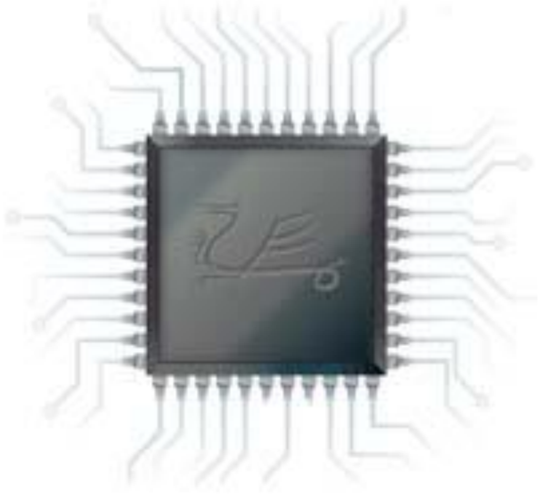
Wireguard FPGA Advanced Co-simulation Verification Environment, a Case Study



Simon Southwell

12th October 2025

Preface

This document is a PDF version of an article written in October 2025 and published on LinkedIn, on the subject of the Wireguard FPGA top level advanced co-simulation test environment.

Simon Southwell
Cambridge, UK
October 2025

Contents

Introduction

In the past, I have written about co-simulation and uploaded various articles on individual aspects of this topic and its advantages. I have even mentioned, in passing, of the use of these techniques for real world projects, though these were commercial projects with source code unavailable or even being used to this day and discussing the details of a company's current internal processes isn't really possible.

More recently, though, I have been involved in a project that is developing an open-source hardware based Wireguard protocol solution that fully embraces the co-simulation techniques that I have been developing and writing about. As such, this project is a perfect vehicle for bringing all these concepts together as a real-world case study that I am free to write about and that the source code is available for inspection and further study. It pretty much brings together all the concepts I have been developing and discussing in my articles, as well as some useful tools from open-source projects that others are developing. Anyone thinking about using these methods can now have a reference project to study as an example of how they might deploy these techniques in their own projects.

In July 2025 I recently presented these ideas at the *FPGA Conference Europe 2025* in Munich and this document is loosely based on what was presented there, though less formally and with some expansion as the time for the presentation was limited.

So, in this document I want to summarise the overall Wireguard FPGA simulation verification environment architecture and the components used, before discussing each of these elements individually in a little more detail. The elements to be described are:

- A "Virtual Processor" co-simulation element
- A sparse memory model, in C, integrated into the simulation
- A RISC-V Instructions Set Simulator, running on the "Virtual Processor"
- Auto-generation of Hardware Abstraction Layer (HAL) software from a SystemRDL description
- An Ethernet UDP/IPv4 C++ model, running on a "Virtual Processor", to drive the Wireguard Project's Ethernet Ports

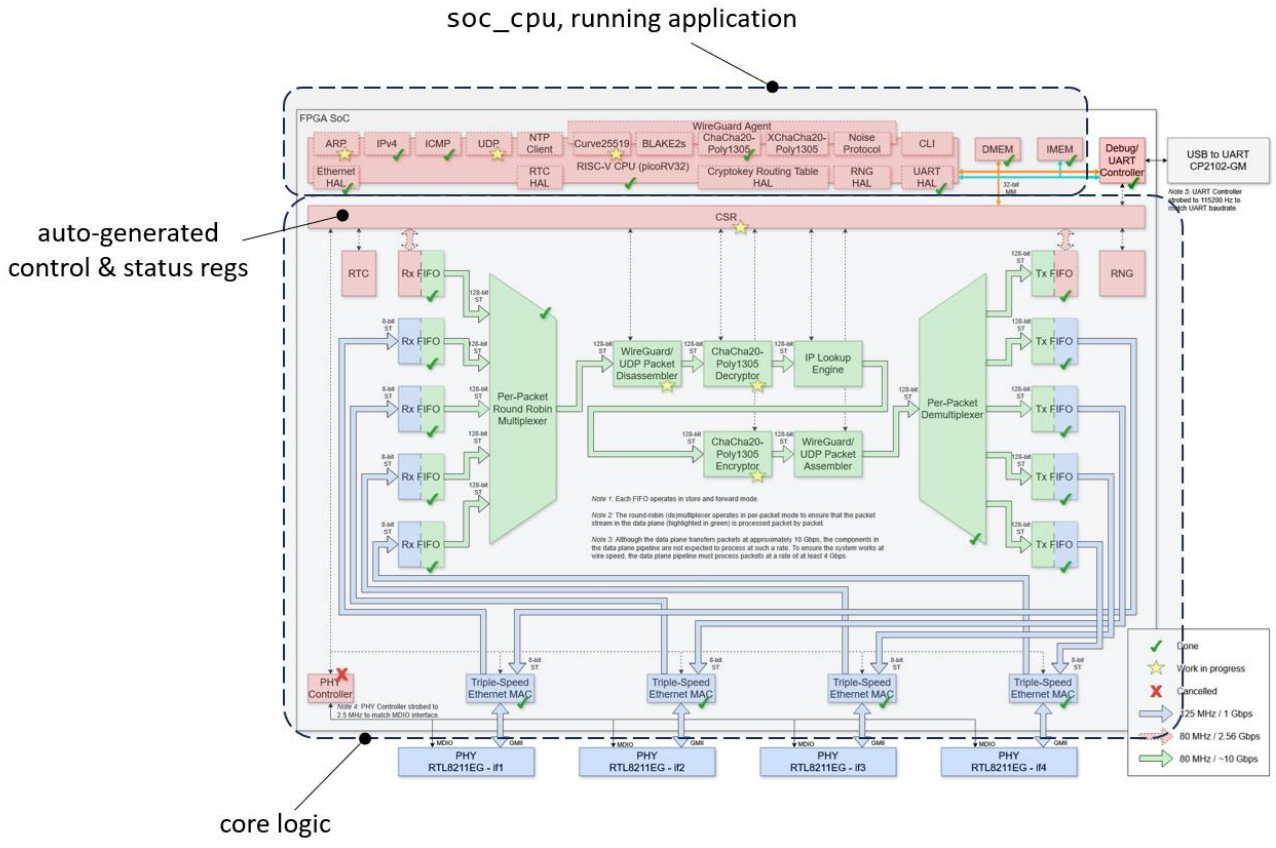
Once each of these is described, they can all be combined to give the full co-simulation and co-development environment used on the Wireguard FPGA project.

Before we get to the co-simulation environment, it is worth a brief moment to summarise the Wireguard FPGA project itself as a framework around which the top-level test environment is built.

The Wireguard FPGA Project in a Nutshell

The *Wireguard FPGA project* is being implemented (at the time of writing) by a group of young engineers, under the Chili.CHIPS*ba banner, out of Bosnia and is supported with funding by the NLnet Foundation. These engineers do it for the pleasure of doing so, to gain new experiences and to learn new skills as well as implement something for the open-source community that is useful for that community. Other projects are in the pipeline, such as developing and open-source PCIe endpoint implementation and a PCIe root complex solution.

The project itself is an open-source SystemVerilog based logic implementation of the Wireguard protocol for creating Virtual Private Networks (VPNs). The solution targets inexpensive hardware platforms, using a commodity Artix7 FPGA and has four 1000Base-T Ethernet ports. The design is done in a self-sufficient way—it does not require the use of a PC host computer to use. The design itself, like the test environment, is supported by other open-source tools such as PipelineC from Julian Kemmerer. The diagram below gives an outline of the project's hardware and software with some things highlighted that will be relevant to the later discussions.



The main takeaway from this diagram, relevant to us, is the fact that that main core has the logic implementation and this has four Ethernet ports with GMII interfaces, supporting UDP/IPv4 packets, which will drive PHY hard macro SERDES. There is also a 32-bit RISC-V processor (marked as soc_cpu on the diagram) that runs the application software. The delivered solution uses the open-source PicoRV32 RISC-V core (from Clifford Wolf et. al.). The software is running 'bare metal' on the processor (i.e. these is no OS), but the main details of its function can be skipped for our purposes, but more details, for those interested, can be found in the documentation.

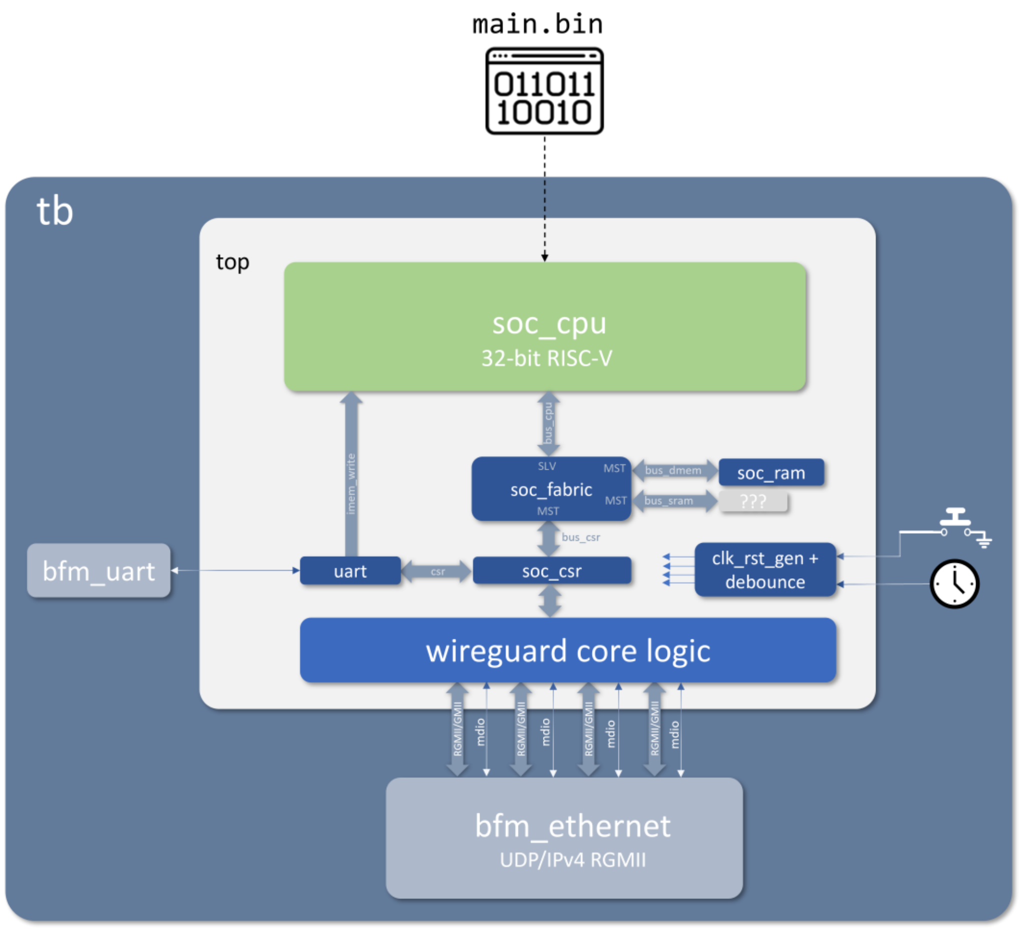
What is more relevant is the interface between the software and the logic hardware. The software has a hardware abstraction layer (HAL) that gives access to the control and status registers (CSRs) and *both* are automatically generated from a common description. This means that additions and changes in the common description get reflected in both the HAL and CSR implementations. We will be looking at this in more detail later. There is also a UART port for loading software and now also has some interesting debugging features.

So, for the co-simulation top level simulation environment we can note that the implementation's external interfaces are four UDP/IPv4 Ethernet ports and a UART port, and we take also note of the RISC-V processor and the HAL/CSR interface.

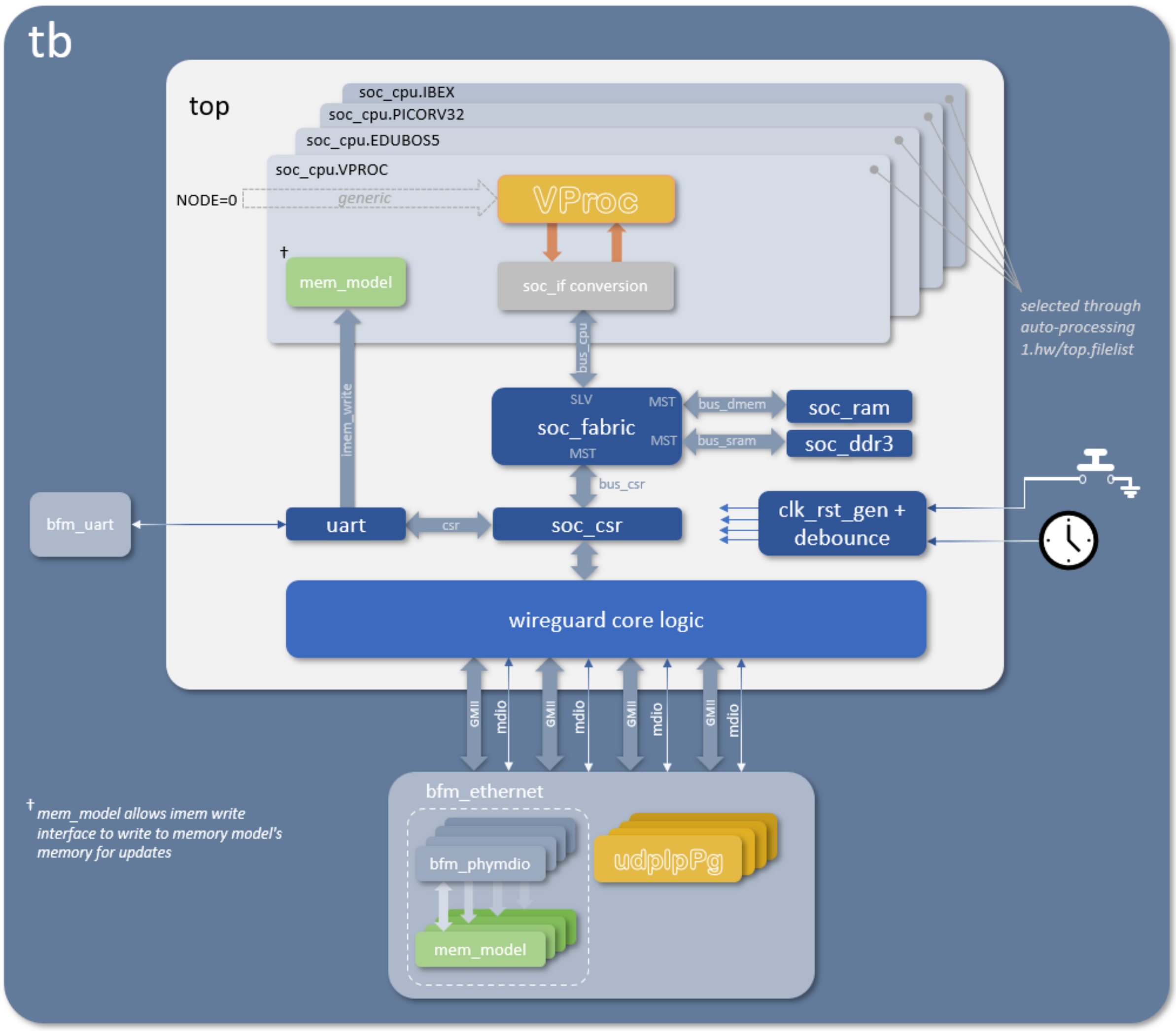
The Basic Top Level Test Bench

In the spirit of open source, the test bench uses the open-source *Verilator* cycle based logic simulator (Wilson Snyder) for its speed advantages, with waveform viewing supported by GTKWave (Anthony Bell) or Surfer (Linköping University).

The top level test bench must be able to drive the external ports and, since we are constructing a co-simulation and co-development environment we need a means of running software—communicating between this code and the CSR registers in the core logic. The basic top level test bench, then, looks like that shown in the diagram below:



From what's just been said, this diagram should hold no surprises—there is a component to drive the Ethernet ports and another for the UART. The processor has a binary image loaded and run on the soc_cpu RISC-V processor, and these are means to generate clocks and reset. In this sense this is very much a 'classical' simulation verification test bench. However, there are some 'secrets' that elevate this to an advanced co-simulation and co-development system. The diagram below is the same as the above but reveals some internal details to show how the 'classic' test bench becomes a co-simulation test bench.



Firstly, the processor component, in the test environment can be selected between a number of versions of the soc_cpu with different RISC-V implementations, such as the *PicoRV32* core, but can also be the *IBEX* core (lowRISC) or the *eduBOS* core from Tarik Ibrahimović. Which core is to be used is selected by the test environment by auto-processing the top level file list to pick the soc_cpu version required and add the SystemVerilog files for that implementation. One of the possibilities is a version (soc_cpu.VPROC) that is based on the *VProc* virtual processor co-simulation component (from wyvernSemi). An associated module is *mem_model* (wyvernSemi) used to model the soc_cpu internal RAM when *VProc* is used. This co-simulation element has, at its heart, a sparse memory model in written C, but is connected to the logic via a Verilog module and the use of the DPI interface. It also presents an API to other software which, as we'll see later, has some advantages.

For the GMII Ethernet port driving the `bfm_ethernet` module has four *udpIpPg* components (wyvernSemi) for driving each of the ports. Not shown (so as not to clutter the diagram), but these *udpIpPg* blocks are also based on *VProc*, but used in a different way from the processor (more later). The `bfm_ethernet` block also has some logic to handle the MDIO signalling associated with the ports. The MDIO interface is just behavioural logic, but the data received and sent from memory via a *mem_model* component. This makes it possible that the source data and the data received is accessible to software.

This has been a whirlwind introduction to the test environment but is the gateway to a true co-simulation and co-development environment. In the next few sections, I want to look at these components in turn, with a little more detail before turning to the software co-simulation and then bring all of the separate components back together and show how all these open-source tools work in tandem for an advanced verification environment.

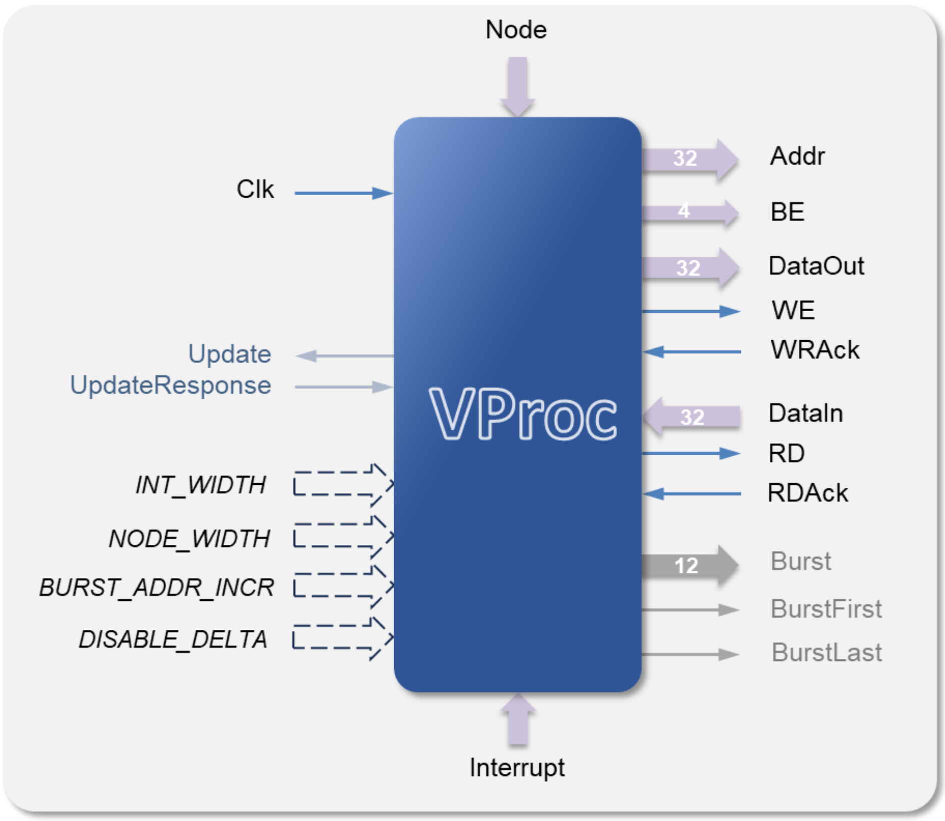
The *VProc* Virtual Processor

I have written about the *VProc* virtual processor before in quite some detail, including how it all works internally. For this document I want to summarise its main features in terms of is usage to get co-simulation features into the top level test bench. So what exactly is it?

VProc is an HDL component (supporting SystemVerilog, Verilog and VHDL) that can be instantiated in a simulation as a processing element, like any other processor core. However, it can run a program that has been natively compiled for the host machine, whether a Windows PC or a Linux workstation, and that program runs on the machine rather than on simulated logic. This makes it execute very fast compared to running a program on a simulated logic processor model. This native program can have access to instigating read and write transactions in the logic simulation over a generic memory mapped bus on the *VProc* HDL component via a supplied API in C or C++. Usually, the bare *VProc* HDL component is wrapped in some bus functional model (BFM) code to turn the generic bus into a specific protocol suitable for the project's requirements. The *VProc repository* provides wrappers for some standard bus protocols such as AHB and AXI. For the Wireguard FPGA project a custom protocol is used, defined as `soc_if`, and a simple wrapper was made for the project to support this with *VProc*. The *VProc* component has some other useful features:

- Has support for burst operations
- Has support for interrupts
- Has support for delta cycle updates
- Supports multiple instantiations

The first two of these features are not needed for the Wireguard Project but allow support for more complex protocols (e.g. AXI) and to make *VProc* a true processing element core. I have written about modelling nested vectored interrupts with *VProc* in more detail in the past if this might be of interest to some. The delta cycle feature is not used for modelling Wireguard FPGA's processor but comes in handy for other *VProc* usage, such as driving arbitrary protocols that are not memory mapped address busses. The VProc HDL component is shown in the diagram below:



For support of multiple instantiations, each instantiated *VProc* HDL component has a unique 'node' number, determined by the port shown at the top of the diagram. The term 'node' is used for historical reasons but think of it as an ID or handle—something that separates it from other instantiations. It is also used to dictate the entry point for the user program to be run on the particular *VProc*. Each instantiated *VProc* is looking for the equivalent of a 'main' entry point. The *VProc* programs are run as part of the simulation (rather than as separate processes—again for speed) and 'main' has already been taken by the simulator itself. So, user program entry points are of the form `VUserMain<n>`, where `<n>` is the node number. For a *VProc* with a node number of 0, the program to be run on that processor, then, has an entry point of `VUserMain0`.

From that point on, any C or C++ program can be written that is valid for the machine and compiler, using any available libraries. The `VUserMain<n>` entry points must have C linkage, and this needs to be accounted for if using C++. For many simulators, running 'foreign' code using the programming interfaces requires the code to be compiled into a shared object (for DLL in Windows) and the user code is compiled into this, along with the *VProc* software (which is lightweight and consists of just two source files). For the Wireguard FPGA project *Verilator* is being used and so the code is compiled into a static library and linked by the simulator itself with its generated C++ code. The result is the same though. When the simulation is run the user code starts executing for each *VProc* instantiation. To be useful, though, it needs to interact with the simulation to do reads and writes over the memory mapped bus.

The *VProc* API

To do reads and writes in the logic simulation an API is provided in both C and C++. This is not a complex API and the relevant function and methods used for the Wireguard FPGA project are shown below:

Low Level C API

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

C++ *VProc* class API

```
• vp0->writeByte  (addr,  data, delta=0);
• vp0->writeHword (addr,  data, delta=0);
• vp0->writeWord  (addr,  data, delta=0);
• vp0->readByte   (addr, *data, delta=0);
• vp0->readHword  (addr, *data, delta=0);
• vp0->readWord   (addr, *data, delta=0);
• vp0->tick       (ticks);
```

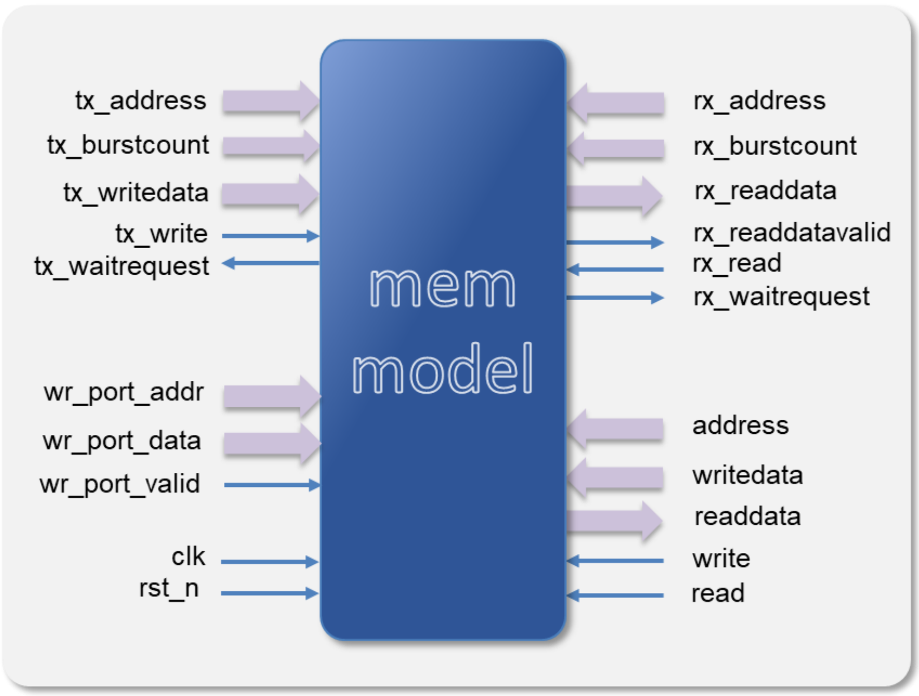
So here we have basic means to do reads and writes. The C API is more primitive with a couple of write functions and a read function, whereas the C++ API allows accesses of different sizes. Both have a 'tick' function or method. This allows the program to advance time in the simulation for a given number of cycles without doing a transaction. The user program, running on the host machine, can now interact with the logic simulation to do memory mapped bus reads and writes, just like a logic core. A simple test program is shown below using the C++ API:

```
#include "VProcClass.h"

static const int node     = 0;

extern "C" void VUserMain0(void)   // VProc "main" entry for node 0
{
    // API constructor
    VProc* vp0 = new VProc(node);  // VProc API object for node 0

    vp0->tick(100); // Wait a bit

    uint32_t addr  = 0x10001000; // Location in DMEM
    uint32_t wdata = 0x900dc0de;

    vp0->writeWord(addr, wdata);
    VPrint("Written   0x%08x  to  addr 0x%08x\n", wdata, addr);

    vp0->tick(10);  // Emulate some processing (10 clock cycles)

    uint32_t rdata;
    vp0->readWord(addr, &rdata);

    if (rdata == wdata)
        VPrint("Read back 0x%08x from addr 0x%08x\n", rdata, addr);
    else
        VPrint("***ERROR: data mismatch at addr = 0x%08x\n", addr);

    // Sleep forever (and allow simulation to continue)
    while(true)
        vp0->tick(GO_TO_SLEEP);
}
```

With this API it is now a simple matter of writing test code to drive memory or CSR registers that are memory mapped on the bus. This is very useful but not all that can be done with this, as we shall see later.

A Co-simulation Sparse Memory Model

Closely associated with *VProc* is a co-simulation memory model called *mem_model*. At is heart is a C model that models memory up to 2^{64} bytes, but without allocating all that space. It, in fact, allocates no memory at initialisation and allocates pages as memory locations are accessed. Since it is unlikely that all the space will be accessed it only uses the small amount of memory that's needed, but this can be anywhere in the full 64-bit address space. The C model is connected to a logic simulation via an HDL component, which is shown below:



This component has various useful interfaces with, at the top, read and write burst ports (modelled on the Avalon protocol), a simple write port (for loading programs to memory) and a memory mapped address bus. As for *VProc*, the component is usually wrapped in some behavioural code to expose just an interface that the actual memory would show to the rest of the logic.

Multiple memory model components can be instantiated in a simulation. However, these all access the *same* address space. In other words, each HDL component is a 'port' into the same memory, just as you would find available on an FPGA from the logic.

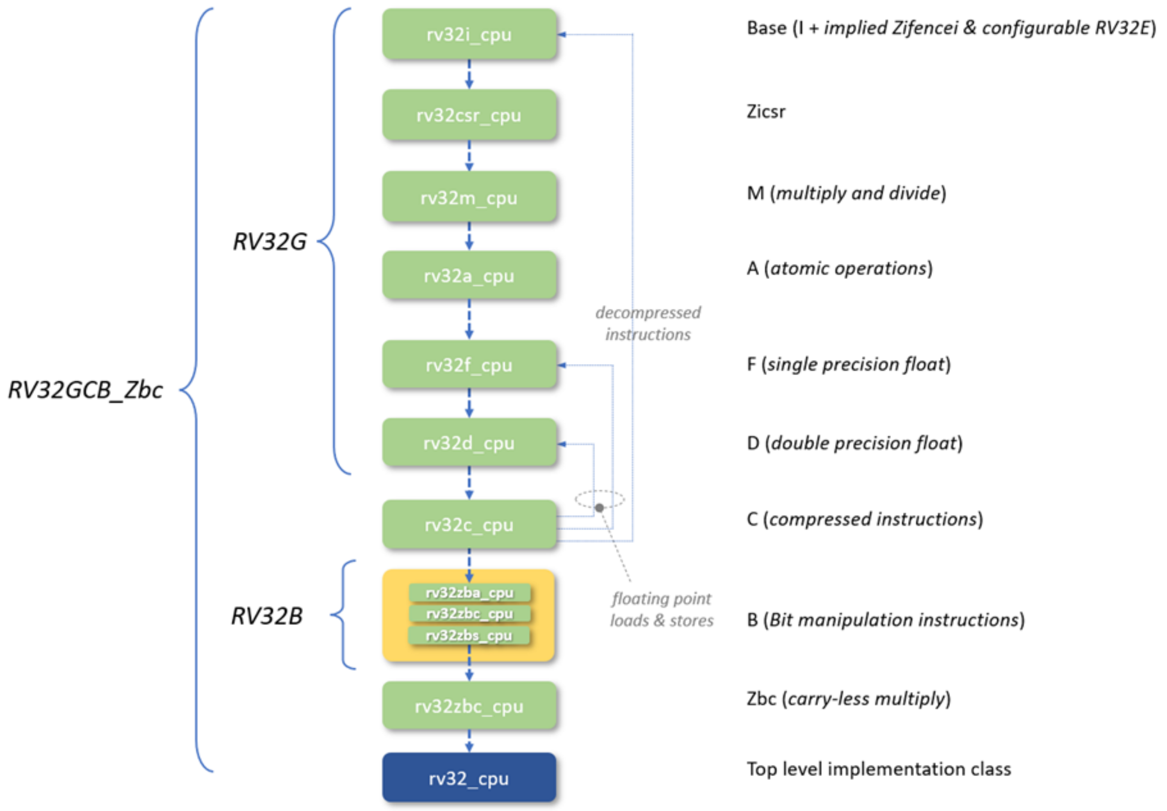
The memory model also has a C API. Although it is used mainly by the logic programming interface code (for DPI in Wireguard FPGA's case) to connect the C model to the HDL, it can also be made available to user code running on a *VProc* component. This is very handy for test purposes. For example, suppose it is useful to load some large data set from a file into memory. Since the memory component will be on the bus the *VProc* program could simply do writes over the bus to load the data. This, though, takes simulation time to do and may be a significant number of clock cycles. Using the API directly, the data can be loaded into memory without running any simulation cycles and will be orders of magnitude faster. Similar advantages can be seen if wishing to dump test or results data that is sitting in memory. A summary of the API, relevant to the Wireguard FPGA project and accessed via a `mem.h` header, is given below:

```
void WriteRamByte     (uint64_t addr, uint32_t data, int memnode);
void WriteRamHword    (uint64_t addr, uint32_t data, int le, int memnode);
void WriteRamWord     (uint64_t addr, uint32_t data, int le, int memnode);
uint32_t ReadRamByte  (uint64_t addr,         int memnode);
uint32_t ReadRamHword (uint64_t addr, int le, int memnode);
uint32_t ReadRamWord  (uint64_t addr, int le, int memnode);
```

A couple of things to note is that the endianness of the word accesses can be selected with the `le` argument, and the `memnode` argument must be 0. Internally, multiple address spaces can be supported, but to access the same space as the HDL, then a `memnode` of 0 must be used.

The rv32 RISC-V Instructions Set Simulator

It has been mentioned that a *VProc* running user program can be anything valid for a C or C++ program. This could be an “Instruction Set Simulator” (ISS) of a processor. That is, a software model that is capable of executing instructions for a given processor’s instruction set architecture. Relevant to the Wireguard FPGA project is the *rv32* RISC-V ISS (wyvernSemi). I have written in more detail about this model and its use of C++ inheritance to model the standard extensions and make highly configurable. The diagram below summarises this:


RV32GCB_Zbc
RV32G
RV32B
rv32i_cpu
rv32csr_cpu
rv32m_cpu
rv32a_cpu
rv32f_cpu
rv32d_cpu
rv32c_cpu
rv32zba_cpu
rv32zbc_cpu
rv32zbs_cpu
rv32zbc_cpu
rv32_cpu
decompressed instructions
floating point loads & stores
Base (I + implied Zifencei & configurable RV32E)
Zicsr
M (multiply and divide)
A (atomic operations)
F (single precision float)
D (double precision float)
C (compressed instructions)
B (Bit manipulation instructions)
Zbc (carry-less multiply)
Top level implementation class

The ISS is a 32-bit model with up to the RV32GCBE_Zicsr_Zbc standard but is configurable to be any valid subset of these extensions and a GUI is provided to configure the model's source code to the desired standard.

Although the model can be compiled as a stand-alone executable with some supporting peripheral models that can even allow FreeRTOS to be run, it can also be compiled to be a library for linking with other code such as, for example, code that is to be run on a *VProc* component.

The model has various callback features for connection with external programs, but the one relevant to the Wireguard FPGA project is for memory accesses. An external program can register a function with the ISS that will be called whenever the model does a memory mapped access, such as executing a `load` or a `store` instruction. The address and data (for writes) is passed in and the user supplied callback function can choose to handle the access, passing back any read data, or flag that it hasn't handled it and for the model to process it instead.

Another useful feature of the ISS is that it has a GDB remote debug interface, allowing debugging of code running on the model to be done just as for a hardware solution. It also has an internal timing model with default cycle counts for each instruction type, but this can be configured to match a particular core, such as the *PicoRV32* core used for the Wireguard FPGA project. In addition, the memory access callback function can return a number of wait states to the model to more accurately account for external timing.

For the Wireguard FPGA project, there will need to be some small amount of code to integrate the ISS using the features I've just mentioned, but this will be discussed in the "Putting it all Together" section.

Hardware Abstraction Layer Auto-generation

In most of the companies I've worked for the software that interacts with the logic hardware registers has done so through a "hardware abstraction layer" or HAL. This takes the set of all control and status registers (CSR) and, possibly, internal memory, and presents methods for accessing whole registers, bit fields within a register, or memory locations in a RAM, in a consistent way. More recently, the CSR registers and memory have been described in a common description language, such as JSON, and the HAL and CSR registers auto-generated from this to avoid mismatching of software and hardware caused by manual updating of two separate implementations. This was the case in my last two employments and is now a widespread practice. There is a standard description language especially for doing such a task called SystemRDL which is an Accellera Standard and is what is used on the Wireguard FPGA project.

The standard dictates how to describe the registers, but a means to turn this into logic and HAL software is required. Fortunately, this has already been done with the open-source *peakrdl* tools (Alex Mykyta et. al.). The Wireguard FPGA project uses the `peakrdl-regblock` tool to generates the logic registers and the `peakrdl-cheader` tool to generate the HAL software for the target RISC-V processor application.

It would also be advantageous if, in addition to a HAL for the RISC-V processor, we could have one suitable for *VProc* user code, that presents the *same* API as for the RISC-V application. This might be a means to execute the application compiled natively for the host machine and run on a *VProc* virtual processor. The HAL generated for the RISC-V processor makes memory accesses as normal, but a HAL generated for native compilation makes calls to *VProc* API methods. This would allow application code to be run natively and without the need for cross compilation for the target platform and will run faster than for the target processor. The timings of the simulation will not be as accurate, but for initial functional testing this becomes a viable method.

In some of my past commercial settings employing these techniques, the HAL has been used in a read-modify-write model, where the slight inefficiency is not a limiting factor with processor speeds in the high hundreds of MHz or faster, and CSR accessing not frequent. HALs in these cases were constructed with hierarchical dereferencing, ending in a pointer to access a whole register, and subfields on writes updated to a pre-read value and written back. Thus, for the HAL targeting the *VProc* API overloading methods can be employed to call a method or function whenever a pointer of a particular type is read or written. This means the same HAL code could be used for target and *VProc* code, with the register type redefined to overload on accesses to call *VProc* API methods.

The *peakrdl* output for the HAL uses a different method, which is valid and more efficient. It generates a set of structures that overlay the memory hierarchy and, ultimately, the registers. Sub-fields within a register are accessed via bitfields in a structure overlaying the whole register. This makes more intuitive sense and requires no additional code as the structure hierarchy ends in a reference to the fields which may be written or read directly. An example *peakrdl* code fragment is shown below:

```
#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>
#include <assert.h>

// Reg - csr::ip_lookup_engine::table::ip_address
#define CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__ADDRESS_bm 0xffffffff
#define CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__ADDRESS_bp 0
#define CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__ADDRESS_bw 32
#define CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__MASK_bm 0xffffffff00000000
#define CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__MASK_bp 32
#define CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__MASK_bw 32
typedef union {
    struct __attribute__ ((__packed__)) {
        uint64_t address :32;
        uint64_t mask :32;
    } f;
    uint64_t w;
} csr__ip_lookup_engine__table__ip_address_t;

// Reg - csr::ip_lookup_engine::table::public_key
#define CSR__IP_LOOKUP_ENGINE__TABLE__PUBLIC_KEY__KEY_bm 0xffffffff
#define CSR__IP_LOOKUP_ENGINE__TABLE__PUBLIC_KEY__KEY_bp 0
#define CSR__IP_LOOKUP_ENGINE__TABLE__PUBLIC_KEY__KEY_bw 32
typedef union {
    struct __attribute__ ((__packed__)) {
        uint32_t key :32;
    } f;
    uint32_t w;
} csr__ip_lookup_engine__table__public_key_t;
```

The downside for *VProc* is that the overloading techniques employed before can't be used with this as bitfields accesses can't be overloaded. The good news is that the people behind *peakrdl* provide another tool, *systemrdl-compiler*, to allow custom output from the SystemRDL file.

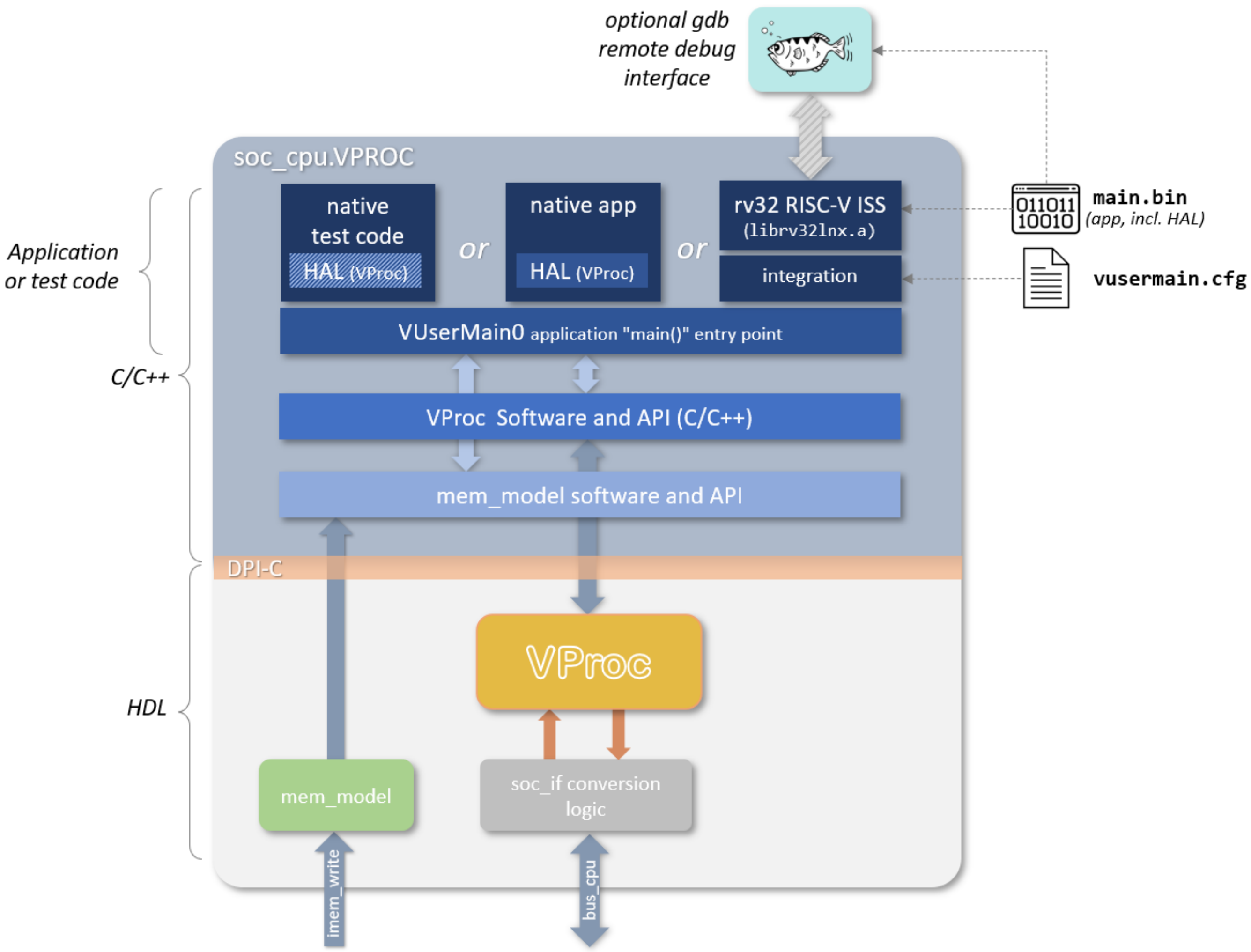
In the Wireguard FPGA project, two HALs are generated, using *systemrdl-compiler*, for the RISC-V target and the *VProc* target. The same API is presented to both target, with either register and sub-field accesses via a read or write method. For the target, this just accesses the *peakrdl* generated structure but for *VProc* it makes a call to the API. The differences between the two are abstracted away by the leaf read or write method. Below are shown code fragments for the two outputs, the first being for the target and second for *VProc* using its C API.

```
class csr__ip_lookup_engine__table__ip_address_vp_t {
public:
    csr__ip_lookup_engine__table__ip_address_vp_t (uint32_t* reg_addr = 0) :
              reg((csr__ip_lookup_engine__table__ip_address_t*)reg_addr) {};

    inline void     full(const uint64_t data) {reg->w = data;};
    inline uint64_t full()                    {return reg->w;};

    inline void     address(const uint64_t data) {reg->f.address = data;};
    inline uint64_t address()                    {return reg->f.address;};
    inline void     mask(const uint64_t data)    {reg->f.mask = data;};
    inline uint64_t mask()                       {return reg->f.mask;};

private:
    csr__ip_lookup_engine__table__ip_address_t* reg;
};
```

```
class csr__ip_lookup_engine__table__ip_address_vp_t {
public:
    csr__ip_lookup_engine__table__ip_address_vp_t (uint32_t* reg_addr = 0) :
              reg((uint64_t)reg_addr) {};

 inline void  address (const uint64_t data) {
        uint32_t wdata = (uint32_t)(data & 0xffffffff);
        VWriteBE(reg + 0, wdata << 0, 0xf, NO_DELTA_UPDATE, SOC_CPU_VPNODE);
        VTick(rand() % 33, SOC_CPU_VPNODE); };

 inline uint64_t address () {
        uint32_t rdata;
        VRead(reg + 0, &rdata, NO_DELTA_UPDATE, SOC_CPU_VPNODE);
        VTick(rand() % 33, SOC_CPU_VPNODE);
        return (((uint64_t)rdata << 0) &
               CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__ADDRESS_bm) >>
               CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__ADDRESS_bp; };

 inline void mask (const uint64_t data) {
        uint32_t wdata = (uint32_t)(data & 0xffffffff);
        VWriteBE(reg + 4, wdata << 0, 0xf, NO_DELTA_UPDATE, SOC_CPU_VPNODE);
        VTick(rand() % 33, SOC_CPU_VPNODE); };

 inline uint64_t mask () {
        uint32_t rdata;
        VRead(reg + 4, &rdata, NO_DELTA_UPDATE, SOC_CPU_VPNODE);
        VTick(rand() % 33, SOC_CPU_VPNODE);
        return (((uint64_t)rdata << 32) &
               CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__MASK_bm) >>
               CSR__IP_LOOKUP_ENGINE__TABLE__IP_ADDRESS__MASK_bp; };
private:
    uint32_t reg;
}
```

Putting it all Together for the Processor

We now have all the components we need to construct a co-simulation processor environment, and for constructing this into an soc_cpu. The diagram below shows more detail of the soc_cpu.VPROC component.


optional gdb remote debug interface
soc_cpu.VPROC
native test code
HAL (VProc)
or
native app
HAL (VProc)
or
rv32 RISC-V ISS (librv32lnx.a)
integration
main.bin (app, incl. HAL)
vusermain.cfg
Application or test code
C/C++
VUserMain0 application "main()" entry point
VProc Software and API (C/C++)
mem_model software and API
DPI-C
VProc
HDL
mem_model
soc_if conversion logic
imem_write
bus_cpu

For the HDL we have what has been shown before with a *VProc* instantiated and some conversion logic to an `soc_if` memory mapped address bus, along with a *mem_model* for local memory. For the software both the memory model and the *VProc* software present APIs to the running user program. The user code can now be one of three basic usage types:

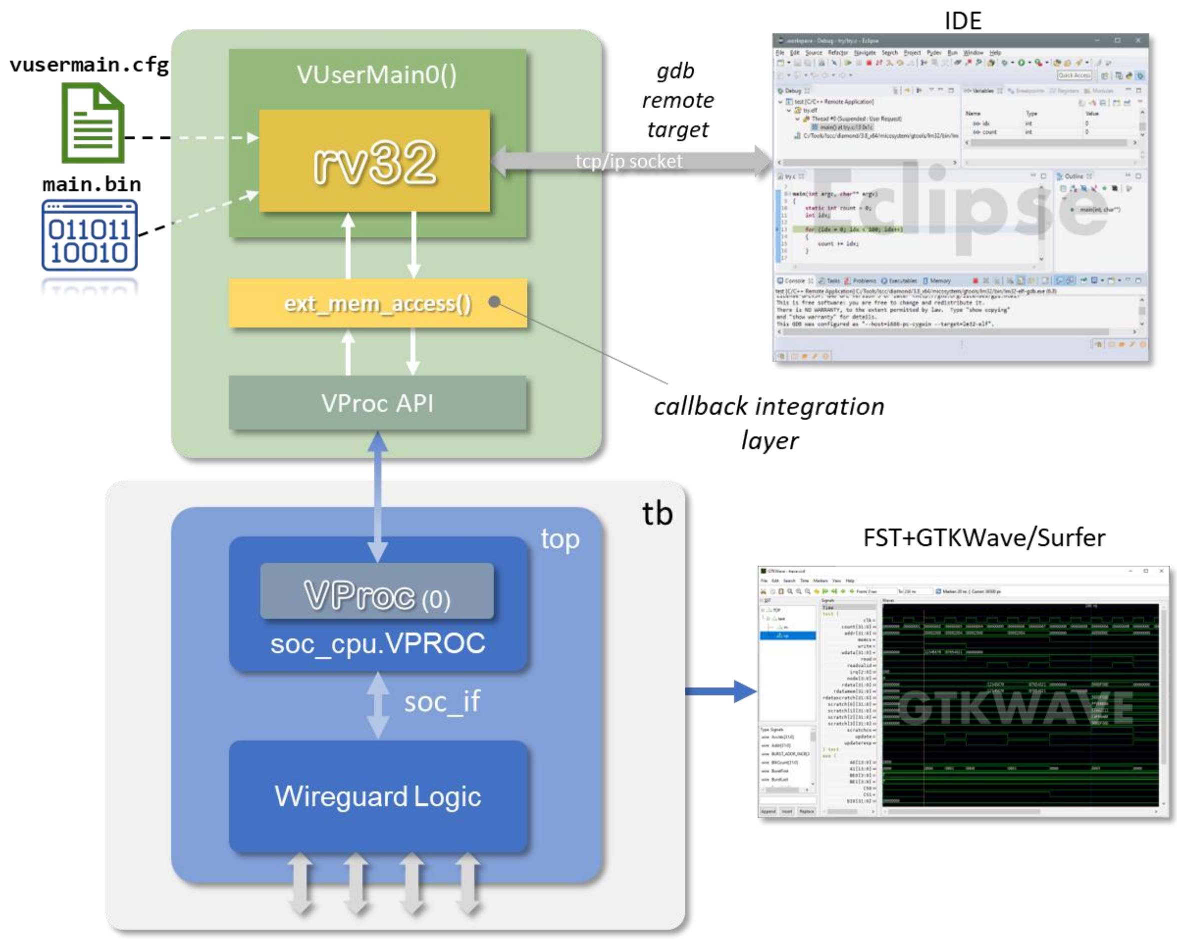
- Native test code to drive `soc_cpu` bus using *VProc* API directly, as discussed previously (or, optionally, also via the HAL)
- Application software compiled natively with modified *VProc* HAL (presented with the same API as application code compiled for a RISC-V target)
- *rv32* RISC-V ISS and integration code to run application compiled for the target platform's processor.

Debugging any of these three alternatives is a requirement. For the first two native methods this is easy as *Verilator* generates an executable and this includes the user code, whether test code or a natively compiled application. So normal gdb debugging for a host program (via an IDE, such as Eclipse, if available) can be employed. For the ISS the supplied remote debugging interface can be used with a RISC-V toolchain gdb.

Which method used depends on the task required. For the first methods this is deployed for basic bring up and component integration testing, targeting registers to instigate directed activity and verification. The second method is used for initial application functional verification where accurate system timing is not a consideration at that point in development, but general system functionality is, and for speed of simulation to run as much software as possible in a practical amount of time. The ISS method gives full co-simulation of the application with the logic hardware with much more accurate timings but still running faster than if run on a RTL model of the processor though, as mentioned before, this is selectable as an option.

Integrating rv32 ISS

In this section we will have a brief look at the features of the *rv32* integration code used to connect the ISS to the test bench. The diagram below summarises the connections:



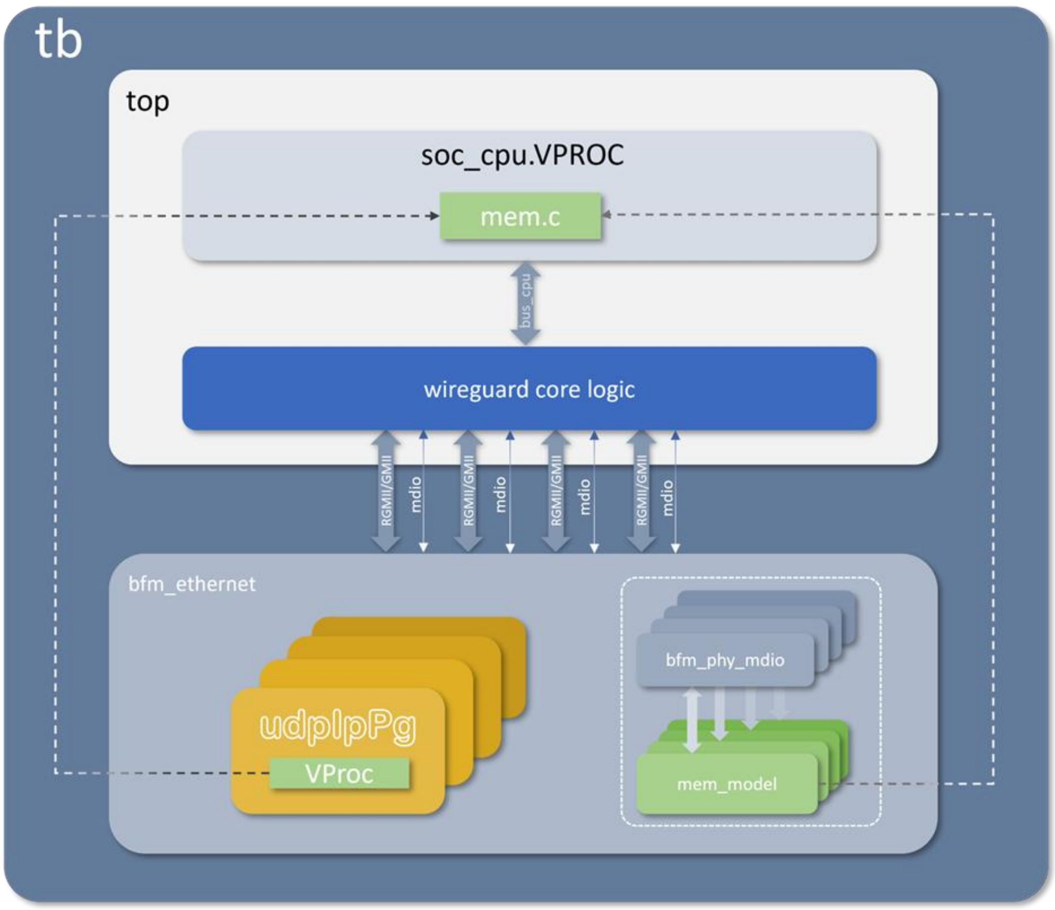
As mentioned above, the ISS has a memory access callback feature where a user provided function is called for every read and write instruction that the model executes. The integration layer resides in a function (`ext_mem_access()` in the diagram) that is registered as the memory callback function. All memory accesses are offered to the callback and the code inspects the address to decide if it sits in a configurable address range for processing. If it is, then it makes the appropriate read or write call to the *VProc* API method to instigate a simulation transaction. Otherwise, it accesses the memory model via the direct access API. In addition, the callback is passed in the ISS's reckoning of clock cycles, based on the timing model of the instruction types it has executed. It compares this with the simulation clock cycle count that is made available and if there is a discrepancy, it will call the tick method with an appropriate cycle count to realign. This means that, although the non-memory access instructions in the model take zero simulation time (as time only advances in the simulation when a *VProc* API call is made), this is accounted for with the call to the tick method, keeping accurate CPU processing time between the bus transactions.

Ethernet UDP/IPv4 Driver

To finish off the description of the Wireguard FPGA project's co-simulation environment the *udpIpPg* features will be discussed that's used in the `bfm_ethernet` module to drive the GMII Ethernet ports.

The component is also based around the *VProc* virtual processor, though used in a slightly different way and is derived from a similar project, *tcpIpPg*. Some C++ code runs on the virtual processor to encode and decode UDP packets wrapped in IPv4 frames and Gigabit Ethernet encoded for suitability for a GMII interface. The *VProc* address bus can now be used to memory map the GMII transmit and receive ports into its address space. Some additional code accesses these addresses to update the TX port and read the RX port, every cycle. If no packet is available to send it transmits idles, else it starts to send the bytes of the transmit packet out over the TX port. All the while, the RX port is inspected for packets arriving and these are buffered as they do so. In fact, this method can use *VProc* to map any arbitrary set of ports and thus any protocol can be modelled in this way. I have written about this in more detail in another article, with the *pcievhost* project being another example for the PCIe protocol.

The *udpIpPg* model software provides a user API that allows packets to be constructed around some payload data and then transmitted. A callback user function can be registered to be called when a whole packet arrives and can save this off for processing in the user code. The user code can then send test packets and receive packets from the logic. In the Wireguard FPGA project, the situation is shown in the diagram below.



The main thing to note here is the use of the memory model. The user code running on the *udpIpPg's VProc* gets its packet data from the *mem_model* memory space, and places received data here as well. This is also true of the `bfm_phy_mdio` modules. This makes data for transmission and reception available from the program running on the CPU *VProc*, and received data can be inspected from there as well. So a test running on the processor has end-to-end access of the data for cross-referencing and self-checking making verification that much easier without consuming any simulation cycles.

Conclusions

In this document a case study for a co-simulation logic simulation was presented in the form of the Wireguard FPGA project's top level simulation test bench. We first looked at a 'classic' looking test bench architecture before revealing some open-source tools that elevated this to a co-simulation and co-development environment. The *VProc* virtual processor provided a bridge between the HDL logic simulation and a user program, compiled natively for the host machine, with access to all the C or C++ infrastructure that is available, with an HDL component instantiated in the simulation and an API for the user program to interact with the simulation over a memory mapped address bus.

Associated with the virtual processor was a sparse C memory model connected to the simulation as another co-simulation element. This allowed a 2^{64} memory space to be accessed either via the HDL instantiated modules or directly from a *VProc* running program. These co-simulation elements were then constructed into an 'soc_cpu' in place of the *PicoRV32* RISC-V softcore RTL. At this point, test programs could be written to drive the bus in the simulation for directed testing of the design.

To be able to run application code in the simulation, the *rv32* ISS was discussed which could run RISC-V compiled application code, with a configurable timing model, a gdb remote debug interface and the means, via callbacks, to hook together with other code. By running this ISS on the virtual processor, and via a small integration layer, this was connected to the bus interface in the simulation to drive transactions to the logic's registers and memory.

As is now common practice in the industry a HAL is auto-generated, along with the CSR logic from a common source using a description language. For the Wireguard FPGA project the register description used the SystemRDL standard and utilised some more open-source tools via the *peakrdl* project to generate the HAL software and the control and status register logic. This opened up the possibility of compiling the application software 'natively' with a common HAL for both the target platform and the co-simulation logic simulation. To give a common API for both compiled versions of the application the use of the *systemrdl-compiler*, provided along with the *peakrdl* toolset, generates two version of a wrapper HAL, where one uses the *peakrdl* output, and the other calls the *VProc* API. Now the application code can be compiled to run in the simulation unmodified, for functional testing, debug and development, with speed advantages though with less accurate timings.

Finally, we looked at the *udpIpPg* model for driving the UDP/IPv4 GMII Gigabit Ethernet ports. This was also built around a *VProc* element where the packets were encoded and decoded in the user program via an API which also allowed for transmission and reception of packets over the HDL components TX and RX GMII interfaces. An advantage of this method is that the user program has direct access to the memory model's address space for streaming in and out the packet data, allowing the soc_cpu's *VProc* program to generate packet data for transmission and verify received packets directly.

With all these open-source tools and a set of methodologies to bind them together, a true advanced co-simulation and co-development environment is produced. None of these tools we've looked at dictate a particular method. For example, the technology behind *VProc* has been integrated into the OSVVM VHDL methodology and libraries to give it co-simulation capabilities which could be used in the same manner as for the Wireguard FPGA project. So some or all of these methods can be employed for your project, with the Wireguard FPGA project serving as an example case study of an integrated co-simulation and co-development environment.