Wireguard FPGA

Advanced co-simulation verification environment

wyvern semiconductors

Introductions

Simon Southwell

aka Wyvern Semiconductors

Freelance system architect, logic development and software consultant, specializing in co-simulation, HPC and network solutions. Generating and collaborating on open-source projects, mentoring and presenting. Contributor to the OSVVM project, adding and supporting co-simulation capabilities.

simon@anita-simulators.org.uk

LinkedIn: Simon Southwell

Cambridge, United Kingdom

Focus of Today's Presentation

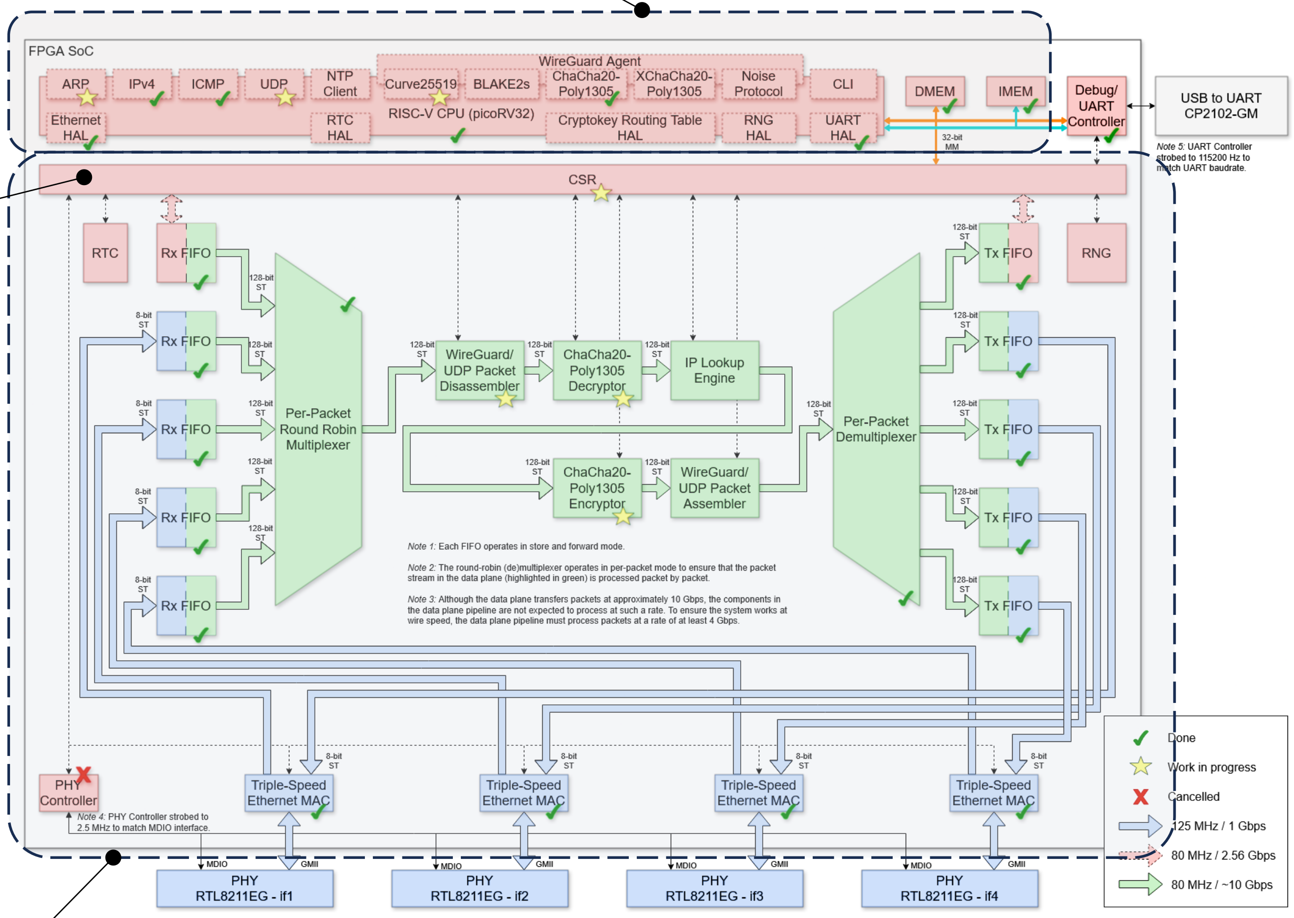
- The main thrust of the presentation will be on the co-simulation used in the Wireguard FPGA project's top level logic simulation
- Will give a very brief overview of the Wireguard FPGA project for context
- A summary of the overall verification architecture
- A look at the open-source individual components to add advance co-simulation capabilities
  - A "Virtual Processor" co-simulation element
  - A spase memory model, in C, integrated into the simulation
  - A RISC-V Instruction Set Simulator, integrated into the Virtual Processor
  - Auto-generation of Hardware Abstraction Layer code from SystemRDL Description
  - An Ethernet UDP/IPv4 C++ model, running on a Virtual Processor to drive the Ethernet Ports
- How all these are combined to give a cohesive top level test bench for co-simulation and co-development

The Wireguard FPGA in a Nutshell

- Implemented by the young engineers at Chili.CHIPS*ba out of Bosnia and with support from the NLnet Foundation
- An open-source SystemVerilog logic implementation of the open-source Wireguard protocol for creating VPNs
- Targets inexpensive hardware platform with four 1000Base-T ports
- Using a commodity Artix7 FPGA
- Done in a self-sufficient way, i.e. without requiring a PC host
- Supported by open-source tools
    - E.g. PipelineC (Julian Kemmerer)
- All gateware written in standard Verilog/ SystemVerilog

soc_cpu, running application

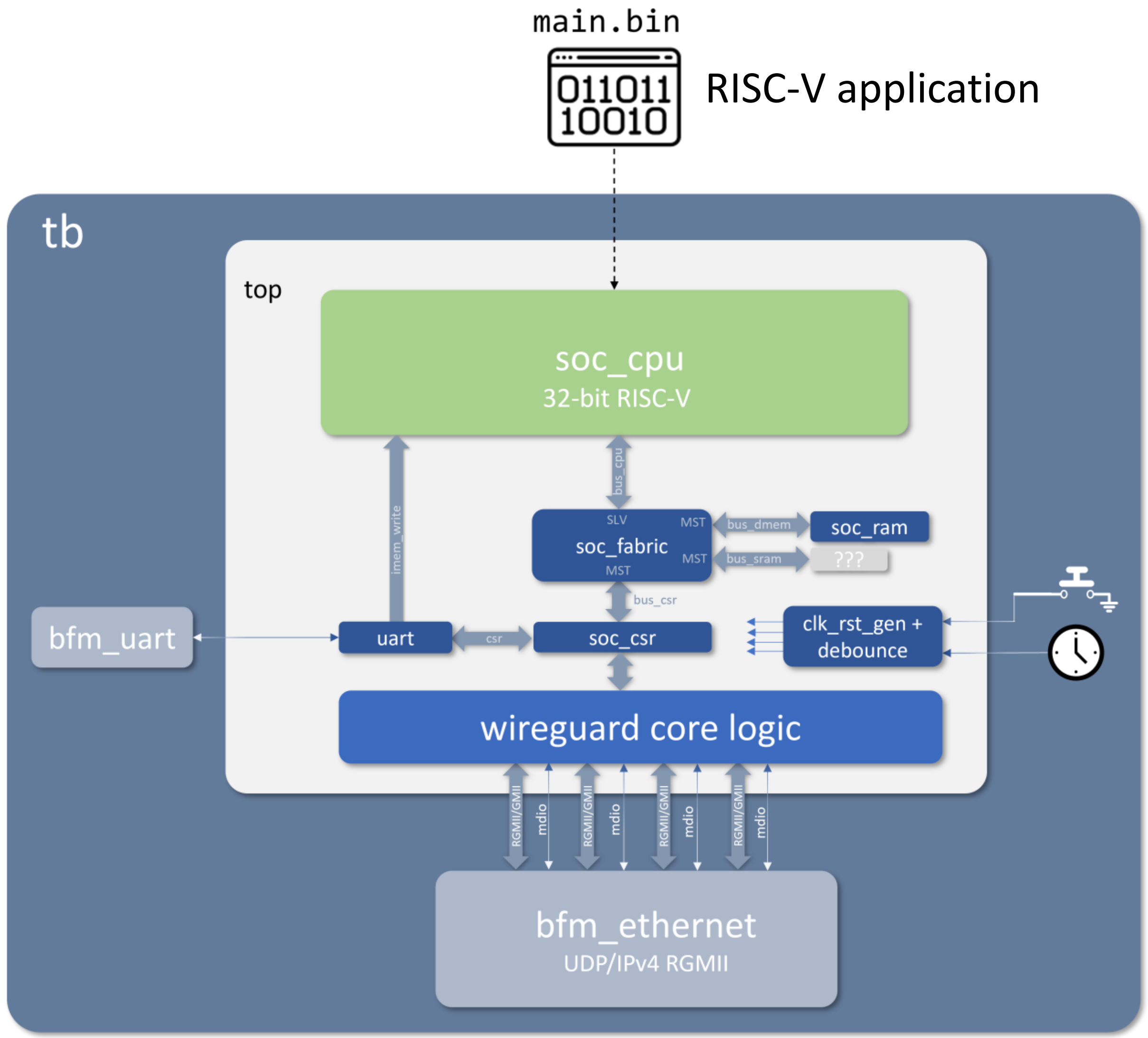
auto-generated control & status regs



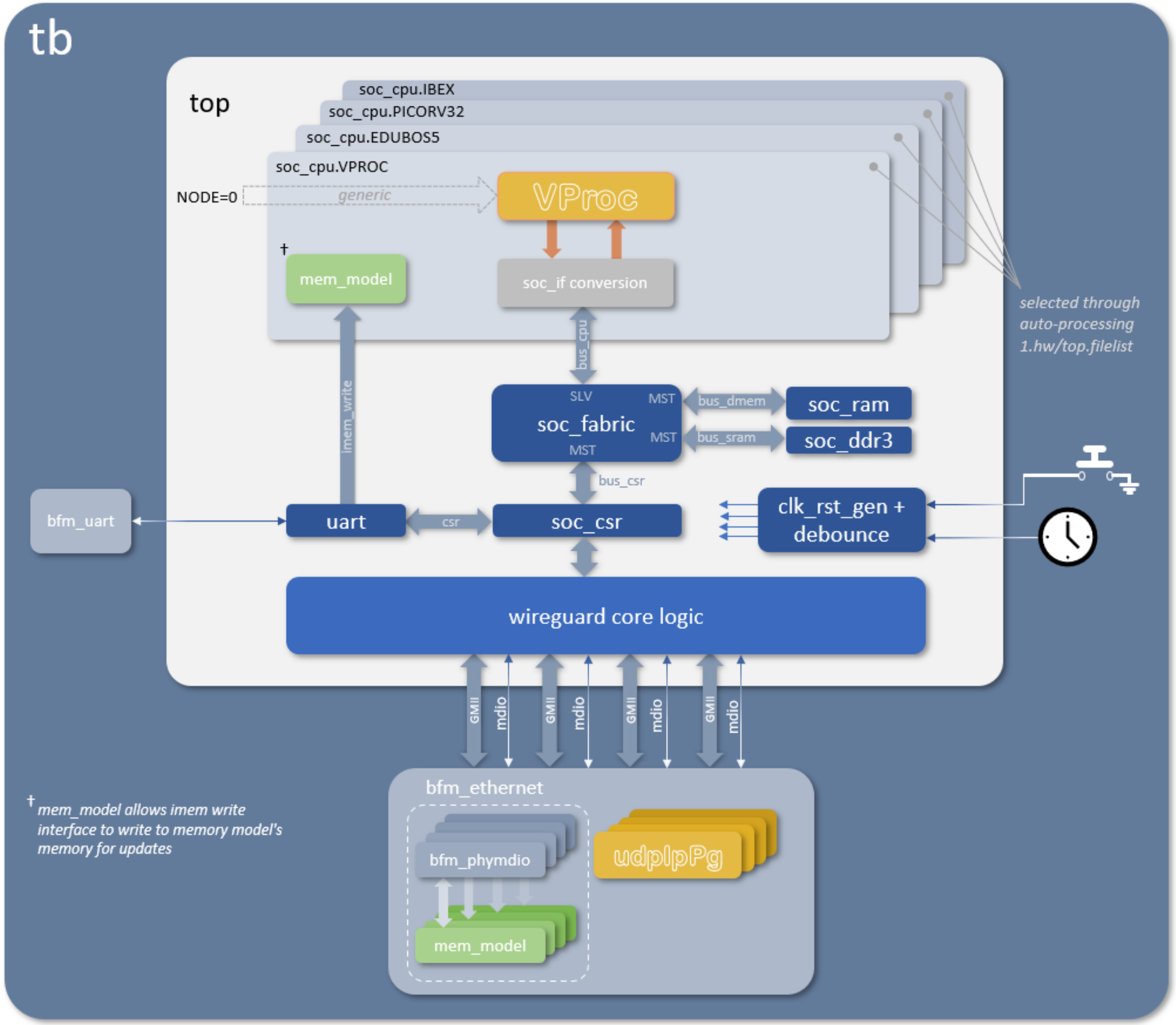
core logic

Wireguard Basic Top Level Test Bench

- Fairly standard top level logic simulation test bench
  - The illustration shows a simplified block diagram
- For the WG project, Verilator (Wilson Snyder et. al.) and SystemVerilog are chosen, with GTKWave (Anthony Bell) or Surfer (Linköping University) for displaying waveforms
- Wireguard's external Interfaces are connected to driver modules
  - Ethernet model with UDP/IPv4 over GMII and MDIO
  - UART that can update code in memory and has some debugging features
- The IP itself has a 32-bit RISC-V processor (`soc_cpu`) with RTL based around the open-source PicoRV32 RISC-V core (Clifford Wolf et. al), though could use others
- RISC-V test code or application software can be loaded and run on the processor
- But there lies some ***hidden secrets*** that extend this generic test bench structure for true co-simulation!



Wireguard *VProc* Based Top Level Test Bench

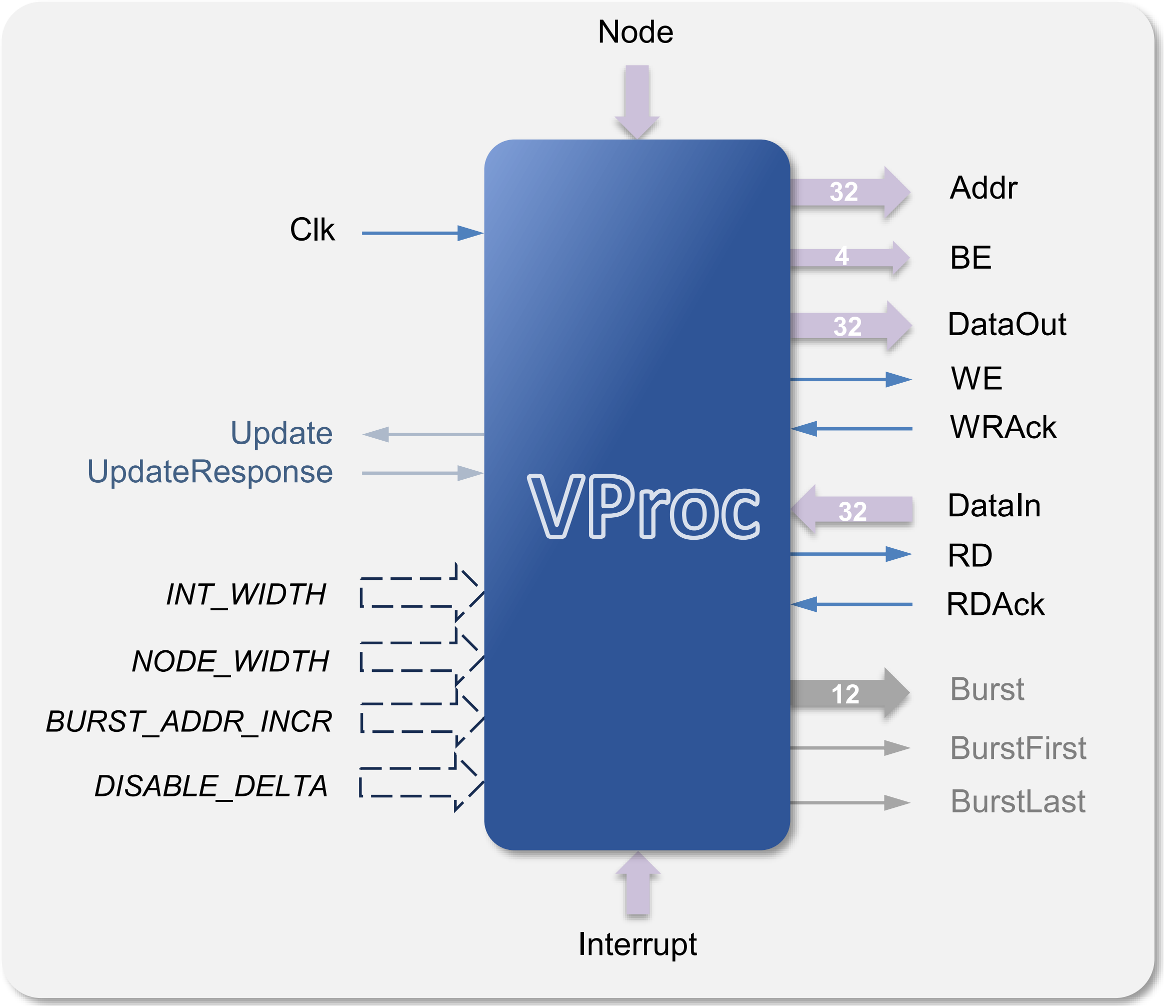


- Means are provided which can allow selection between flavours of `soc_cpu` components between various open source RISC-V RTL cores within the Wireguard FPGA IP
  - examples: PicoRV32, IBEX (lowRISC) and EDUBOS (Tarik Ibrahimovic)
- One selection, though, chooses a "Virtual Processor" known as *VProc* (wyvernSemi)
- Memory is modelled, in C, and "ports" to the model are made available in the test bench as HDL components via "mem_model" (wyvernSemi)
  - Programs running on the virtual processor can access the memory C model directly via its API
- Ethernet is driven by a UDP/IPv4 pattern generator C++ model, based around *VProc—udpPgIp* (wyvernSemi)
- MDIO is driven by HDL but has access to the memory model
- This, then, is our route to true co-simulation and co-development, so let's look at these co-sim components

What is the *VProc* virtual processor?

- Can "run" a normal user C or C++ program, compiled for the host machine (PC or Linux Workstation), which can drive an address bus on an HDL component instantiated in a logic simulation
- The *VProc* HDL 'processor' Component has a generic memory mapped master interface
  - For WG, wrapped in an **`soc_cpu`** BFM for specific '**`soc_if`**' bus protocol
  - Optional Burst ports. Not used for Wireguard.
  - Supports interrupts. Not used for Wireguard
  - Has 'delta cycle' update capability (more later)
- In WG, DPI-C is used to connect between HDL and C/C++
  - Other simulators and programming interfaces supported
  - VHDL also supported
- Provides a C or C++ API to drive address bus
- Can instantiate multiple *VProcs*
  - Called 'nodes'.
  - Each *must* have a unique node number on **`Node`** port
- Each node's software has a specific "`main`" entry point for user code
  - E.g. **`VUserMain0()`** for node 0
  - c.f. `WinMain` for Windows graphics programs

The *VProc* HDL component



VProc's Software API

- User writes normal C or C++ code that is compiled into a static library and linked to the *Verilator* executable
  - Other simulators use a shared object loaded at run-time
- The basic C and C++ APIs:
  - Low Level C API, e.g.

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

  - C++ *VProc* class, e.g.

```
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);
```

- Using the API directly is suitable for writing custom tests to drive bus of `soc_cpu` and reading and writing registers or memory etc.
  - But we can do better then that!

Simple use of *VProc*'s 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);
}
```

Sparse Memory Model (*mem_model*)

- A sparse memory model in C is connected to an HDL component: [mem_model]()

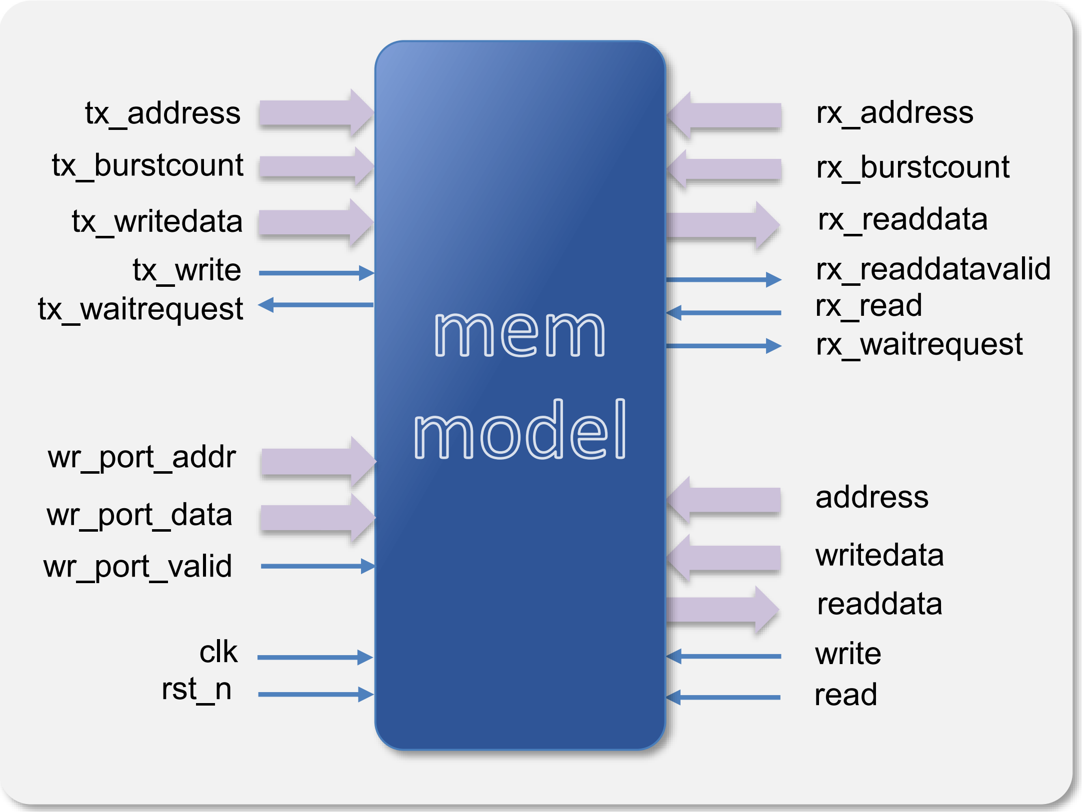
- *VProc* user code can access memory directly using the memory model's C API (`mem.h`). E.g.

```
WriteRamByte  (addr, data,     memnode)
WriteRamHword (addr, data, le, memnode)
WriteRamWord  (addr, data, le, memnode)
ReadRamByte   (addr,           memnode)
ReadRamHword  (addr,       le, memnode)
ReadRamWord   (addr,       le, memnode)
```

- The sparse memory model in C can implement up to a 2^{64} byte address space

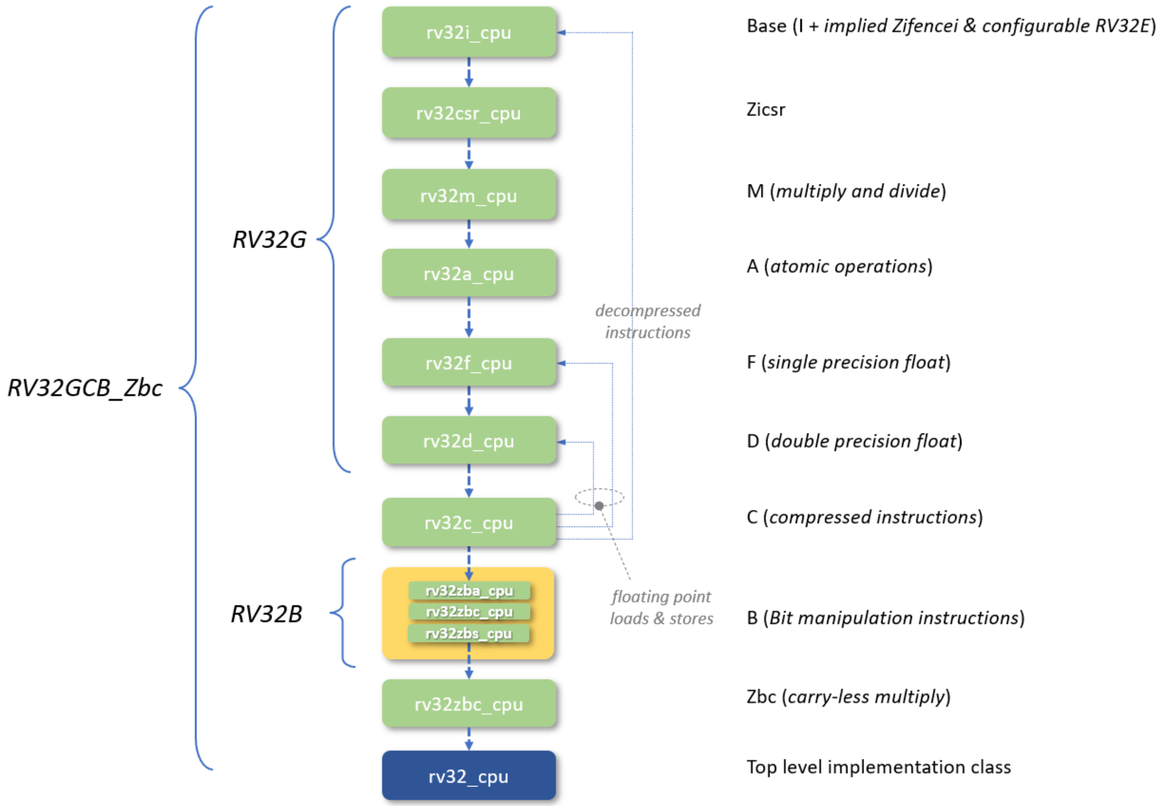
- Connected to HDL with DPI for *Verilator* in this project
  - Gives access to C memory model from HDL
  - Module has various Altera Avalon style ports

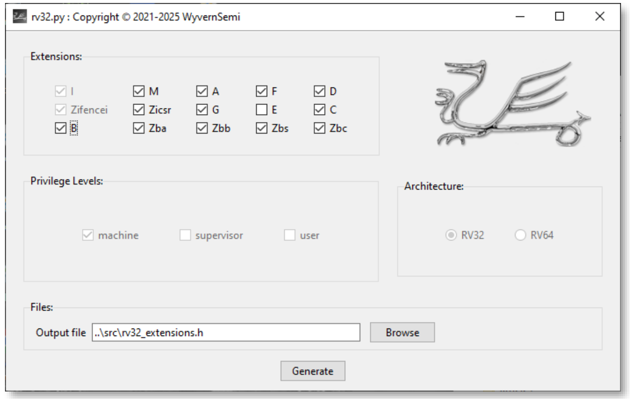
- Can instantiate multiple `mem_model` components
  - NB: Each is a 'port' to the <u>*same address space*</u> (`memnode = 0`), *not* a new memory



The *rv32* RISC-V Instruction Set Simulator

- One of the "Programs" that can be run on the Virtual Processor is a C++ model of a RISC-V core
- *rv32* (wyvernSemi) is a C++ instruction set simulator model (ISS) for RISC-V, configurable up to RV32GCBE_Zicsr_Zbc specifications
  - Can be configured to implement subsets of this (min RV32I)
  - Each RISC-V extension has own class and inherits the derived class before it to build up to a certain specification
- Can be compiled as a standalone executable or as a library for integration with other code (as done for WG)
- Can register external callback function, called whenever a memory access is made. Callback can decide to processor or hand back.
- Has GDB remote debug interface
- Has internal instruction timing model that can be configured to model different RISC-V cores' instruction timings
  - E.g. the PicoRV32 used in Wireguard FPGA
- On Wireguard, the ISS can be selected as the user code to run on the **`soc_cpu.VPROC`** component
  - A simple software layer is required to integrate with simulation (more later)
  - The RISC-V compiled application code is then run on the ISS





HAL Auto-generation for Co-simulation

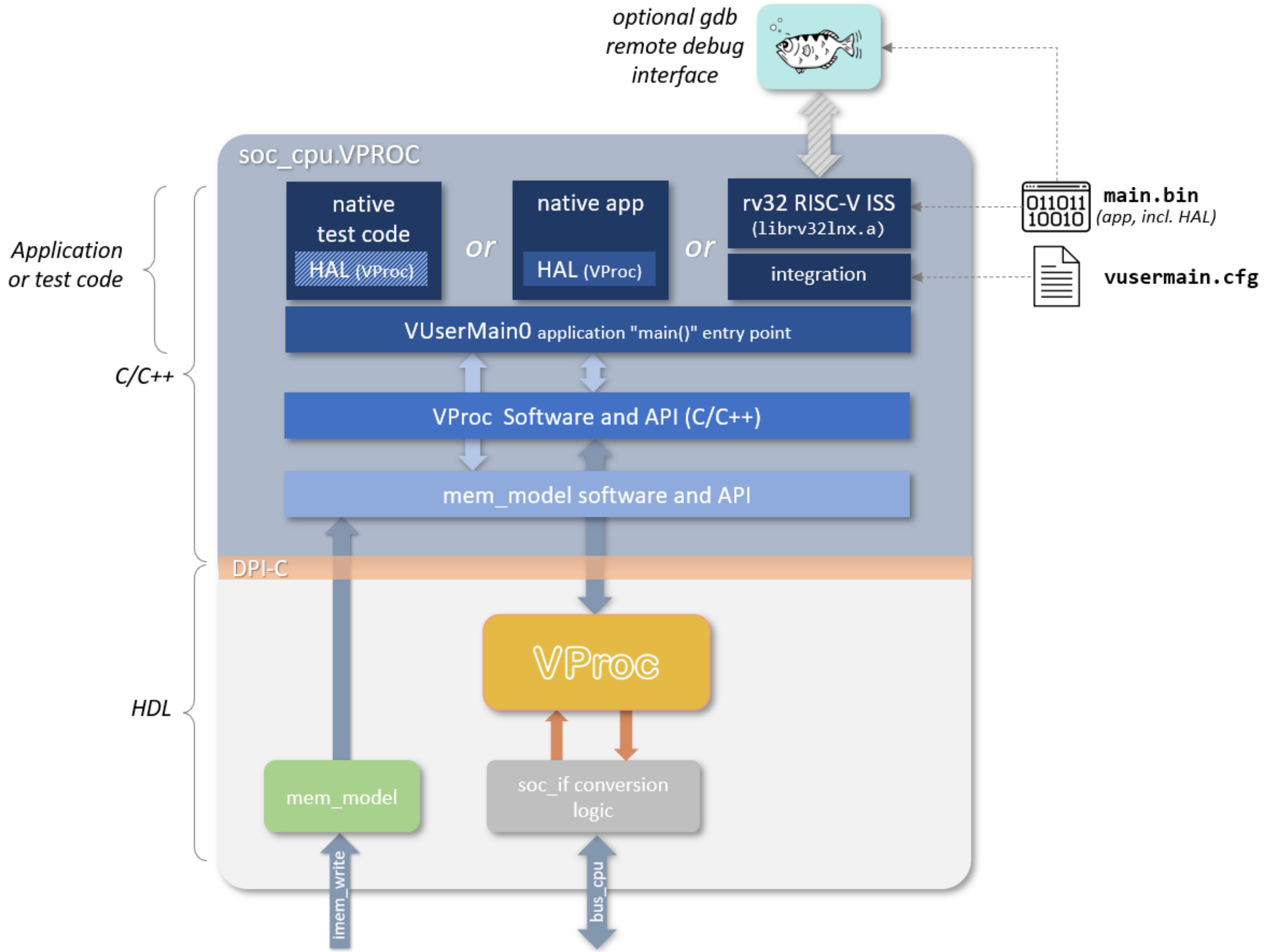
- Wireguard uses a SystemRDL (an Accellera standard) description of its CSR registers
- SystemRDL description parsed by the open-source *peakrdl* tools (Alex Mykyta et. al) :
  - Used to generate CSR RTL
  - Used to generate S/W hardware abstraction layer (HAL)
- Would like to compile the target application *natively* for host computer
  - Present the *same* HAL API to both native and target compiled code
  - Target (RISC-V) HAL calls *peakrdl* output
  - Native HAL calls *VProc* API functions
  - *systemrdl-compiler*, bundled with *peakrdl* tools, is used to generate the two types of co-simulation HAL
- With this, the Wireguard application can be developed in tandem with RTL without the need to model RISC-V
  - Will run faster
  - Functionally the same
  - Timings, though, will be approximate but good for fast functional verification
- Why not use the peakrdl output for both?
  - Use of bitfields negates overloading techniques previously used on other HALs

PeakRDL-regblock

PeakRDL-cheader

systemrdl-compiler

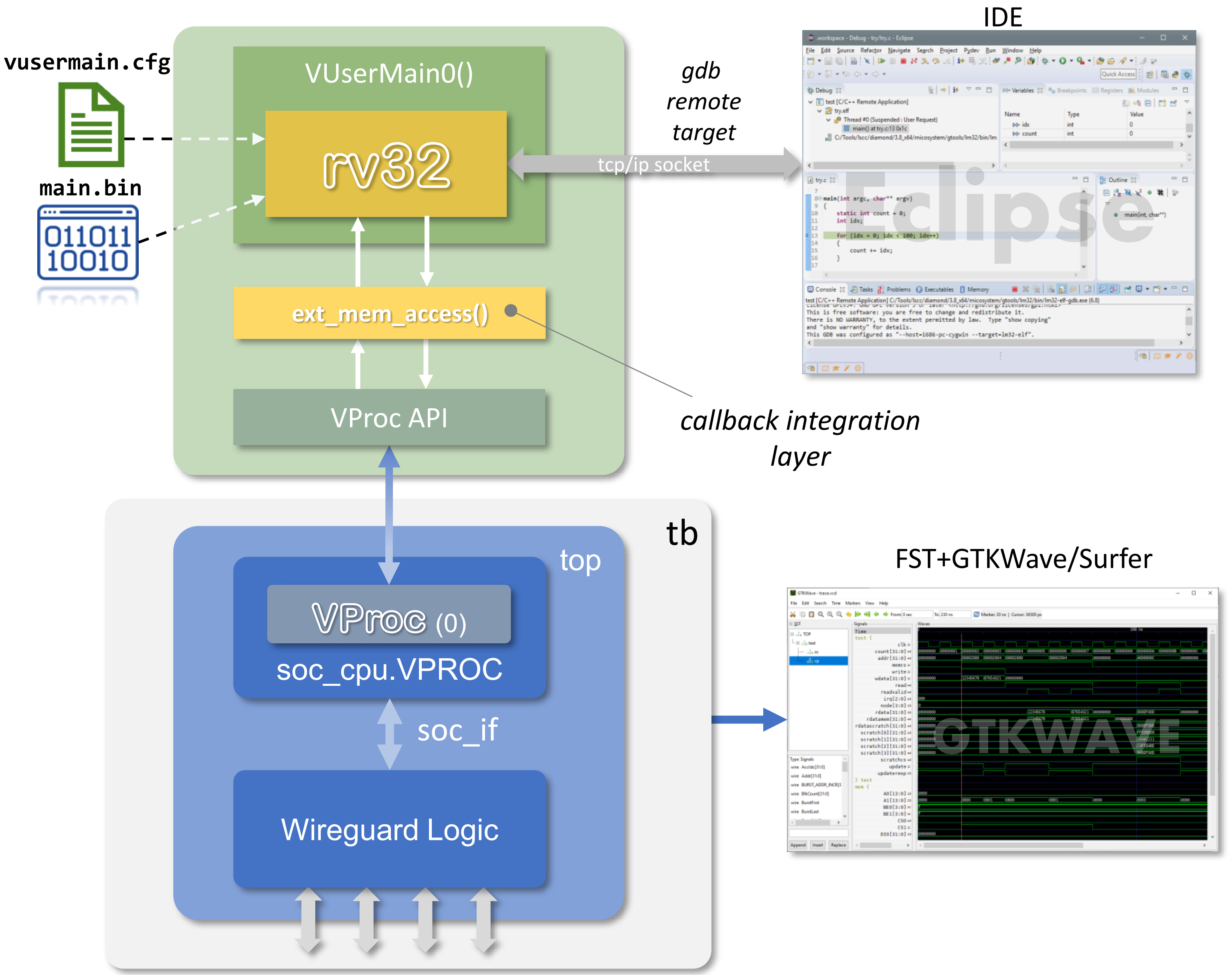
Putting it All Together for soc_cpu.VProc



- Three use cases available for the TB:
  - Native test code to drive `soc_cpu` bus using *VProc* API directly, as discussed previously (or HAL)
  - Application software compiled natively with modified VProc HAL (same API as code compiled for RISC-V target)
  - *rv32* RISC-V ISS and integration code to run application for target.
- Debugging code a non-negotiable requirement
  - Native code can be debugged with normal gdb for host programs
  - RISC-V code can be debugged with remote gdb features of *rv32* ISS
- The *rv32* ISS can be configured at run time via a `vusermain.cfg` file
- Uses internal sparse memory model
  - *mem_model* HDL components give access from logic
  - *VProc* code can use it's direct C API
  - Thus software and hardware share the same memory space

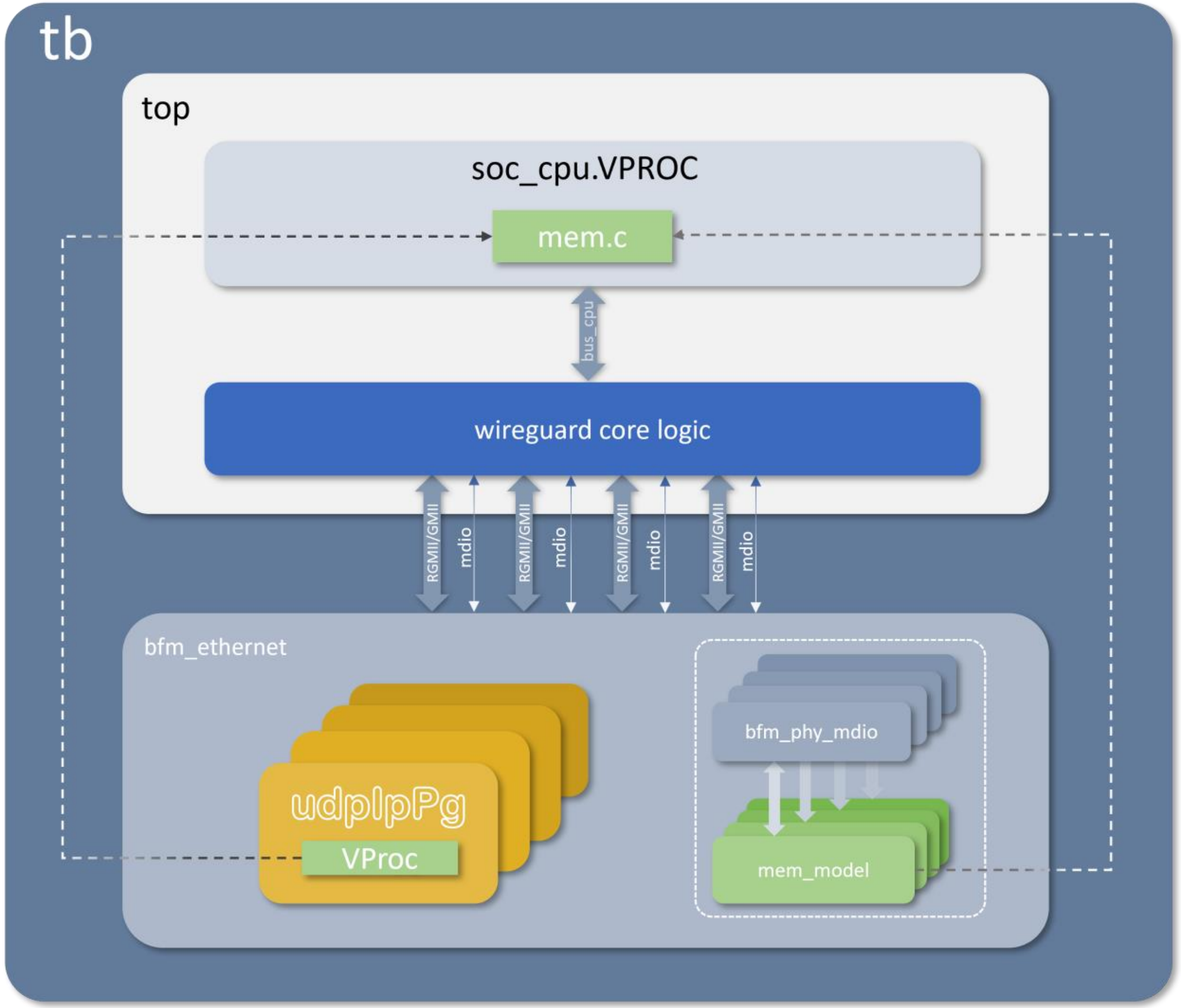
Integrating *rv32* ISS into Wireguard TB

- rv32 has a memory access callback feature
  - All loads/stores are 'offered' to a registered user callback function, along with the model's reckoning of time
  - Callback makes appropriate *VProc* read and write calls and to `'VTick'` to synchronise simulator with CPU's sense of time
  - If callback doesn't process access, returns access back to *rv32* to process
- Wireguard TB co-sim integration layer is implemented in this callback
  - If access within a configured range, makes accesses to CSR registers via *VProc* API
  - Otherwise uses *mem_model* C API to store in common address space



Ethernet UDP/IPv4 Driver VIP

- Based on *udpIpPg* model (wyvernSemi) built around *VProc*
  - Developed from the *tcpIpPg* model (wyvernSemi)
- Consists of a C++ model to generate and decode UDP/IPv4 ethernet packets for Gigabit Ethernet and run on a *VProc* component.
- *VProc* can be used to memory map ports on a module and drive multiple signals using its delta cycle feature
  - Any arbitrary interface can thus be driven
  - A small layer provides an API to the component ports
- The HDL components have GMII interfaces
  - An RGMII converter module is also provided
- An MDIO BFM HDL module is also available
  - Uses *mem_model* component to read from and write to (when receiving register accesses) configured segments of memory
- The *VProc* software of *udpIpPg* and the *mem_model* used by the MDIO BFM both have access to the same memory space that `soc_cpu.VPROC` has
  - The test programs on soc_cpu.VPROC can now set patterns used by the other models or read data written by them, closing the loop for end-to-end verification



Conclusions

- From a starting point of a "classic" test bench architecture, have carefully deployed various readily available open source tools to make this into a true co-simulation and co-development verification environment

- A "Virtual Processor" provided the means to seamlessly bridge the HDL to C/C++ gap to "run" a natively compiled user program on an HDL processor core in a logic simulation

- A C memory model co-sim component gave a common memory space between HDL and user software

- A RISC-V instruction set simulator was integrated with VProc to allow the target's application code to be run in simulation as well as on the platform

- An auto-generated HAL, with a common API for target and VProc allowed the application to be compiled and run natively

- A soft UDP/IPv4 model was deployed to drive the ethernet ports, reutilising the virtual processor

- All this allows various options to be utilised, trading speed of execution, versus accuracy of timing
  - RTL processor (PicoRV32) with full timing accuracy but slow
  - *rv32* ISS, high timing accuracy, but fast
  - Natively compiled application, lower accuracy but faster
  - Test specific code, no accuracy, fastest

- None of the open-source tools we've looked at dictate any specific methodology
  - The technology behind *VProc* has been used to add co-simulation capabilities to OSVVM, giving access to a richer featured test environment (in VHDL), if working in that environment

- All of the tools mentioned don't rely on any of the others (mostly)
  - *mem_model* almost always goes with *VProc*, but it's not obligatory
  - *udpIpPg* is based around *VProc*
  - the *rv32* ISS can be compiled as an executable, and as a static library for integration into other environments such as C++ SoC models or SystemC etc.

- There are a rich set of open-source tools available, with some advanced capabilities, on which to develop you're project. We have looked at just a few today, integrated together to get something bigger than the sum of their parts.

Open Source Components and More Information

- The Wireguard FPGA project (Chili.CHIPS*ba Wireguard FPGA team)
  - Support from NLnet Foundation
- *mem_model* co-simulation sparse memory model (Wyvern Semiconductors)
- *VProc* virtual processor co-simulation VIP (Wyvern Semiconductors)
  - Article on *VProc*, its use and how it functions
  - PLI article with *VProc* and *mem_model* as examples
- The *rv32* RISC-V instruction set simulator (Wyvern Semiconductors)
  - Article on *rv32* and its internal architecture
  - Article on *rv32*'s gdb remote debug interface
- *udpPgIp* gigabit ethernet UDP/IPv4 RGMII/GMII co-simulation model (Wyvern Semiconductors)
  - Based on *tcpIpPg* (Wyvern Semiconductors)
- *peakrdl* and *systemrdl-compiler* (Alex Mykyta et. al)
- *Verilator* Verilog and SystemVerilog cycle based logic simulator (Wilson Snyder et. al)
- *GTKwave* (Anthony Bybell) and *Surfer* (Linköping University) waveform viewers

END

Backup Slides

peakrdl c-header CSR access output

- Provides the macros for masking and shifting
- Provides a bitfield structure of the fields within a register
- Union for structure or whole register access
- Provides hierarchical structures (not shown)

```
#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;
```

Co-simulation HAL for Target (RISC-V)

- Python script written to generate common HAL using *systemrdl-compiler*
- For target, wraps *peakrdl* output in simple inline function calls
  - Hierarchical dereferencing
  - Function name matches field
  - Can access 'full' register
- Why not use *peakrdl* output directly?
  - We need a common API for native vs target
  - Bitfields can't be used in 'classic' method of overloading accesses to a given type to call *VProc* API functions

```
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;
};
```

```
#include "csr_hw.h"

int main (int argc, char** argv)
{
  // Create a CSR register object
  csr_vp_t* csr = new csr_vp_t(CSR_BASE_ADDR);

  csr->ip_lookup_engine->table[0]->allowed_ip[0]->address(0x12345678);
  uint32_t val = csr->ip_lookup_engine->table[0]->allowed_ip[0]->address());

  csr->ip_lookup_engine->table[0]->allowed_ip[0]->mask(0x65ef9078);
  val = csr->ip_lookup_engine->table[0]->allowed_ip[0]->mask();
}
```

Co-simulation HAL for *VProc*

- Wireguard's Python script can also generate a *VProc* HAL

- Produces the same functions to access the registers, fields and hierarchy
  - Usage is exactly the same

- Now *VProc* API calls are made to do the reads and writes
  - Uses the *peakrdl* generated macros for masks and shift of data for correct access alignment.
  - Will mimic write only access for aligned fields, doing read-modify-writes only for misaligned fields.

- Other considerations for native compilation of application include wrapping up delay functions
  - In native application make calls to `VTick()` to advance simulation time

```
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;
}
```