called fro a separate thread. Whan called, the `ghdl_main` function will return and the thread it's running in will terminate. The main program can wait on this thread in order to ensure the simulation has, in fact, finished, using a `pthread_join` call. The abbreviated code fragment below summarises the use of these features.

```
#include <pthread.h>
#include "OsvvmCosim.h"

extern "C" int ghdl_main (int argc, char **argv);
extern "C" int VUserMain0 ();

static int    node = 0;
static int    largc;
static char** largv;

// ------------------------------------------------------
int run_sim (int dummy) {
    int status = ghdl_main(largc, largv);
    return status;
}
// ------------------------------------------------------
int stop_sim_thread (int dummy) {
  OsvvmCosim cosim(node);

  // This call will hang because the simulation will not return
  // a respoinse since it was stopped.
  cosim.tick(1, true, false);
  return 0;
}
// ------------------------------------------------------
void stop_sim(void) {
    pthread_t stop_thread;
    pthread_create(&stop_thread, NULL, (pThreadFunc_t)stop_sim_thread,
                   (void *)((long long)node));
}
// ===================== M A I N ======================
int main (int argc, char **argv) {

    OsvvmCosim cosim(node);
    pthread_t sim_thread;

    // Export arguments for use by thread code
    largc = argc; largv = argv;

    // Create a thread for calling GHDL simulator
    pthread_create(&sim_thread, NULL, (pThreadFunc_t)run_sim, (void *)((long long)node));

    // Allow the simulation's co-simulation code to initialise and be ready
    // for the first API call.
    cosim.waitForSim();

    // Do some tests, calling the VUserMain0 code directly.
    VUserMain0();

    // Post a done status to the simulator.
    stop_sim();

    // Wait for the simulator to exit and the thread to complete.
    pthread_join(sim_thread, NULL);

    return 0;
}
```

The steps necessary to set up the GHDL callable simulation are summarised below.

- Analyse VHDL files: `ghdl -a <list of files>`
- Elaborate top level: `ghdl -e <top level>`
- Compile co-sim code: `make -C <path to OsvvmLibraries/CoSim> [options]`
  - The USRFLAGS must be set to include `-DDISABLE_VUSERMAIN_THREAD`
- Compile user code: `gcc -c main.c -o main.o`
- Gather analysed file objects, user code, and GHDL library objects into a library:
  - ```
    ar crs libtb.a main.o                       \
    `find VHDL_LIBS -name *.o`                  \
    `find <GHDL rootdir>/std/08;  -name *.o`    \
    `find <GHDL rootdir>/ieee/08; -name *.o`
    ```
- Generate a new executable:
  - ```
    gcc ${GHDLFLAGS} e~tbaxi_cosim.o VProc.so VUser.so libtb.a \
                    -o tbaxi4_cosim.exe
    ```
  - The GHDLFLAGS are (on MSYS2/mingw-w64):
    ```
    <GHDL rootdir>/libgrt.a \
    -ldbghelp               \
    -L.                     \
    -L<root>/lib/gcc/x86_64-mingw32/12.2.0/adalib
    ```

The above steps, not including the analysis of the VHDL files are encapsulated in the procedure `MkVprocGhdlMain`, which has the same arguments as `MkVproc` described in section 7. This makes use of a make file, `makefile.ghdl`, in `OsvvmLibraries/CoSim`. This make file is customisable with user overridable variables, as documented in the file's header, so ca be used for user code outside of the `OsvvmLibraries` directory srtucture.

9 Summary

This document defines the features of the co-simulation capabilities of OSVVM. A sub-set of the Address Bus Model Independent Transaction interface is exposed within a C++ environment to allow reads and writes of words or bursts, driving an `AddressBusRecType` signal to initiate transactions on one of the pre-existing Address Bus virtual components. A single point of entry from VHDL is give with the `CoSimTrans` procedure, abstracting away the details of the programming interfaces of the different simulators, and keeping the OSVVM environment intact. Support for interrupts using the `InterruptHandler` VC component is also give for two usage models.

With the domain crossed from VHDL to C++ the possibilities for modelling systems that a partially simulated in C++ and partially simulated in HDL in a logic simulator is demonstrated. Two methods were described (with examples) with a model's code called directly from user co-simulation code, or a TCP/IP socket link is established to drive the simulator from an external program.

10 About the OSVVM

The OSVVM utility and verification component libraries were developed and are maintained by Jim Lewis of SynthWorks VHDL Training. These libraries evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

11 About the Authors and Contributors

11.1 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

11.2 About the Co-simulation Contributor – Simon Southwell

Simon Southwell has thirty plus years of embedded software, ASIC and FPGA design experience in fields that include high performance computing, wireless, cellular modem (LTE) and processor system modelling. Mr. Southwell has developed and successfully deployed co-simulation techniques at a variety of companies using his own open-source IP.

Mr. Southwell is currently developing open source IP in areas such as RISC-V and co-simulation, is actively mentoring undergraduate and new graduate engineers in digital systems design and is also collaborating on the development of OSVVM.

If you find bugs the co-simulation packages or would like to request enhancements, Mr. Southwell can be reached at simon.southwell@gmail.com.

12 References

[1] Address Bus Model Independent Transactions User Guide
[2] Interrupt Handler User Guide