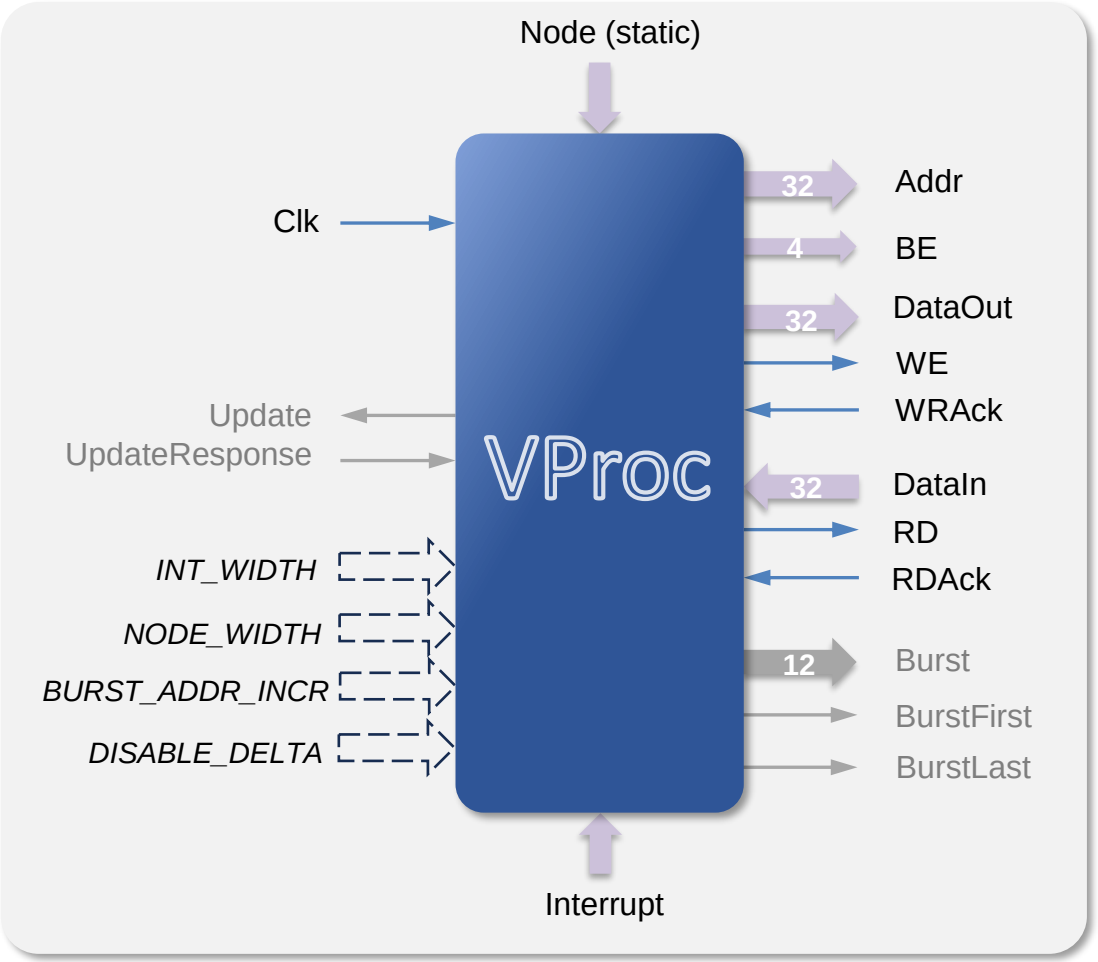
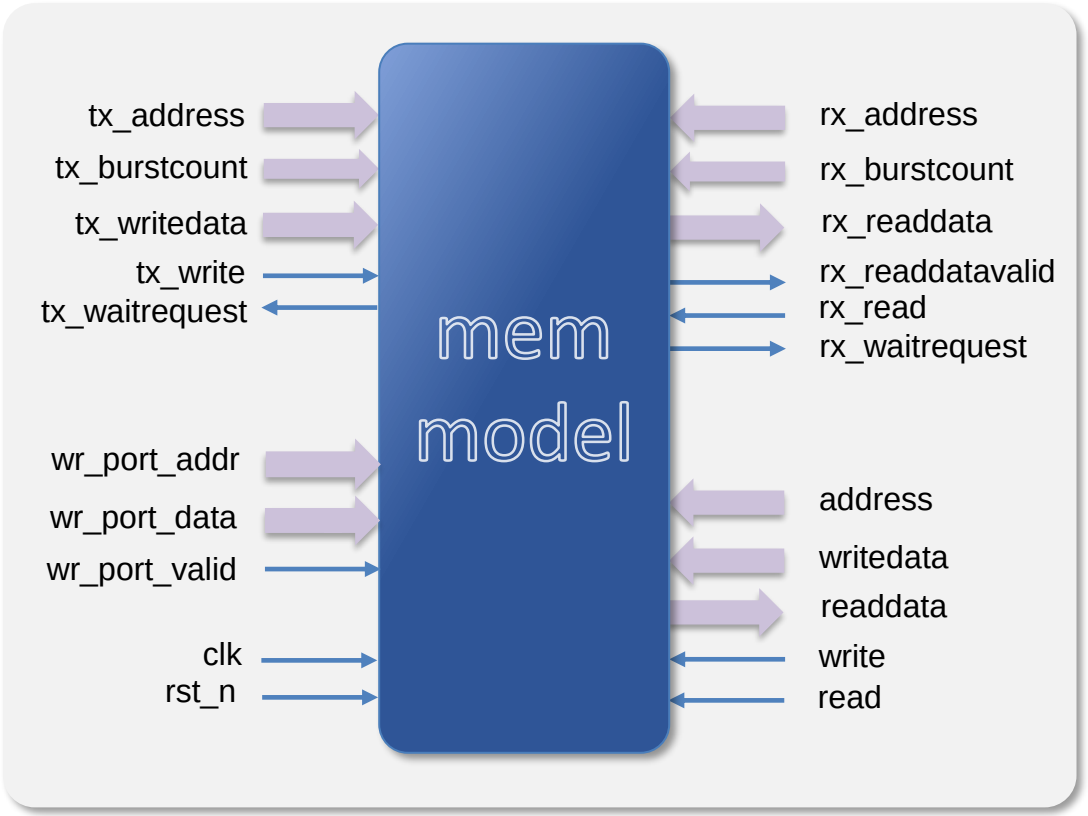
The *VProc* Virtual Processor

- *VProc* is Verification IP to run host compiled code in a logic simulation
- HDL 'processor' module with generic memory mapped master interface
  - Burst ports optional. Not used for WireGuard.
  - Usually wrapped in a BFM for specific bus protocol
  - Supports interrupts
  - Has 'delta cycle' update capability. Disabled for WireGuard
- Uses PLI/VPI/DPI (and VHDL equivalents) to:
  - Connect HDL to C domain
  - Provides software layer to run user code on host machine and:
    - Synchronise code with simulation
    - Generate read and write accesses over HDL bus
    - Provide a C or C++ API to user code to generate transactions in simulation
- Can instantiate multiple *VProcs*
  - Each ***must*** have a unique node number
- Each 'node' has a specific "main" entry point
  - E.g. `VUserMain0()`
  - c.f. `WinMain` for Windows graphics programs



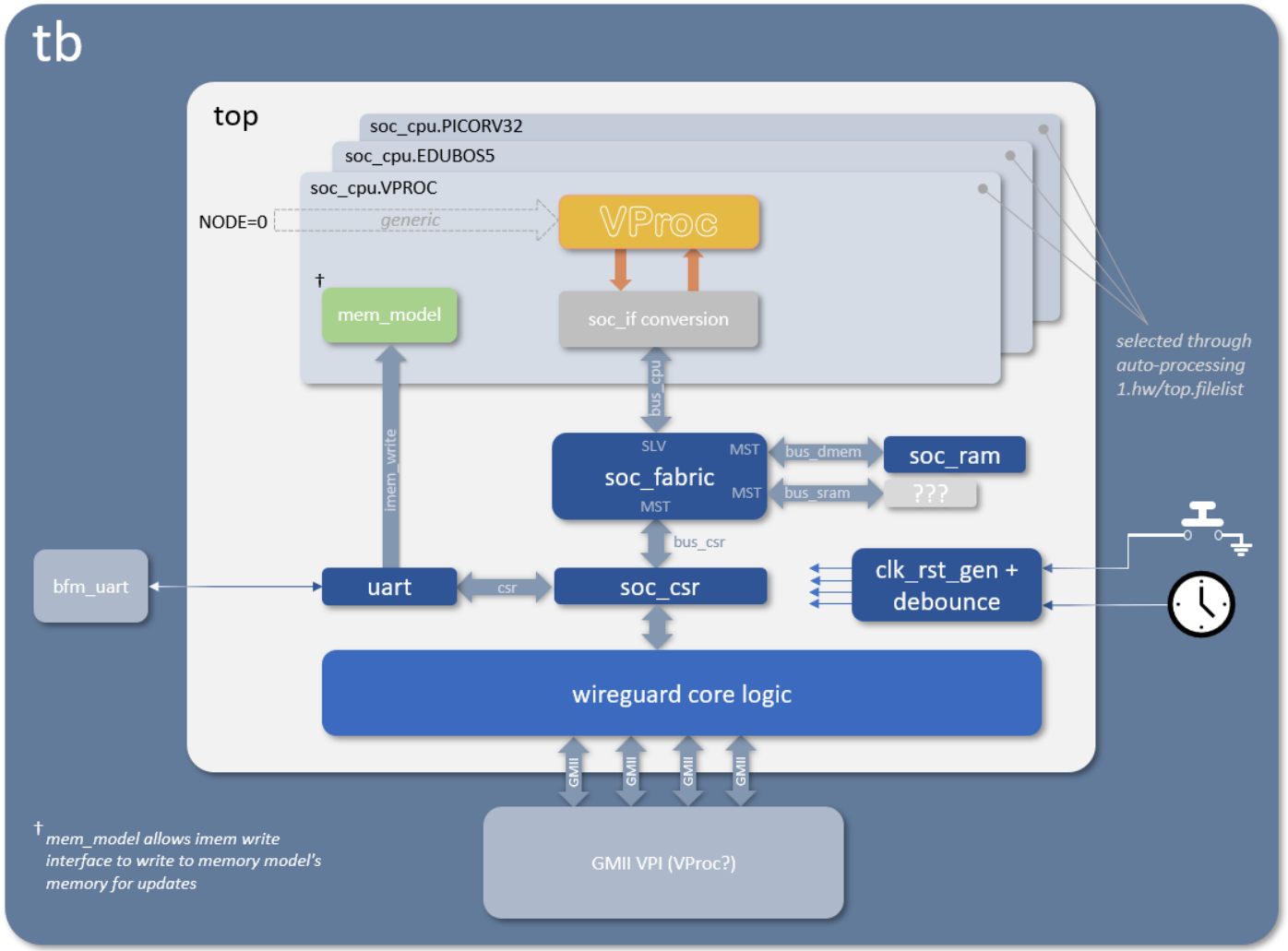
Sparse Memory Model (mem_model)

- Closely associated with *VProc*
  - *VProc* user code can access memory directly using Mem Model's C API
- A sparse memory model in C implements 2^{64} address space
- Connected to HDL with PLI/VPI/DPI
  - Module has various Altera Avalon style ports
- Can instantiate multiple `mem_model` modules
  - Each is a 'port' to the *same address space*, not a new memory



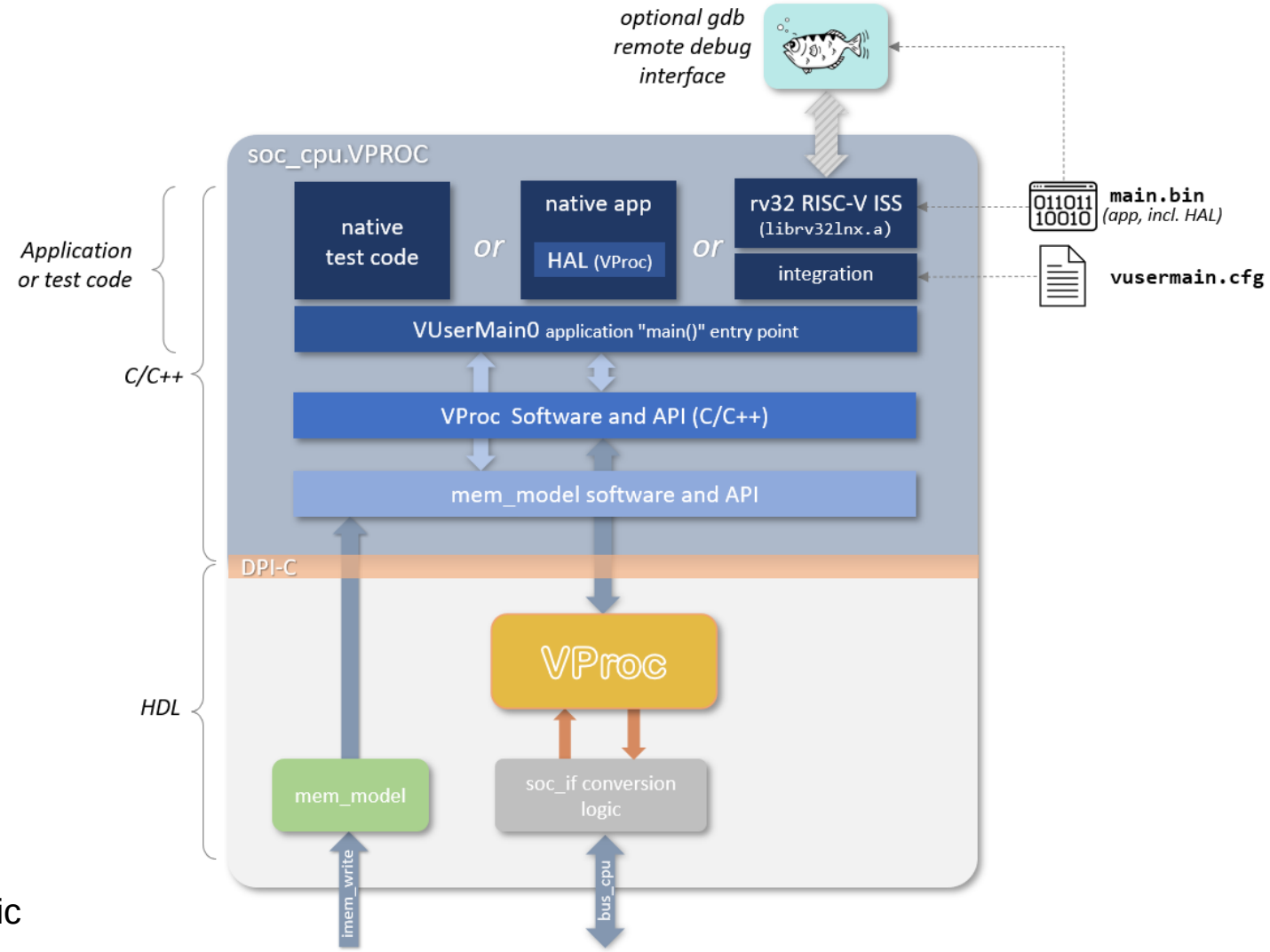
WireGuard Top Level Test Bench HDL

- Took existing TB and made some slight changes to get to compile
- Created `soc_cpu` BFM wrapper for *VProc* to convert to `soc_cpu` interface
- Write input port connected to a `mem_model` component for access to internal memory
- `top.filelist` processed to remove any other `soc_cpu` file references and replaced with VProc files
  - Could be made to select from any number of compatible components
- Driving of GMII ports as yet undefined
  - A modified `tcpIpPg` (10Gbe/XGMII) under consideration.
  - *VProc* based soft model
  - TCP, not UDP
  - XGMII not GMII
  - No MDIO



WireGuard *VProc* soc_cpu and S/W Stack

- *VProc* runs natively compiled code which can drive `soc_cpu` memory mapped bus in simulation.
- Three use cases proposed:
  - Native test code to drive `soc_cpu` bus using *VProc* API
  - Application compiled natively with modified HAL (same interface)
  - `rv32` RISC-V ISS and integration code to run application for target.
- Debugging
  - Native code can be debugged with normal gdb for host programs
  - RISC-V code can be debugged with remote gdb features of `rv32` ISS
- `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



Building and Running *VProc* code and TB

- In `4.sim/` directory is a make file:
  - `MakefileVProc.mk`

- A target of 'run' or 'gui' will build everything and execute the simulation
  - Requires *VProc* and *Mem Model* repositories
  - Make file will check these out if don't exist

- By default it will compile user code from `4.sim/usercode/` directory
  - Uses *VProc*'s make file to build *VProc* and user software into `libvproclnx.a` static library.
  - Library linked with *Verilator* code

- Configurable by setting make variables on the command line
  - list of user code and containing directory
  - C flags and *Verilator* flags
  - Name of RTL top file list
  - Set build type: ISS or native (default)

```
$ make -f MakefileVProc.mk help
make -f MakefileVProc.mk help          Display this message
make -f MakefileVProc.mk               Build C/C++ and HDL code without running simulation
make -f MakefileVProc.mk run           Build and run batch simulation
make -f MakefileVProc.mk rungui/gui    Build and run GUI simulation
make -f MakefileVProc.mk clean         clean previous build artefacts
make -f MakefileVProc.mk deepclean     clean previous build artefacts and checked out repos

Command line configurable variables:
  USER_C:       list of user source code files (default VUserMain0.cpp)
  USRCODEDIR:   directory containing user source code (default $(CURDIR)/usercode)
  OPTFLAG:      Optimisation flag for VProc code (default -g)
  TIMINGOPT:    Verilator timing flags (default --timing)
  TRACEOPTS:    Verilator trace flags (default --trace-fst --trace-structs)
  TOPFILELIST:  RTL file list name (default top.filelist)
  SOCCPUMATCH:  string to match for soc_cpu filtering in h/w file list (default ip.cpu)
  USRSIMOPTS:   additional Verilator flags, such as setting generics (default blank)
  WAVESAVEFILE: name of .gtkw file to use when displaying waveforms (default waves.gtkw)
  BUILD:        Select build type from DEFAULT or ISS (default DEFAULT)
  TIMEOUTUS:    Test bench timeout period in microseconds (default 15000)

make: 'help' is up to date.
```

Native Test Code Build Usage Case

- Normal C++ code that is compiled into a library and linked to *Verilator* executable
  - Other simulators need a shared object, loaded at run-time, but same code.
- Uses the *VProc* API directly
  - Low Level C API, e.g.
    - `VWrite(addr,  data, delta, node);`
    - `VRead (addr, *data, delta, node);`
  - C++ *VProc* class, e.g.
    - `vp0->write(addr,  data, delta=0);`
    - `vp0->read (addr, *data, delta=0);`
- Suitable for custom tests to drive bus of `soc_cpu`, reading and writing registers
  - Have access to all libraries and features of native C/C++ program
- **Available now**

```
#include "VProcClass.h"
extern "C" {
#include "mem.h"
}

static const int node    = 0;

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

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

    uint32_t addr  = 0x10001000;
    uint32_t wdata = 0x900dc0de;

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

    vp0->tick(3);  // Emulate some processing

    uint32_t rdata;
    vp0->read(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);
}
```

Native Application Build Use Case

- Application compiled for Host

- Augmented HAL
  - But with same API as for target
  - Application code unmodified
  - Code runs faster than on an ISS

- Must wrap timing functions
  - Delays/Sleep
  - *VProc* API 'ticks' to advance simulation time

- HAL API used to access registers and their fields *something* like that shown
  - Using systemrdl-compiler to generate *VProc* and Application HAL 'adapter'
    - Part of SystemRDL like *peakrdl*
  - Platform HAL uses the *peakrdl* c-header output
  - systemrdl-compiler issue with 'buffer_writes'

- Ongoing but have initial output for target

```
#include "csrvp.h"

csr_t *regs = new csr_t(regbaseaddr);

int main (int argc, char **argv)
{
    // Read register fields through HAL
    printf("version %d.%d\n", regs->version->major(),
                              regs->version->minor());

    // Write whole register through HAL
    regs->scratch->w(0x12345678);

    // Read whole register through HAL
    printf("scratch = 0x%08X\n", regs->scratch->w());

    return 0;
}
```

RISC-V ISS simulation Usage

- RISC-V code compiled as for target to generate an executable binary file

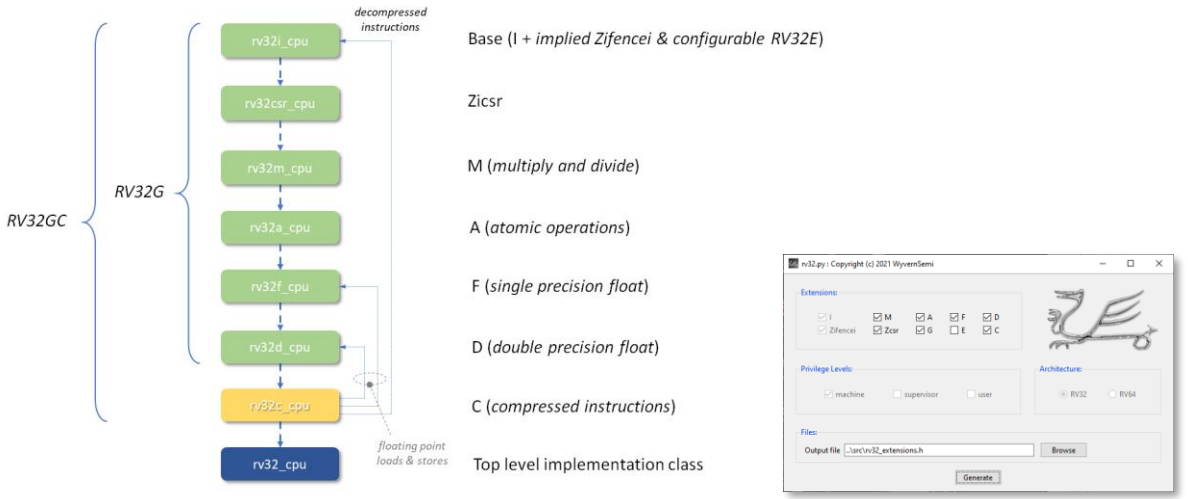
- To build simulation with `rv32` ISS, use make file with a build of ISS
  - `make -f MakefileVProc.mk BUILD=ISS [run|gui]`

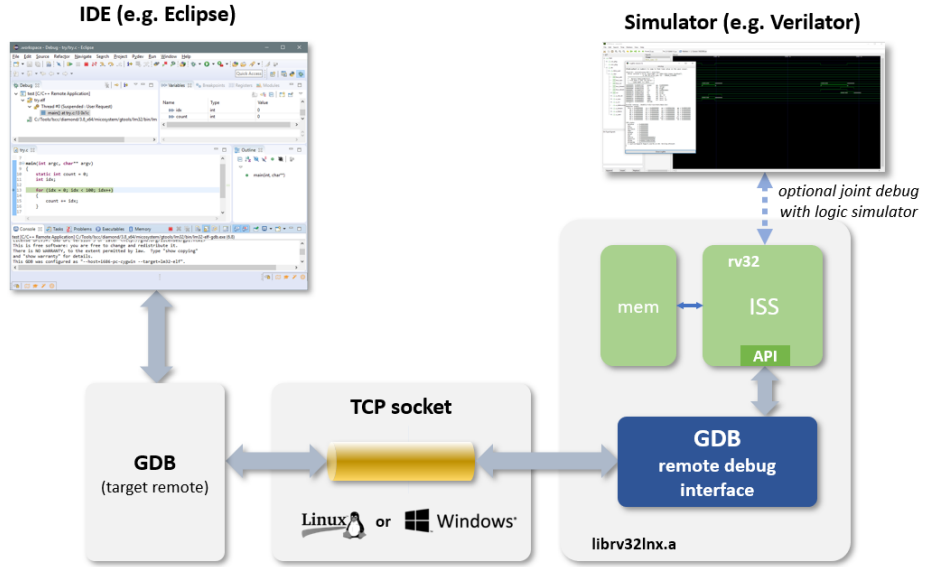
- All 'user' code is in `4.sim/models/rv32/usercode`
  - This code is the ISS integration layer
  - The make file will automatically use this code when build is 'ISS'
  - Links with a pre-compiled rv32 ISS static library (`librv32lnx.a`)
  - A local `README.md` in `rv32` directory documents this code

- The ISS and integration code is configured with a `vusermain.cfg` file
  - Including specifying the binary to run

- RISC-V application can be debugged using the `rv32`'s remote debug interface
  - The ISS's manual also details how to setup an Eclipse IDE.

- **Available Now**



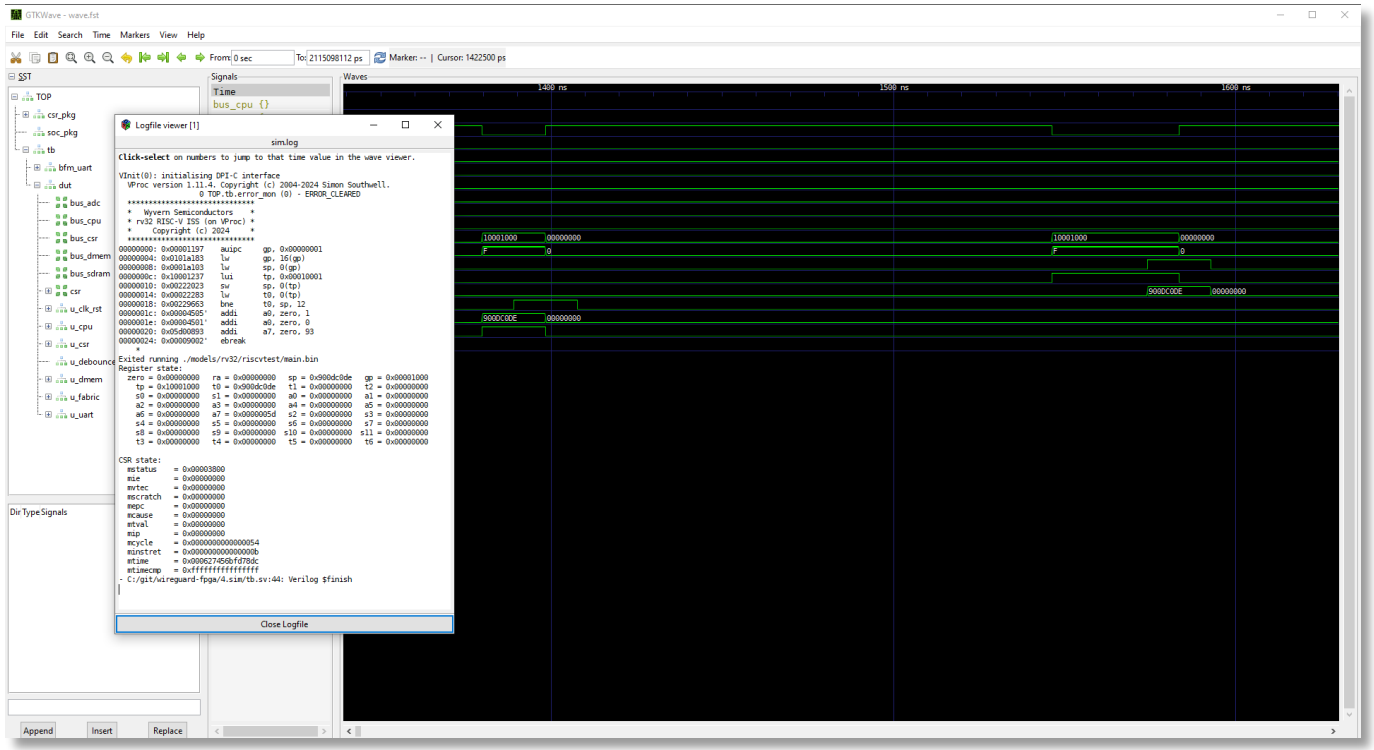


Configuring the ISS with vusermain.cfg

- Command line option style
  - Standalone ISS can use command line options
  - Can't access command line from *Verilator*
- Number of options grown for WireGuard
  - Select RISC-V binary executable
  - Specify start address
  - Specify 'halt' conditions (if desired):
    - After executing a number of instructions
    - On executing **`ecall`** or **`ebreak`**
    - Decoded unimplemented instruction
    - Specific address
  - Dump registers on exit
  - Run-time disassembly
  - Start address
  - gdb remote port#
  - Cache timing enable and configuration
  - Select core timing model to use
  - Region of memory map for simulation accesses
    - Else uses `mem_model` direct API (faster)

```
Usage:vusermain0 -t <test executable> [-hHebdrgxXRcI][-n <num instructions>]
      [-S <start addr>][-A <brk addr>][-D <debug o/p filename>][-p <port num>]
      [-l <line bytes>][-w <ways>][-s <sets>][-j <imem base addr>][-J <imem top addr>]
      [-P <cycles>][-x <base addr>][-X <top addr>][-V <core>]
   -t specify test executable (default test.exe)
   -n specify number of instructions to run (default 0, i.e. run until unimp)
   -d Enable disassemble mode (default off)
   -r Enable run-time disassemble mode (default off. Overridden by -d)
   -C Use cycle count for internal mtime timer (default real-time)
   -a display ABI register names when disassembling (default x names)
   -T Use external memory mapped timer model (default internal)
   -H Halt on unimplemented instructions (default trap)
   -e Halt on ecall instruction (default trap)
   -E Halt on ebreak instruction (default trap)
   -b Halt at a specific address (default off)
   -A Specify halt address if -b active (default 0x00000040)
   -D Specify file for debug output (default stdout)
   -R Dump x0 to x31 on exit (default no dump)
   -c Dump CSR registers on exit (default no dump)
   -g Enable remote gdb mode (default disabled)
   -p Specify remote GDB port number (default 49152)
   -S Specify start address (default 0)
   -I Enable instruction cache timing model (default disabled)
   -l Specify number of bytes in icache line (default 8)
   -w Specify number of ways in icache (default 2)
   -s Specify number of sets in icache (default 256)
   -j Specify cached IMEM base address (default 0x00000000)
   -J Specify cached IMEM top address (default 0x7fffffff)
   -P Specify penalty, in cycles, of one slow mem access (default 4)
   -x Specify base address of external access region (default 0xFFFFFFFF)
   -X Specify top address of external access region (default 0xFFFFFFFF)
   -V Specify RISC-V core timing model to use (default "DEFAULT")
   -h display this help message
```

Demo


GTKWave - wave.fst
File Edit Search Time Markers View Help
From: 0 sec
To: 2115098112 ps
Marker: -- | Cursor: 1422500 ps
SST
TOP
csr_pkg
soc_pkg
tb
bfm_uart
dut
bus_adc
bus_cpu
bus_csr
bus_dmem
bus_sdram
csr
u_clk_rst
u_cpu
u_csr
u_debounce
u_dmem
u_fabric
u_uart
Dir Type Signals
Append
Insert
Replace
Signals
Time
bus_cpu {}
Waves
1400 ns
1500 ns
1600 ns
10001000
00000000
900DC0DE
Logfile viewer [1]
sim.log
Click-select on numbers to jump to that time value in the wave viewer.
VInit(0): initialising DPI-C interface
VProc version 1.11.4. Copyright (c) 2004-2024 Simon Southwell.
0 TOP.tb.error_mon (0) - ERROR_CLEARED
Wyvern Semiconductors
rv32 RISC-V ISS (on VProc)
Copyright (c) 2024
00000000: 0x00001197 auipc gp, 0x00000001
00000004: 0x0101a183 lw gp, 16(gp)
00000008: 0x0001a103 lw sp, 0(gp)
0000000c: 0x10001237 lui tp, 0x00010001
00000010: 0x00222023 sw sp, 0(tp)
00000014: 0x00022283 lw t0, 0(tp)
00000018: 0x00229663 bne t0, sp, 12
0000001c: 0x00004505' addi a0, zero, 1
0000001e: 0x00004501' addi a0, zero, 0
00000020: 0x05d00893 addi a7, zero, 93
00000024: 0x00009002' ebreak
Exited running ./models/rv32/riscvtest/main.bin
Register state:
zero = 0x00000000 ra = 0x00000000 sp = 0x900dc0de gp = 0x00001000
tp = 0x10001000 t0 = 0x900dc0de t1 = 0x00000000 t2 = 0x00000000
s0 = 0x00000000 s1 = 0x00000000 a0 = 0x00000000 a1 = 0x00000000
a2 = 0x00000000 a3 = 0x00000000 a4 = 0x00000000 a5 = 0x00000000
a6 = 0x00000000 a7 = 0x0000005d s2 = 0x00000000 s3 = 0x00000000
s4 = 0x00000000 s5 = 0x00000000 s6 = 0x00000000 s7 = 0x00000000
s8 = 0x00000000 s9 = 0x00000000 s10 = 0x00000000 s11 = 0x00000000
t3 = 0x00000000 t4 = 0x00000000 t5 = 0x00000000 t6 = 0x00000000
CSR state:
mstatus = 0x00003800
mie = 0x00000000
mvtec = 0x00000000
mscratch = 0x00000000
mepc = 0x00000000
mcause = 0x00000000
mtval = 0x00000000
mip = 0x00000000
mcycle = 0x0000000000000054
minstret = 0x000000000000000b
mtime = 0x000627450bfd784c
mtimecmp = 0xffffffffffffffff
- C:/git/wireguard-fpga/4.sim/tb.sv:44: Verilog $finish
Close Logfile

More Information

- Article on *VProc* and how it works
- General article on using simulation PLIs for co-simulation
  - References *VProc*, *Mem Model* (and the PCIe virtual host, *pcievhost*)
- The *VProc* and *Mem Model* documentation
- The *rv32* ISS documentation
- WireGuard Test Bench documentation section in README
- WireGuard *rv32* ISS integration code documentation

BACKUP SLIDES

Natively Compiled Application

- This method has been/is being used in commercial R&D Labs
  - Faster than ISS
- Not part of *VProc*, but an 'adapter' on top
- Hardware abstraction layer (HAL) used to virtualise away *VProc* Details
  - API to rest of the software the same
  - 'redirects' loads/stores to *VProc* API when running on *VProc*
  - Uses operator overloading to achive this effect or direct API calls, depending on HAL
- Places some small restrictions on code style
  - Best to code for this at the start if using this method
- Same code can then compile for platform, ISS or as native *VProc* code
- ***peakrdl*** c-header output has raised an issue
  - Uses bit fields in structures for fields of registers
  - No C++ operator overloading for accessing these fields
  - Never been an issue before as bit fields are not widely used for portability reasons
  - It does, however, generate the macros defining bit positions masks etc., though it doesn't use them itself.
- All the HALs encountered before use the macros to shift and mask in inline methods
  - Portable to all architectures
- Current plan is to use the standard peakrdl headers and augment with the classes that use its generated masks and bit positions
  - Ongoing. Nothing finalised

Comparing peakrdl with augmented proposal

- With direct use of *peakrdl* methods, access to registers looks something like:

```
#include "csr.h"

volatile csr_t *regs = (csr_t *)regbaseaddr;

int main (int argc, char **argv)
{
    printf("version %d.%d\n", regs->version.f.major,
                              regs->version.f.minor);

    regs->scratch.w = 0x900dc0de;
    printf("scratch = 0x%08X\n", regs->scratch.w);

    return 0;
}
```

- With proposed augmentation for VProc:
  - Used for both platform/ISS and VProc native

```
#include "csrvp.h"

csr_t *regs = new csr_t(regbaseaddr);

int main (int argc, char **argv)
{
    printf("version %d.%d\n", regs->version->major(),
                              regs->version->minor());

    regs->scratch->scratch(0x12345678);
    printf("scratch = 0x%08X\n", regs->scratch->scratch());

    return 0;
}
```

Example use of overloading for *VProc*

- Class called `reg_t` to define individual register
  - For platform builds, defined as volatile pointer to `uint32_t`
  - For *VProc* builds, defined as class reg_t shown
- Uses assignment overload for writes to call `VWrite()`
- Uses overload for `uint32_t` cast to call `VRead()`
- Now writes to, or reads from register, appear as transactions in the simulation
- These can be gathered into register blocks
  - Can use index overloading as well, if necessary

```
class reg_t
{
public:
    // Constructor
    reg_t(uint32_t a , unsigned n = 0) : addr(a), node(n)
    {
    }

    // Overload = for writes
    void operator=(uint32_t data) {
       VWrite(addr, data, 0, node);
    }

    // Conversion of uint32_t for reads
    operator uint32_t() const {
       uint32_t rdata;
       VRead(addr, &rdata, 0, node);

       return rdata;
    }

    uint32_t addr;
    unsigned node;
};

reg_t myReg(0x1000);

myReg          = 0x12345678; // This is where
uint32_t value = myReg;      // the magic happens
```