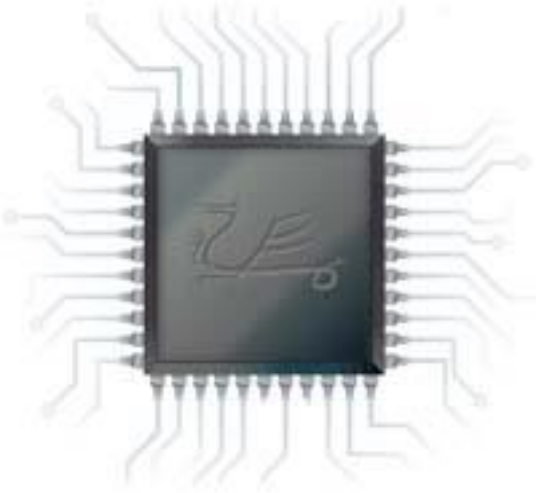
Extending the Power of Logic Simulations using the Programming Interfaces



Simon Southwell

May 2022

Preface

This document brings together three articles written in May 2022, and published on LinkedIn, that cover using the programming interfaces of SystemVerilog, Verilog and VHDL to write simple C functions called from logic in a simulation (part 1) and then take these concepts to build a virtual processor (part 2) and then co-simulate embedded software with logic using the virtual processor (part 3). In all of this only the initial simple concepts of part 1 are used, but real-world examples are given for how powerful these concepts can be in extending the ability and speed of logic simulation, and of development of digital IP and embedded software.

Simon Southwell
Cambridge, UK
May 2022

Contents

Part 1: The Programming Interfaces

Introduction

Some years ago I was working as software engineer implementing cycle-accurate models of SoC systems in C++ for a company making (amongst many other things) microcontrollers using their own proprietary RISC CPU core design. The CPU core and the microcontroller peripherals development went along in parallel and, of course, all needed verifying. The verification team was using third-party tools to drive the peripheral testing and, since the CPU was not available, reading and writing control and status registers on the system bus (or busses) was done with proprietary tools which could emulate a variety of available microprocessors, such as MIPs, ARM etc. This meant test code for driving the peripheral blocks had to be cross-compiled to a target CPU and then run on the simulation of that CPU that the third-party tool provided. One day the verification manager spoke to me saying that all he wanted for running the peripheral tests was to be able to write a program, compiled natively on the simulation host machine (a Windows PC or Linux server), that could read and write to memory locations on the bus, without all the need for the extra third-party tools and cross-compiling for a given processor. With him knowing that, at the time, I was hooking up the C++ models to the HDL simulator we were using, he asked me if this was possible. Circumstances then overtook things, with some company downsizing before this could be followed up at that particular company.

However, it got me thinking, as this was a good idea that was applicable to many situations where testing of logic required, at various points, control, and status registers to be accessed with reads and writes over the system's bus. In my own time I started playing with the Verilog PLI v1.0 supplied by the open-source Icarus Verilog simulation tool to achieve this. Since that time, to my surprise, most logic design engineers I've worked with have not used the programming interfaces provided by their simulation tools and many were not even aware of them. Even more shocking is that some did not even know how to write basic software in C or C++. I have spent most of my career as a logic designer, though I have been a software engineer as well—but I knew how to write C and C++ before that time. If you are sitting every working day in front of a computer, how can you not want to know how to program it? If you are a logic designer reading this and are not yet able to program in C or C++ (or even Python), get your company to send you on a course. Not only will it open up your abilities in logic development and test, make you more aware of embedded software needs, but will, more importantly, allow you to follow the rest of this article—as you will need some C knowledge.

In Verilog these APIs are standard, with PLI 1.0 (programming logic interface) and VPI (Verilog Procedural Interface—PLI 2.0). VHDL has standardised with VHPI (VHDL Procedural Interface) and Mentor Graphic's ModelSim FLI (Foreign Language Interface) has been available in VHDL for many years. With the advent of SystemVerilog, interfacing to a program from the HDL world is even easier using the DPI (Direct Programming Interface). These all do pretty much the same thing, with the differences just in the details. They allow a program function (written in C, say) to be called from logic and (if required) the other way around as well. In addition, a whole host of other things can be done, but essentially there is a means to exchange information across a logic-software boundary and process it in the program code.

To get to the point where a program, in C or C++, can be run that can read and write registers on a simulated system bus, a few things must be solved. In these articles we will first review some programming interfaces and how they can be used, using the absolute bare minimum of features, to call a function and return data with the goal of demonstrating that it doesn't really matter which flavour of PLI is used once one knows how to set it up. This means it is then straightforward to use the appropriate tools for the environment you are working in. From there we will explore how to cross the logic-software boundary and make simple functions calls to use software to do various tasks. From this point we will be able to discuss how to solve the problem of running a program to access a system bus on the HDL simulator for register reads and writes. This will not require any new PLI features to do this and once we have crossed the software-logic chasm the sky's the limit. Once we have the ability to access the system bus from software, just like a CPU, we will then look at how we might go even further and compile some or all of the target embedded software to be the 'software' accessing the system bus, and thus have co-simulation capability. All this using the simplest part of the PLI features to cross the divide and then build layers on top of this. The power of this is not in the PLIs themselves (good though they are) but, having crossed the boundary, what then becomes possible.

For the rest of this first article we will just review the programming logic interfaces mentioned above, except for VHPI as I don't have any real experience with it and does not appear to have as much support for it in simulators I've used. Hopefully, the comparison of the other interfaces will show that they all pretty much do the same thing, so if you need to use the VHPI interface you will be able to adapt what is here.

Motivation

It is true that all of the things I have outlined do not *need* to be done to perform the testing required, or even the co-simulation. So why bother? Well, simulation cycles are costly in processing power and take CPU time and are thus relatively slow. It would be better to use those simulation cycles, as much as possible, to exercise the unit under test and not waste them on the test bench processing. Once we have crossed over to a software program, the speed of processing data will increase by orders of magnitude. Indeed, in my deployment of these methods in logic test benches, I have always tried to get data into software with as little processing in the logic simulation as possible. Once there, one has access to all the vast number of software libraries available to help do the processing of data and cross-checking for validity, generation of statistics, logging of data etc. Use the simulator processing cycles wisely.

All these methods have been tried and tested in real commercial product development, from the development of Supercomputers—co-simulating the fabric kernel driver code, running in QEMU, with ASIC logic—to testing the embedded logic for industrial inkjet printer control systems. In these articles references will be made to example open-source code that does all these things and thus can act as reference designs for those who would dive deeper into this subject, or as useful tools for your own development projects.

SystemVerilog's DPI

The easiest of the programming interfaces to use is the SystemVerilog direct programming interface (DPI). More strictly we will use the DPI-C interface as we are interfacing to C code. We need to tell the test bench logic about the C function we want use and be able to call it from the SystemVerilog code. First, we need some C functions to call. Simple memory read and write functions are what we need.

```
void Write  (const int  addr, const int  data, const int be)
{
    /* do some processing */
}

void Read   (const int  addr,       int* data, const int be)
{
    /* do some processing */
}
```

This C functions are just what you might have in a normal C program—as part of an instruction set simulator model, say. They can even be compiled as C++, adding extern "c" before them as the interface links with C functions. We then have access to a whole C++ environment. What code goes in these functions isn't defined here,

but could, say, access a large array to read and write to, for a simple memory model. (We can do better than that, and more on this later.) These functions need to be linked to the System Verilog somehow.

In the SystemVerilog code from which we will call the C functions we can import them, within the module, as shown below:

```
import "DPI-C" function void Write(input int addr, input int  data, input int be);

import "DPI-C" function void Read (input int addr, output int data, input int be);
```

You'll notice that the declaration looks quite similar to the C prototypes, but a direction has been added as input or output, with output matching the pointer integer argument of the Read() function. Now there is a whole lot of different types that can be matched between SystemVerilog types and C types, but we will stick to integers, keeping the minimalist philosophy intact. Also one can go in the other direction and map SystemVerilog tasks and functions to be callable functions from C—but we'll let that go as well for now.

Now, from within the SystemVerilog module the functions can be called, much the same as a SystemVerilog task:

```
always @(posedge clk)
begin
  // some code...

  // Do a write
  if (write)
    Write(waddr, wr_data, wr_byte_en);

  // Do a read
  if (read)
    Read(raddr, rd_data, 'h15);

  // ...some more code
end
```

The above code implies that the arguments are all SystemVerilog integer types, but they could be 32-bit reg or input vector types etc., and SystemVerilog will cast them to integers as that is how the function is declared. One thing to note, though, is that if the vector has a `'Z'` or `'X'` on any of the bits, the whole value is cast to 0.

Compiling the code

We have C functions and we have calls from a SystemVerilog module, but we need to compile all this into the simulation. This can vary from simulator to simulator. I have been using ModelSim, as it covers all the languages and PLIs I want to discuss, but whatever simulator you have available will not be too different. ModelSim

allows the C source code to be compiled as part of the file list and will recognise the `.c` suffix. For example:

```
vlib work
vlog my_sv_code.sv my_c_funcs.c
```

This is fine for small file lists, but a more generic method is to compile all the code into a shared object (or dynamic link library on Windows) with whatever C compiler is suitable for the host machine. For ModelSim compiling this to a 32 bit or 64 bit so/dll is required, depending on the version of ModelSim. For example:

```
gcc -shared -Bsymbolic -m32 -I $MODEL_TECH/../include -I. my_c_funcs.c \
    -o my_c_funcs.so
```

Note that, in windows, ModelSim supports MinGW and (I believe) Visual Studio. It doesn't support Cygwin, though, and one must be careful not to use Cygwin compiled code. Under Linux none of this is an issue. Now, with the shared object (or DLL) we can load it as a SystemVerilog library.

Running the Simulation

To run the simulation with the shared object loaded (assuming some test environment with `top_module` as the top-level module) something like the following is run on ModelSim

```
vsim -sv_lib my_c_funcs top_module
```

The `vsim` program knows to look for a `.so` (or `.dll`) file with the name given as the argument of the `-sv_lib` command line option. By going down the shared object route this can be compiled with all the C or C++ functionality required, including the DPI functions that use this other code. More than one shared object can be specified, so if it make sense to compile libraries and other programs separately, this is supported, so long as it's all there on the vsim command line.

And that's all there is to it. Having done this, for SystemVerilog, lets now look at Verilog's PLI to do the same thing. We'll be able to skip some of the detail in this initial section as it is basically the same, such as compiling the C.

Verilog PLI 1.0

The PLI task/function interface (tf) is probably the most widely supported programming logic interface, and so is useful to know. However it is more awkward to use than the DPI interface and is now deprecated (but not yet obsoleted)! It is replaced by the VPI interface (PLI 2.0), but it is so ubiquitous that it may be all that

is available to you, particularly in open-source simulators such as Icarus and Verilator (on both of which I have used this interface). Let's define the C function prototypes for the same Read and Write as before.

```
int Write (void);

int Read (void);
```

The main thing here to note is that the C functions do not have any parameters, and we will have to use provided PLI library functions to access the passed in calling arguments, and to pass back values. To read input values from the task call in Verilog, the `tf_getp(<argument index>)` function is used. To put a value on an output of the calling task, the `tf_putp(<argument index>)` is used. Both tasks take an integer index which selects which of the arguments is being got or put, with the first argument at 1 (argument 0 is the function/task itself). So the functions can now look like the following:

```
int Write (void)
{
  int addr, data, be;

  // get inputs values
  addr = tf_getp(TF_ADDR_IDX); // first argument
  data = tf_getp(TF_DATA_IDX); // second argument
  be   = tf_getp(TF_BE_IDX);   // third argument

  /* do some processing */

  // Return OK status
  return 0;
}

int Read (void)
{
  int addr, data, be;

  // get input values
  addr = tf_getp(TF_ADDR_IDX);
  be   = tf_getp(TF_BE_IDX);

  /* do some processing... */

  // Set output value(s)
  tf_putp(TF_DATA_IDX, data);

  // Return OK status
  return 0;
}
```

So, from the C point of view we just have to make a few extra PLI calls at the beginning and, where output is to be set, calls at the end. Other than that the code

can be the same as that for SystemVerilog and DPI. However, to let the simulator know about the functions, there is a little bit more work to than for DPI.

For PLI 1.0 a table must be provided mapping the C functions to Verilog tasks. This varies from simulator to simulator as well and (if memory serves me correctly) it is done differently in, say, VCS than for ModelSim, and Icarus needed some other function to register the table. In ModelSim the table, and a function to return a pointer to it, looks like the following:

```
#include <veriuser.h>
#include <vpi_user.h>

s_tfcell veriusertfs[] =
{
    {usertask, 0, NULL, 0, Read,   NULL,  "$tfread",   1},
    {usertask, 0, NULL, 0, Write,  NULL,  "$tfwrite",  1},
    {0}
};

p_tfcell bootstrap ()
{
    return veriusertfs;
}
```

Without going into too much detail, the table is an array of structures, terminated by a null entry (the `{0}`). There is one entry for each C function we wish to use, with the first field indicating that this is a user defined task and will thus be called like any other Verilog built-in task (such as `$display`, `$memreadh` etc.). The name of the task, to be called from Verilog, is defined in the string of the seventh field. I changed them slightly from the C names, to avoid clashes with existing system task names. The other fields don't matter for what we are doing and can be just set as shown. Refer to you documentation for more details. The `bootstrap()` function needs to be defined, with that name, and return the pointer to the table.

This code can be part of the C function source code or, perhaps more properly, in a separate source file. With all this compiled into a shared object, just as for SystemVerilog, the C functions can be called from Verilog, via the defined task names:

```
always @(posedge clk)
begin
  // some code...

  // Do a write
  if (write)
    $tfwrite(waddr, wr_data, wr_byte_en);

  // Do a read
  if (read)
    $tfread(raddr, rd_data, 'h15);
```

```
  // ...some more code
end
```

This all now acts like the SystemVerilog, with integers being input and output, and the same vector to integer casting rules.

Compiling the code

With the additional C code for the task table, the C code is compiled into a shared object (or DLL) in just the same way as for SystemVerilog. Let's assume that the additional PLI C source is in our `my_c_funcs.c` source file, then it would be almost the same as SystemVerilog, but an extra library (mtipli) is required:

```
gcc -shared -Bsymbolic -m32 -I $MODEL_TECH/../include -I. my_c_funcs.c \
    -L$MODEL_TECH -lmtipli                                              \
    -o my_c_funcs.so
```

Running the Simulation

Running the simulation with PLI code is similar to SystemVerilog, but the shared object is referenced slightly differently

```
vsim -pli my_c_funcs.so top_module
```

Here, the `-pli` command line argument replaces `-sv_lib` and needs the whole name of the shared object file (and doesn't imply the suffix). Other than that it is the same.

As I said before, though not yet obsoleted, the PLI 1.0 interface is being deprecated in favour of VPI (i.e. PLI 2.0). So let's have a look at this.

Verilog VPI

This interface is a much more flexible and consistent interface than PLI 1.0, but for our purposes it is slightly different and a little more complicated. For the C function prototypes we now have:

```
int Write (char *userdata);
int Read  (char *userdata);
```

The introduction of the char pointer argument, over PLI 1.0, is of no consequence, and we won't be using it. To get access to the arguments of the calling task we need a 'handle' to the argument list, then an 'iterator' to go through each argument in turn. As we are likely to do this a lot, lets define a function to do this:

```
#include "veriuser.h"
#include "vpi_user.h"

int getArgs (vpiHandle taskHdl, int value[])
{
  int                   idx = 0;
  struct t_vpi_value    argval;
  vpiHandle             argh;

  vpiHandle             args_iter = vpi_iterate(vpiArgument, taskHdl);

  while (argh = vpi_scan(args_iter))
  {
    argval.format       = vpiIntVal;

    vpi_get_value(argh, &argval);
    value[idx]          = argval.value.integer;

  }

  return idx;
}
```

Our function takes, as input, a handle (of type `vpiHandle`) to the calling task and a pointer to an array of integers in which to put the retrieved argument values. Within the function we need a variable of type `t_vpi_value` and another `vpiHandle` for the argument values. Finally we need our iterator object (type `vpiHandle` again) and initialise it with a call to `vpi_iterate()`, passing in a `vpiArgument` type to say we want an argument iterator, and the handle of the calling task. Now, using this iterator, we can get each argument (of type `t_vpi_value`) in a loop until it returns null. At each loop, we get a handle to the argument's object. The `argval` structure has its format field set for an integer type, to indicate that the argument is of that type, and then `vpi_get_value()` is called, passing in the argument handle and the argval structure. The actual argument value is returned in `argval.value.integer`, which we can place into the passed in array.

Updating arguments is not dissimilar, so let's define a function to do this.;

```
int updateArgs (vpiHandle taskHdl, int value[])
{
  int                   idx = 0;
  struct t_vpi_value    argval;
  vpiHandle             argh;

  vpiHandle             args_iter = vpi_iterate(vpiArgument, taskHdl);

  while (argh = vpi_scan(args_iter))
  {
    argval.format        = vpiIntVal;
    argval.value.integer = value[idx++];

    vpi_put_value(argh, &argval, NULL, vpiNoDelay);
  }
```

```
  return idx;
}
```

The general form is the same, but now the value array contains the update data, and `argval.value.integer` is set with the value before calling `vpi_put_value()`. This has a couple of extra parameters, which can be set as shown to emulate the previous code we have looked at. With these two functions defined, our Read and Write templates become:

```
int Write (void)
{
  int addr, data, be;
  vpiHandle taskHdl;
  int argVals[4];

  taskHdl = vpi_handle(vpiSysTfCall, NULL);
  getArgs(taskHdl, &argVals[1]);

  // get inputs values
  addr = argVals[TF_ADDR_IDX];
  data = argVals[TF_DATA_IDX];
  be   = argVals[TF_BE_IDX];

  /* do some processing... */

  // Return OK status
  return 0;
}

int Read (void)
{
  int addr, data, be;
  vpiHandle taskHdl;
  int argVals[4];

  taskHdl = vpi_handle(vpiSysTfCall, NULL);
  getArgs(taskHdl, &argVals[1])

  // get input values
  addr = argVals(TF_ADDR_IDX);
  be   = argVals(TF_BE_IDX);

  /* do some processing... */

  // Set output value(s)
  argVals[TF_DATA_IDX] =  data;
  updateArgs(taskHdl, &argVals[1]);

  // Return OK status
  return 0;
}
```

So a couple of things to note here. Firstly, the `taskHdl` handle is created externally to the two functions, mainly so it can be reused in the `Read()` function for input and output with two calls to `vpi_handle`. Secondly, the array to receive the values has a

dimension of four, and then the pointer to the array index 1 is passed to the functions. This is to make it look more like the PLI 1.0 code, and the defined indexes for the arguments will than work for both the PLI and VPI code. This allows code to be more easily be written and compiled for either version of Verilog PLI.

The VPI tasks need to be registered, similarly to PLI 1.0, only in a different way.

```
void register_vpi_tasks()
{
    s_vpi_systf_data data[] =
      {{vpiSysTask, 0, "$tfread",    Read,     0, 0, 0},
       {vpiSysTask, 0, "$tfwrite",   Write,    0, 0, 0},
      };

    for (int idx= 0; idx < 2; idx++)
    {
        vpi_register_systf(&data[idx]);
    }
}

void (*vlog_startup_routines[])() =
{
    register_vpi_tasks,
    0
}
```

We create a function (`register_vpi_tasks()` in this example), and in it we create an array of `s_vpi_systf_data` structures, and initialise them as shown, indicating that these are tasks in the first field, the name of the user task in Verilog in the third field, and the C function name in the next field. All other fields are set to zero. We register these entries by looping across the array and calling `vpi_register_systf()` for each one. Now we must add the pointer to the `register_vpi_tasks()` function to an array of pointers-to-functions and terminate the list with 0. The type of each pointer-to-function entry is just as for `register_vpi_tasks()`, and the array pointer is already defined in the VPI libraries provided by the simulator.

Compiling the VPI C code, calling the C functions from Verilog, and running the simulation are *identical* to the PLI 1.0 case, so I won't repeat it here (refer to that section above).

Now, you might be thinking that this is all a bit of a faff to do the same things as we've already done in a much simpler way. Well, you'll get no argument from me and it maybe that, once PLI 1.0 is obsoleted and if you don't have access to SystemVerilog's DPI, it's the only choice available to you. But now you know how to do it, wrapping up the esoteric details into two functions that we've written, so we don't have to worry about it too much again. That wraps it up for the Verilog type

languages, what about for those developing in VHDL. Let's look at the ModelSim FLI as the last programming interface.

VHDL FLI

The foreign language interface from ModelSim basically does for VHDL what PLI does for Verilog. The C functions are defined exactly the same as for the SystemVerilog, with parameters for each of the input and output signals of the calling task—the outputs being pointers. To gain access to them in from VHDL some procedures need to be defined, in a package just as one would for normal procedures, but with dummy bodies, and a special attribute.

```
package my_pkg is

  procedure Write (
    addr          : in  integer;
    data          : in  integer;
    be            : in  integer;
  );
  attribute foreign of Write : procedure is "Write my_c_funcs.so";

  procedure Read (
    addr          : in  integer;
    data          : out integer;
    be            : in  integer;
  );
  attribute foreign of Read: procedure is "Read my_c_funcs.so";
end;

package body my_pkg is

  procedure Write (
    addr          : in  integer;
    data          : in  integer;
    be            : in  integer;
  ) is
  begin
    report "ERROR: foreign subprogram Write not called";
  end;

  procedure Read (
    addr          : in  integer;
    data          : out integer;
    be            : in  integer;
  ) is
  begin
    report "ERROR: foreign subprogram Read not called";
  end;
end;
```

Note that the procedures in the package still need to have bodies, but dummy code is put there. If the foreign code is missing the error messages will be displayed instead. The attributes map the C functions to the VHDL procedures. Note the last

string also defines the shared object file in which the C functions are located. So long as this package is compiled into the work library, then the C functions can be called from VHDL code:

```
use work.my_pkg.all;

entity my_fli_block is
  port (
         # port list #
       );
end entity my_fli_block;

architecture behaviour of my_fli_block is

  process(clk)
  begin
    if clk'event and clk = '1' then
      # some code...

      # Do a write
      if write = '1' then
        Write(waddr, wr_data, wr_byte_en);
      end if;

      # Do a read
      if read = '1' then
        Read(raddr, rd_data, 'h15);
      end if

      # ...some more code
    end if;
  end process;
```

Compiling the code

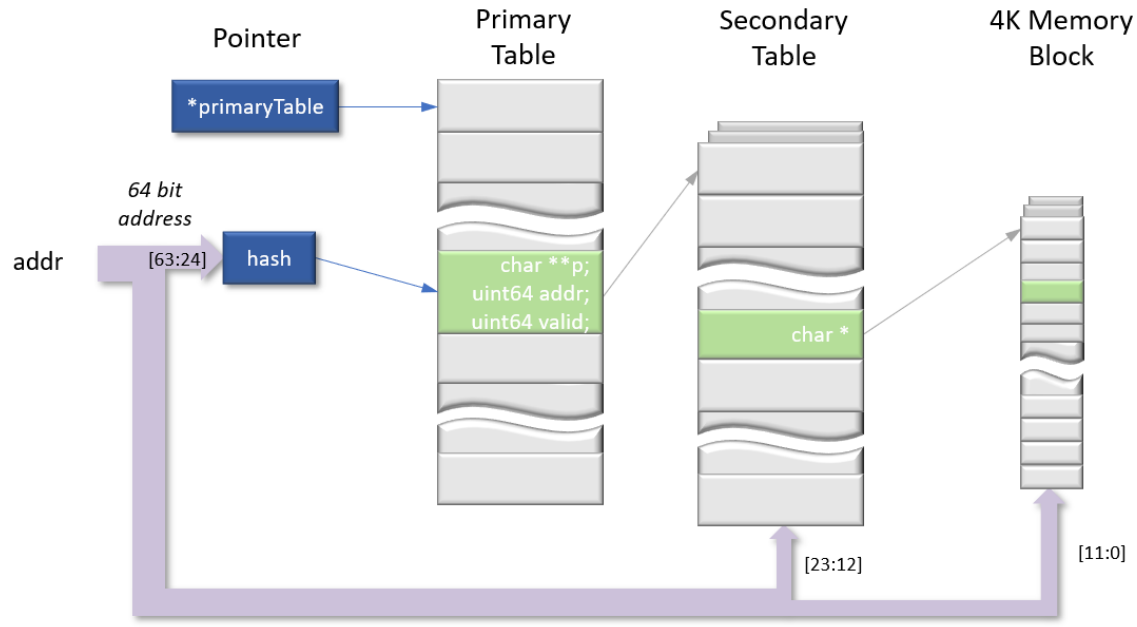
The C is compiled for ModelSim into a shared object file, exactly as for the SystemVerilog DPI interface. Compiling the VHDL is just as for normal VHDL simulations, but also including the package source file. When running the simulation the `-pli` command line argument is used once more in to load the shared object.

Real World Use Case

The above discussion has shown how to call simple functions in C from SystemVerilog, Verilog and VHDL. A couple of outline functions were called, with integer arguments to do a read and a write to some undefined coded function to illustrate the principles of bridging the chasm. So what can *really* be done with this?

Well, as a real-world example I want to outline an open-source memory model that I wrote using these techniques. The problem I was trying to solve was that the

simulation components being tested had access to a very large address space (gigabytes in fact), and might make accesses at very disparate places in that space. Obviously, I couldn't create an array of logic bit vectors in the gigabytes capacity. The simulation, though accessing over a very wide space, won't actually access the entire space (that would take forever). It happened that I'd already written a C++ model of a memory that could address an entire 64-bit address space, by allocating memory blocks as and when they were accessed. The general principle is shown in the diagram from the model's documentation, using a 3-stage approach and 4Kbyte blocks:



The API for this C++ model has read and write functions for various word sizes, from bytes to DWORDS, and so it was easy to interface to the simple read and write functions of the PLI C interface of the type we have been discussing (the names are slightly different). Some simple logic around calls to the read and write PLI tasks in the Verilog, with some bursting and SRAM type interfaces, I now have a Verilog memory test component with a very large address space, which runs orders of magnitude faster than a Verilog model and consumes little memory resources. More details of this model can be found in the documentation in the github repository.

Conclusions

We've crossed the logic/software divide and compared techniques for doing so. The article has deliberately restricted to the bare essentials of what is actually available in these interfaces but, having made the leap, we have seen that this opens up a

whole opportunity of adding large, possibly complex, functionality in the C/C++ domain using all that's available there, just as in the memory model example.

Which interface to use? I don't think it really matters—which ever are available. Though some are more complex to use than others, once that initial setup is done, they are largely the same. With many simulators being mixed language, you may have access to SystemVerilog, VHDL and Verilog all in the same environment, so you can use whichever programming logic interface is appropriate. Just bare in mind portability. I have usually avoided mixing languages where possible to maximise reuse.

So, have we met the brief regarding the requirement to read and write to memory addresses from a natively compiled program, as requested by my verification manager all those years ago? Well, no. The examples we have looked at are all what I call 'passive' calls. That is the C is called from the HDL and then returns. This is still extremely useful, but not the full Monty. We can't yet write a standalone program where the C/C++ is running, making read and write calls to the simulation. We need a means to make the C/C++ program look like it is running the show when the simulator is in fact the main program. Not easy, and there are a couple of approaches I've used in my professional career, one of which is easier to understand and keeps the passive call approach, and one of which I have constructed in such a way as to allow co-simulation features and truly has a single C/C++ program running, which unlocks more potential from bridging over a PLI. But this will have to wait for the next part.

Part 2: Virtual Processor

Introduction

In part 1 on the programming logic interfaces, we had a look at some of the different interfaces for the three main HDL languages. If you haven't had a look at that article then I suggest you have a look before tackling this one, as we will be using what was discussed there. In that last article I started with a requirement to be able to read and write, over a memory mapped bus, to memory and IP control and status registers using a program (in C or C++) that is natively compiled for the host machine. We got as far as crossing the software-logic gap with two simple read and write C functions that can be called from the HDL and we saw an example of how these simple concepts might be used to create a memory model which can handle full multi-gigabyte address spaces. But we still haven't met the brief since these functions are a 'passive' call-and-return from the HDL. This means we don't have a program flow from a C/C++ program that can make function calls to write and read to registers or memory over a bus. In this article we will tackle that problem by without using any more sophisticated PLI concepts than from the previous article.

In this article I want to outline two methods I have used in my professional life. The first is, perhaps, conceptually easier to understand and has a certain flexibility to, as we can use other languages such as Python. It also maintains the passive call-and-return concept. The second method I want to outline is a little more involved but is much more closely tied to the simulator and runs much, much faster than the first method. It does involve the use of threads in the C/C++ code, which bring their own hazards, but the design deliberately negates these with tight coupling in synchronisation between C/C++ and the simulator's PLI. If you are a logic designer thinking that you don't have much experience with threaded code, don't worry, because this has already been done for you , and we will be referencing, as a use case, a 'Virtual Processor' that uses just these concepts.

The problem that needs solving here is, as an ex-colleague of mine succinctly put it, the simulator program is the master, but the software looks like it's the master. So how can we solve this?

Using a Separate Process

The originally stated requirement was to have a natively compiled program that can read and write to register over a bus in a simulation. So, if we write a program, with some function calls that 'somehow' do these accesses, we can compile this to a

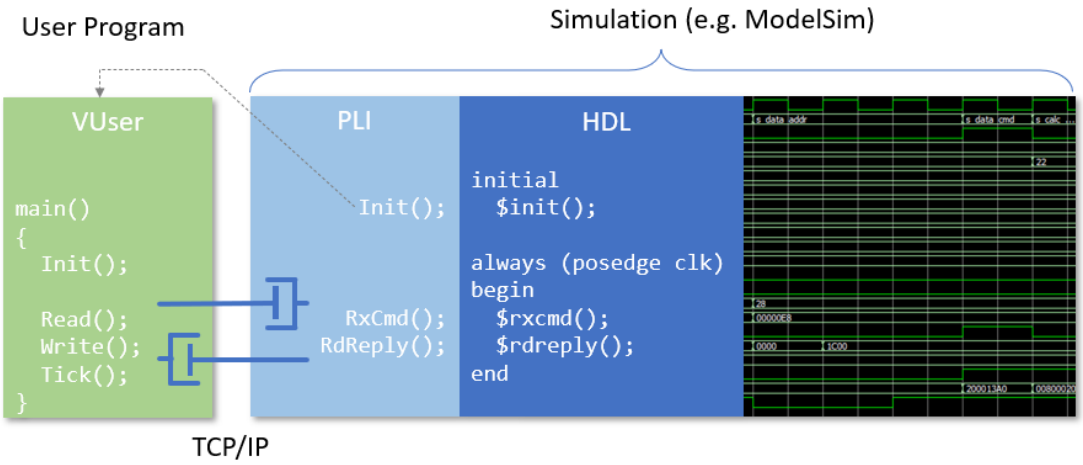
standalone executable program and run it. This seems to move the problem, rather than solve it, as that 'somehow' needs to be solved.

In one company I worked, that problem as solved with the use of TCP sockets. I did not construct this environment, so do not have access to the source code, but the concepts are straight forward enough. Sockets are available on Windows or Linux but are (in C/C++) handled slightly differently. For example of opening and communicating with sockets in C++ take a look at the GDB code (`rv32_cpu_gdb.cpp`) of the rv32_cpu project's instruction set simulator. This is used to connect a gdb debugger over a socketed remote connection to allow debug of code running on the ISS. So, if our compiled program can now send and receive data over a socket, we can connect it to code accessible to the HDL over PLI. The functions for the PLI can be connected in just the same way as we did in part 1.

Now, we can't just have the separate program and the simulator free-running without synchronising these two processes. This is solved by the simple protocol chosen. An initialisation PLI function (e.g. `$init`) is required which will create the socket to receive data from the program, with an address and port number. This initialisation function could also run the separate program as well or could just wait for a connection when the program is run manually. The program would also have a socket created and try and connect to the IP address and port to establish the link. Once connected communications and synchronisation can take place.

To synchronise two programs, the protocol is that the HDL running on the simulator will make a call to a PLI task (e.g. `$rxcmd`) which will than wait for a communication from the external program. At this point the simulator is halted and not advancing time. The program, when it reaches a point to read or write to memory, calls its read or write function, which will have code to send a packet over the connection, it will then wait for a reply. The details of the packet are not important but would have information encoding a command (e.g. read, write, idle), an address, data (if a write) and possibly byte enables. When the PLI code receives this packet (if valid), it returns from the call, updating the output arguments of the HDL task, so that an access over the bus can take place using this information, taking as many cycles as the bus protocol requires. If the access was a write, then a reply would be returned to the external program immediately. If a read, then no reply is returned at that point. Instead, the bus access is done first, retrieving the read data and a call to a separate PLI task (e.g. `$rdreply`) is made, passing in the read data, and this PLI C function generate the reply, including the read data. With this command-reply protocol, the flow of the two programs are interlocked. As it described, the external program will issue reads and write at a maximum rate, as any code running that isn't an access effectively runs in zero simulation time. This is why I gave an example of an idle command. This command would be interpreted by

the HDL behavioural code as to wait so many cycles before calling $command again. This allows the external program to specify any amount of time to pass in the simulation before it will do a read or a write again. The protocol might also have an exit command to terminate the simulation.



The advantage of this method is that the construction of the program is independent of the PLIs and can be compiled as a standalone executable. Also, the PLI code is the simple call-and-return method we have already looked at in part 1. Another advantage is that the language of this external program can be anything that supports connecting to a TCP/IP socket. This could be Python, as was used at my former company, or any other convenient language of your choosing. Does this method meet the brief of having a natively compiled program that can read and write registers over a simulated bus? Yes it does, and I suspect my verification manager would have been more than happy with this solution. So, do we need to take a look at another method?

Well, the disadvantage of this first methods is that it is heavy weight. We have to open a TCP/IP socket and for every read or write to the bus we have to communicate over this link, with all the layers of software between the external program and the simulator PLI code. When I started looking at solutions to this (before I worked at the company using this first method), I was in the high-performance computing (HPC) industry, and I wanted something very fast and very light-weight. Having bridged the HDL/C gap, we are already in the C/C++ domain, so it is just a matter of extending this code without the need for any other program. Let's now look at this method.

Using Threads

We still have to solve the same 'two master' problem as before and since we don't want to have an external program, the choice left to us is the use of threads. Instead

of running an external program we create a new thread instead. This, like the external program, is free running with respect to the simulation, and so we need to couple these two threads like we did for the two programs of the first method. In the first methods we blocked on the sockets, waiting for a command (or a reply in the external program). In this method we can use semaphores to block and communicate basically the same type of information via some shared memory. The semaphores are there to ensure that the receiving end does not access the shared memory whilst it is being updated and so both threads are only active one at a time. This makes the code thread-safe.

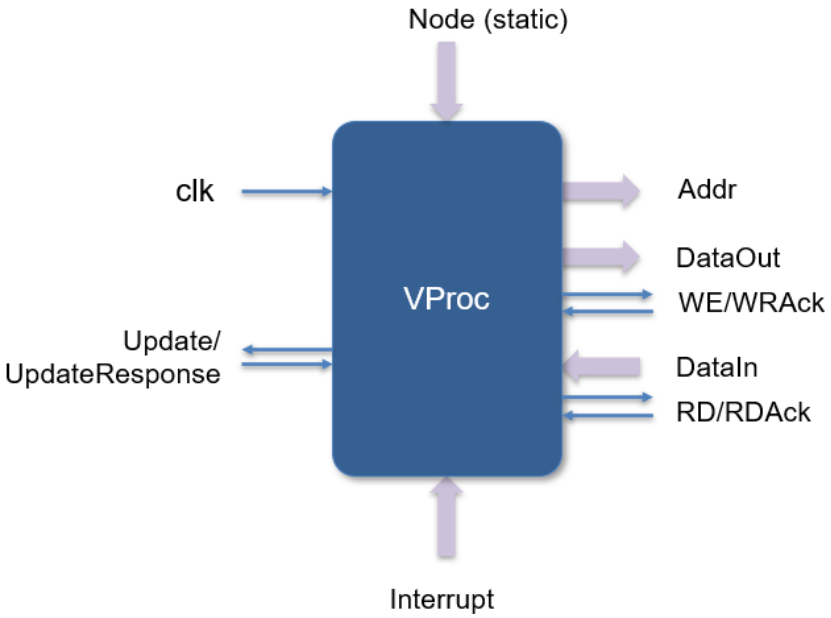
Again, there will need to be an initialisation task to set things up and create the new user thread (e.g. `$vinit`), and a regularly scheduled task to get the commands (e.g. `$vsched`). When a read or write (or even idle) command is sent, the user thread will set a semaphore to allow the PLI code to proceed and receive the message, it will then wait on a response semaphore controlled by the PLI code. The PLI code, once unblocked when the user semaphore is set, will get the command, and pass back the data to the calling task's outputs. The HDL can then do the access of the bus. If a read it can get the returned data and pass this back into to the next call to `$vsched` for pick up by the user thread once the response semaphore is set once more. And the cycle repeats. Now we have a means to write a program that runs, making calls to a read or write function to instigate reads and write over the simulated bus. So now this method meets our brief.

This acts in the same way as the first method, just using threads and semaphores to achieve the same goal. Now, you might be thinking that using threads and semaphores is somewhat complicated and, as a logic designer or verification engineer, it's a little outside of my comfort zone. The advantage of this added complexity, though, is that the user code is now simply compiled in with the PLI code to a shared object (see part 1) and it all runs as part of the simulation. This means that it runs very fast. The only layers we have between our code and the simulator are the semaphores, and the fact that the OS is swapping contexts between the user thread and the simulator (it does this between the user and simulator processes of the first method anyway). No calls over TCP/IP. Also, this has already been done for you in the open-source VProc virtual processor project. Let's take a look at this as a case study.

Real World Use Case—A Virtual Processor

The open-source VProc virtual processor project uses the methods described in method 2 above, to implement a Verilog (or VHDL) behavioural component that, externally, has a very simple 32-bit bus interface, with address, a data out with a write enable and acknowledge, and a data input with a read and read acknowledge.

In addition, it supports some interrupt functionality. An update and update-response pair of signals also allow for delta-cycle updates if required but tied together if not (more on this later).



More than one virtual processor can be instantiated, and so a node input port is provided to give each a unique node ID.

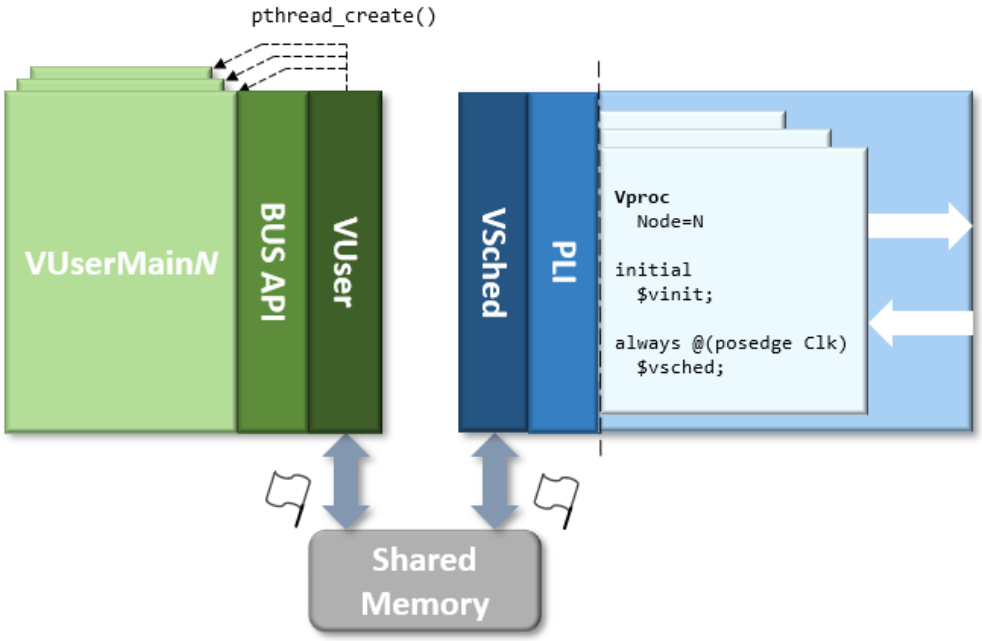
VProc supports both VHDL (via FLI) and Verilog (via PLI or VPI). This component is meant to be a generic processing element component and would normally be wrapped inside some behavioural logic to be compliant with some other standard protocol (e.g. Intel/Altera's Avalon bus or AMBA AHB). This combination would then be a bus functional model. We are not going to go over the VProc source code line-by-line but will have an overview as a case study. More detail can be found in the VProc documentation.

From the perspective of the PLI tasks that are called in the VProc component, these are (using Verilog) `$vinit` and `$vsched`, just as mentioned for the second method. In VHDL these are `VInit` and `VSched` procedures. Skating over the thread initialisation and semaphores, from a user code perspective, for each node instantiated (with a unique node number), separate thread and semaphores are created. The user code for each node has a top-level entry function `VUserMain`*x*, where *x* is the node number (e.g. `void VUserMain0(void)`). This is analogous to `main()` or, perhaps more analogously to `WinMain()` for windows graphical applications (which is where I got the idea from).

From this top-level function, any C or C++ code can be written (the `VUserMain`*x* functions must have C linkage, though). The user code then has access to some simple API functions:

```
int VWrite (unsigned int addr,  unsigned int  data, int delta, unsigned int node);
int VRead  (unsigned int addr,  unsigned int *data, int delta, unsigned int node);
int VTick  (unsigned int ticks, unsigned int node);
```

These functions implement the write, read and idle functionality we have been describing. The diagram below, from the VProc documentation, gives an overview of the layers from HDL to user code.



An addition to the fundamental description of the functionality is the delta argument to the bus API functions, and we will deal with delta-cycle updates in a later section.

Interrupts

The VProc HDL supports rudimentary interrupt support it has a 3-bit interrupt input port, giving eight different interrupt states, with state 0 being no interrupt. If this port changes to a non-zero state, then, in that cycle, a user function can be called if it was registered. An API function is defined:

```
void VRegInterrupt (int level, pVUserInt_t func, unsigned node);
```

This can be used to register a C function (of type `int func(void)`) which is called for each cycle that the interrupt input port is equal to that level. This is not a true

interrupt, in that it does not interrupt the user code at wherever it is (in fact is only detected when in a read, write or tick function), but simply makes a call to a function. However, when in the user registered function, the main code will be stalled on a semaphore, and it is perfectly safe to write to any shared memory set up. An event loop can then be emulated by wrapping up the calls to VWrite, VRead and VTick, checking if any message has been posted by the interrupt function(s), and executing the 'handler' code before the call to the user API functions. Since the main code has not yet called its access to the API, the handler code is free to do API calls itself.

None of this needs advanced thread safe techniques. The nature of the synchronisation between user thread and simulator means only one thing is ever running at a time and API calls are naturally atomic. A conflict would only occur if the user code itself was multi-threaded. In this case the API to the PLI can be wrapped up in a higher-level API as the single access point to VProc API and, say, mutexes (or semaphores) used to ensure atomic access. All then would be back in harmony and thread safe, as far as the VProc API is concerned. Between the user threads, normal threading precautions must still be maintained.

There is also a `VRegUser()` method to register another user function to be called if the `$procurer` task is called. This takes a single integer input. It is not used in the current HDL VProc code and is provided to allow user extension.

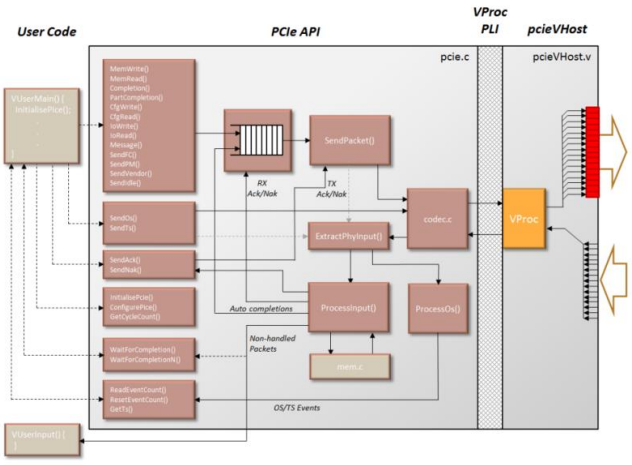
Delta Cycle Updates

As it stands, VProc can read and write 32-bit data in a 32-bit address space. This makes it look like a generic (virtual) 32-bit processor. Wrapping it up in a BFM of a 32-bit protocol is straight forward. What if we wanted more than that? A 64-bit protocol, or some other protocol such as PCIe or TCP/IPv4 that have different signalling requirements? The delta cycle update and response signals allow for this, with the requirement that extra code needs to be written to support the protocol.

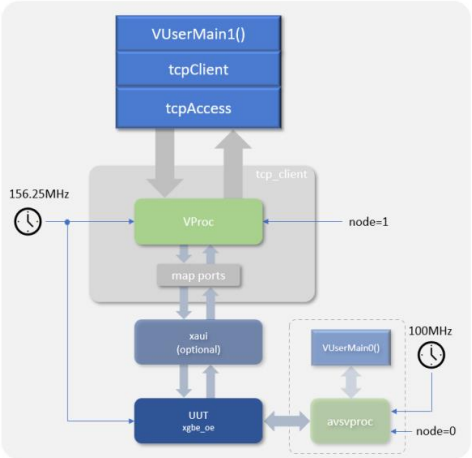
In the discussion above of VProc it is assumed that it is modelling a processing element, with address and data. We have interpreted these signals to be that of a memory mapped bus, but they needn't be. Taking the 64-bit example, for instance, it may be that we define a wrapping BFM with registered 64-bit address and data outputs, and 64-bit data input. Now, we can add a layer of indirection. The address from the VProc component now just indexes the 32-bit chunks of these ports (the lower and upper halves of the address or data) or indexes individual bits, such as a write enable. So, for example, the lower address 32-bits can be at VProc address 0, and the upper at address 1 and so on. We can write to these in turn to set the outputs to whatever is an appropriate value for a 64-bit bus access. Similarly we can read the

input values in 32-bit chunks or less. If we're careful, by writing to the write or read enable last, then the access will have all other bits set correctly before the transaction is instigated. However, this would take some cycles to do, cycles that are not really being correctly used. The update and update-response signals can be used to update the separate ports (and sub-sections of them) in delta time. The update signal will change state for each delta cycle access and will wait for a response on the update response signal. The delta argument of the `VWrite` and `VRead` API functions are used to indicate whether they are delta cycle updates or not. Normally this argument is set to non-zero for all the updates to accessing each port sub-section until the last one, allowing the simulation to move forward in time. (This delta-cycle update manifests itself to the VProc HDL by the `VPTicks` argument of a `$vsched` PLI task call returning -1.) The test HDL code that instantiates the VProc component and that uses this feature will need to wait on the `Update` signal swapping state, decode the transaction, update the addressed output, or read the addressed input, and then invert the `UpdateResponse` signal to allow VProc to continue.

This delta-cycle feature has been used in real-world models. The open-source PCIe root complex model was used to allow development of a PCIe endpoint, and to co-simulate with kernel software running on QEMU, and the open-source TCP/IP packet generator was used to generate test data to verify the integration of a 3rd party TCP/IP solution. Both used the delta-cycle feature to update interfaces much wider than the 32-bit transactions from the base calls to `$vsched`.



PCIe Model



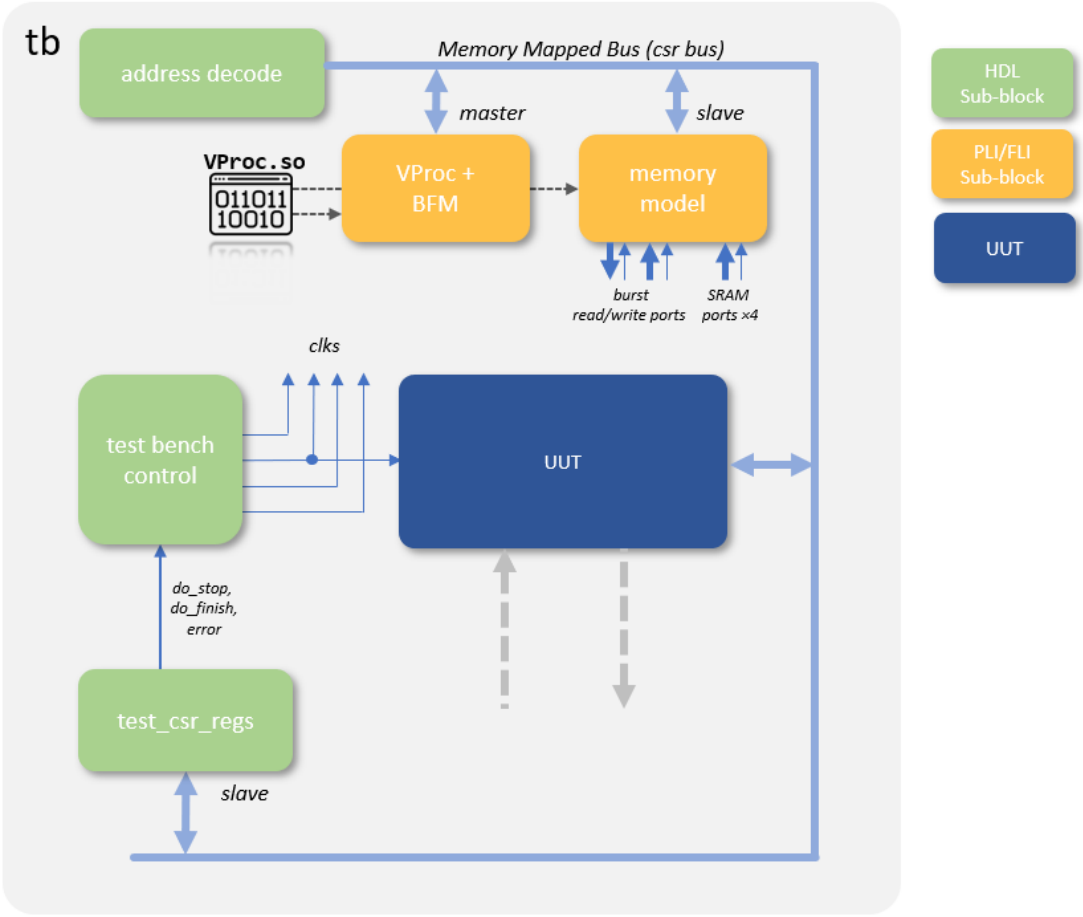
TCP/IPv4 model

A layer of software must be written to implement the appropriate protocols and do the correct delta cycle updates, along with some simple behavioural HDL to make use of the data-cycle update signalling in a bus-functional model. The majority of the complexity is meant to be in the software model to reduce the burden on simulation compute, and both of the example projects are light on HDL code. This

does mean, depending on the complexity of the model, quite a lot of software (the PCIe model's code is much bigger than the TCI/IP model's code, for instance) and may need collaboration between logic/verification engineering and software engineering. There is more detail in the VProc documentation, and the above two examples can be used as reference projects for the use of the delta-cycle features, with more details in their own documentation (here for PCIe, and here for TCP/IP)

A Generic Test Bench using VProc

So we've met the original brief, so how might this be used in a simulation test bench environment?



The diagram above shows a suggested setup. A top-level test bench (`tb`) instantiates a VProc component wrapped in a BFM module to implement a memory mapped bus protocol suitable for the unit under test, such as Avalon, Wishbone or AMBA AHB etc. The memory model, referenced in part 1, is also instantiated to give as much memory space as needed. The test bench needs code to decode the addresses from VProc to generate the sub-block selects (or equivalent), and some memory mapped registers for the test bench itself will be useful to allow writes to control the test

bench, via a control block, to generate clocks and resets, to stop or finish the simulation, or to post error data etc.

The UUT itself its instantiated as one or more of the IP blocks being developed (preferably wrapped into a single 'core' module). It would be expected that other verification IP blocks would be added to the UUT's other ports to drive them, and to capture output. These might be memory mapped devices as well, and so could then be controlled by the virtual processor test software. If the UUT has burst busses to memory, these can be hooked up to the memory model to capture write data, or to generate read responses with data patterns.

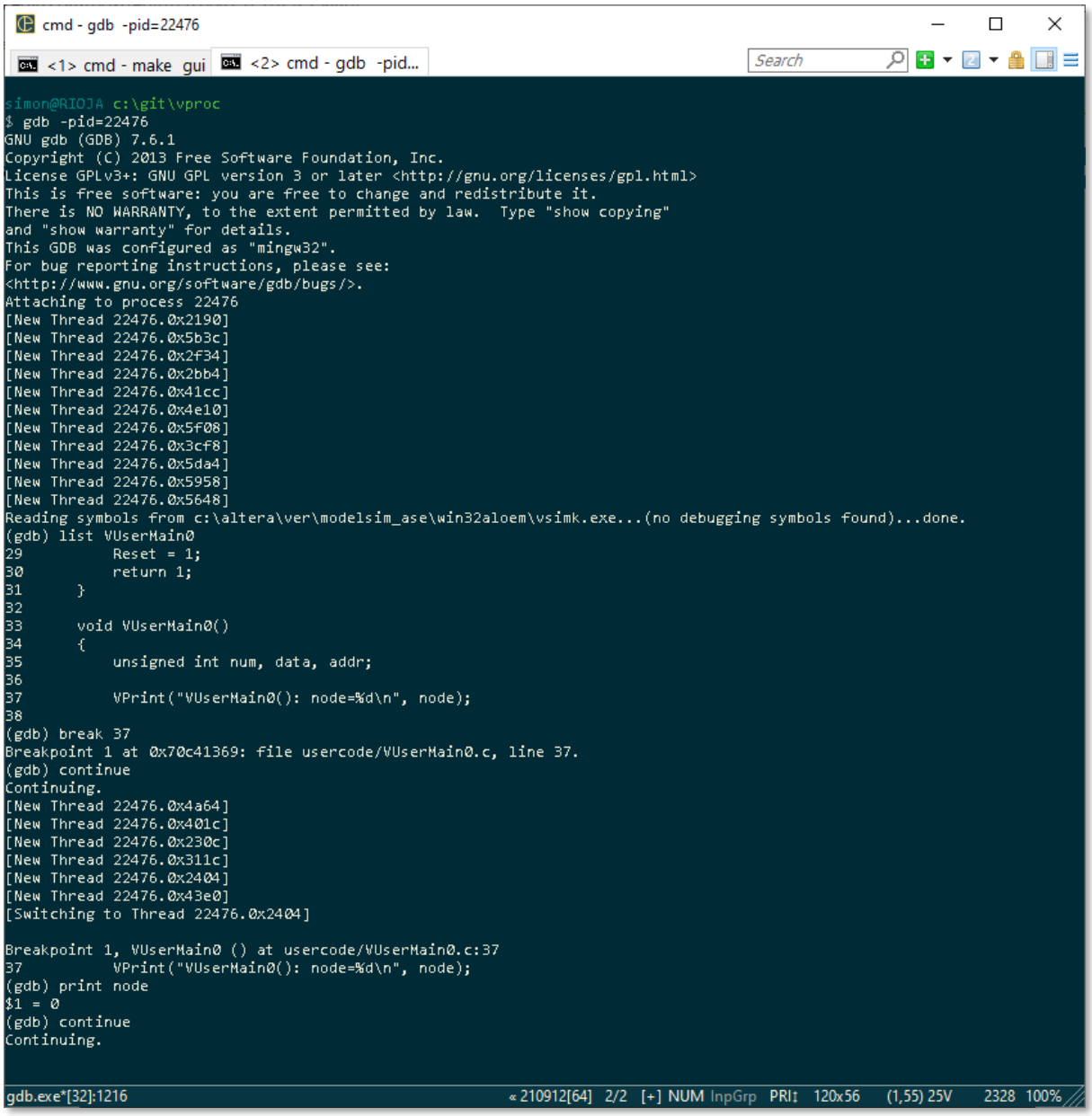
One very useful advantage of having VProc code using the thread method is that it has visibility of the memory model's API, as it is all compiled into the same shared object. Therefore the program running on the virtual processor can read and write directly to the model without the need to do bus accesses in simulation (though this is possible in the example shown above). This allows data to be loaded to the memory (e.g. from a file) at any location, very efficiently, consuming no simulation compute. Similarly, data written to memory from the UUT can be read directly and verified by the virtual processor software, again using no simulation compute. Strictly, the memory model is running in the simulation 'thread' whilst the user code is running in its own thread, which might seem hazardous. But, as mentioned before, when in the user thread, the simulation is blocked, and cannot be calling the memory model PLI functions, so it is perfectly safe to access any data in the memory model without conflict.

Debugging the VProc Software

With a virtual processing element, and a suitable lightweight BFM wrapper, we can now write software which compiles on our machine and run code of any complexity that we require to control our simulation. This moves a *lot* of computation away from the logic simulation. But, you may be wondering, how are we going to be able to debug the software? After all, it is not visible from the simulation tools used to debug the logic.

Well, it is natively compiled code, so we should be able to use a normal debugging tool, such as gdb, and therefore an IDE that sits on top of this, such as Eclipse. The simulation application will have loaded our shared object (e.g. `VProc.so`), with all the PLI code and our software, into its kernel simulation engine when it was executed and will be part of the process on the host machine that has this engine kernel code. For ModelSim, for instance, this is a process called `vsimk`. There are other `vsim` processes, but `vsimk` is the kernel simulation engine code. Assuming the shared object code was compiled with debug symbols (using the `-g` command line

option on `gcc`) then gdb can be attached to the running vsimk processes (with the `-pid=<PID>` command line argument) and the symbols be visible. The PID for vsimk can be found (on Windows, say) by using the task manager to inspect the running processes under the 'details' tab. Assuming the execution of vsim did not start the simulation running immediately, gdb can be attached to the processes and set up (setting breakpoints etc.) before the logic simulation is run, with '`run -all`' for example.



The above `gdb` session shows it being invoked to attach to a `vsimk` PID and then some of the source code is listed to show it's found the VProc code. A break point is set for the print statement and then the program allowed to continue. It is at that point that the simulator can be run (it will appear to have hung when `gdb` attached)

with, say, a '`run 1us`' command. The simulation will break when the breakpoint is reached in the VProc code, and state can be inspected or updated etc., just as for a normal `gdb` session. The simulation can be allowed to continue once more with another `gdb` continue command, and so on. This allows dual debugging of simulation and C/C++ VProc code, with break points set in both the logic simulation and the C/C++ code. Note, though, that if at a breakpoint in `gdb`, the logic will have 'frozen' and the `vsim` GUI/command line will be unresponsive. Similarly, if the logic simulation has halted, `gdb` will be unresponsive.

As mentioned before, `gdb` can be executed from within an IDE such as Eclipse, with all the advantages that that brings. This is not the article to go into that detail but, as debugging the VProc C/C++ code is the same as debugging normal code, a local software engineer should be able to help set that up and even to compile the shared object.

Conclusions

We took the basic methods of crossing from HDL to C/C++ from part 1 and have seen how it is possible to construct a means to have a natively compiled program read and write over a bus in an HDL simulation. Two methods were discussed, one using TCP/IP sockets and one using threads, both of which had advantages and disadvantages. Using the thread method we looked at a real-world use case of the virtual processor (VProc) and saw how this could be used to produce different verification processing element components driving various protocols, from simple 32-bit processors and extending to 64-bit busses, PCIe and TCP/IPv4. It was also shown how the code 'running' on the virtual processor can be debugged using standard debugging tools and IDEs.

In doing this we have not used *any* new features of the PLIs (of which there are many, many more) than we discussed in part 1. But we have now met the brief of code being natively compiled on the host machine that can read and write over a bus to registers and memory in a logic simulation.

So we must've come to the end, right? Not a bit of it. The verification manager who posed the original question would say yes (I suspect), but we can do better than that. Now that we have a 'virtual processor', might we be able run some or all of the embedded software that's being developed for a commercial project we might be working on, on the virtual processor? In other words, co-simulate the embedded software and logic, developing them in tandem before the silicon becomes available? Well, hopefully, but that must wait for the next, and final, part.

Part 3: Co-simulation

Introduction

In the first two articles of this series I did a review of some of the programming interfaces available for three popular HDL languages, describing how to use the minimum of features to cross the logic-software divide, with a real-world case study of an advance memory model (see part 1). Then I showed a couple of methods that can be used to allow a natively compiled program to interface with the simulation to do read and write accesses over a bus or interconnect—all without using any new features of any PLI. A real-world case study of a virtual processor, which uses one of these techniques, was then described (see part 2).

In this, the last article, I want to take the concepts from the first two articles and discuss how we might extend these concepts further to do co-simulation of embedded software and logic—again, without using any new features from the programming interfaces. The open-source virtual processor reviewed in part 2 will form the basis for these discussions, but the methods can be adapted. As ever, we will have a real-world case study, where the PCIe model alluded to in part 2 was expanded to allow run Linux kernel software to be co-simulated with the network logic it was controlling, utilising open-source simulation software (Verilator) to do so.

The content of this article spans the embedded software and logic boundary to the fullest extent, and so describes concepts from both disciplines, which maybe unfamiliar if you sit in the opposite camp. By its very nature, co-simulation is a cross-disciplined activity and requires co-operation between logic and logic-verification engineers with embedded software engineers (a good thing, right?). The targets for this article are all of these engineers and may require and dialogue between more than one person. If you can, though, read through at least to understand what might be possible and useful for your projects, and come away with questions for others with knowledge in the complementary discipline (or ask me!).

Interfacing Embedded Software to the Logic

In part 2 the VProc virtual processor was introduced, giving us the means to run a program on this element, doing accesses over a bus as if it were a processor core—but running code compiled for the machine it's running on, instead of for a particular processor architecture. Any embedded software we'd like to run on this virtual processor will, in all likelihood, be targeting a specific processor (e.g. RISC-V

or ARM) that's different to the host machine's processor. If, as is probable, it is written in a portable language such as C or C++, then it can still be compiled by the host machine's compiler, but we will have to solve some issues before this is possible. It may not be possible to do this for all embedded software if, say, it uses pre-compiled proprietary libraries where source code is not available (though there are possible solutions here too). It may be that only the lowest layers, nearest the logic, can be targeted at the virtual processor but that may still be useful. There are usually solutions to all these project specific problems but they will vary in degree of complexity and time to implement and a judgement call is required on effort versus reward and risk mitigation. When we look at one case study later in the article, we will see that it is possible to co-simulate to quite a complex degree by using the virtual processor concepts along with other open-source tools for simulation (Verilator) and virtual machines (QEMU).

Let's assume, for now, that the main issue we have solve is the software to accesses of the bus or interconnect. In a straightforward embedded system the registers of the logic that the driver software accesses might be in some contiguous uncached address space, which has some base address that the driver software knows (maybe via a virtual address, if an MMU based system), and the different logic component's registers are memory mapped within that region. The VProc component supplies a simple API to allow reads and writes, or maybe there is a layer of software to model more complex interconnect protocols such as PCIe or TCP/IP, but there will still present some sort of API for making accesses. I.e. a set of functions calls of object methods. The embedded software (or sub-set of it) won't know anything about these APIs and so something must be done. The approach will depend on the nature of the embedded software, the point in its development and its architecture—in particular whether it has a hardware access layer (HAL) and whether this is auto-generated or not.

The best-case scenario is if we are at the beginning of development, with little or no software written for the logic drivers, it will have a HAL, and this is to be auto-generated from a common register description using, say, JSON, XML, or even a spreadsheet, as the description language. Auto-generation of register logic and HAL software will not be covered here, as it is a whole other subject, but I have been involved with auto-generation in three separate companies, so it is very common and a valid assumption, I think. We will start, then, at this point.

Interfacing HAL Software

A hardware access layer (HAL) is a virtualisation layer between the logic that's memory mapped into the visible address space of the processor running the embedded software, and the driver software that is controlling that logic. It is

usually a thin layer, perhaps a set of classes that provide methods to access registers, their bit fields, or memories within the logic and the like. If all accesses to the logic space are through the HAL, then we have means to intercept these and make calls the VProc API (wither directly, or via a protocol layer model) to instigate the transfers.

If the HAL is auto-generated, then the simplest thing might be to auto-generate a new, replacement, HAL that makes these calls to the provided API. Most of the code would be common, but the source for the actual reads and writes are replaced with the API calls. Since both are auto-generated from the same source the risk of differing behaviours is, if not completely eliminated, certainly reduced to a low level.

If the HAL is not auto-generated, but if it is not yet written, then co-simulation can be more easily supported by implementing the code to have all the accesses to memory mapped space wrapped in thin functions or methods. The functions or methods for the target system then simply do a bus access as normal. For co-simulation compilation, these functions or methods can then be replaced or overloaded to make calls the virtual processor API instead.

The more difficult situation is where there is pre-existing code that we don't want to spend a lot of time upgrading for co-simulation support. Since reading or writing to a logic block's register is usually just a memory mapped access, they are likely, in pre-existing code to look some like:

```
regStatus = *pControllerBase[CTRL_STATUS_OFFSET];
*pControllerBase[CTRL_CONFIG_OFFSET] = ctrlCfgUpdate;
```

Here, the reads and writes simply use pointers to memory to get the data value from a register or to set it. When compiled for the target system, this is what would happen. How can this be 'intercepted' for co-simulation? Well, firstly, the assumption is that this is C++, as we need to use the features of this language. Secondly, the type of the registers' pointers mustn't be a system type (e.g. `uint32_t`) directly, but have a defined type which, for the target compilation can map to a system type. For co-simulation compilation, we will make the registers' pointer type a simple class and then overload the operators on that type for reading and assignment.

Below is a code fragment of some rudimentary operator overloads that will call external functions on reads and write when assigning or reading:

```
uint32_t read(uint32_t* in)
{
    /* access API */
    return value;
```

```
}


void write(uint32_t* addr, uint32_t in)
{
    /* access API */
}


class reg_t
{
public:
    uint32_t* pAddr;


    // Constructor
    reg_t(uint32_t*p = 0) : pAddr(p)
    {
    };

    // Overload = for writes
    void operator=(uint32_t in)
    {
       write(pAddr, in);
    };


    // Conversion of uint32_t for reads
    operator uint32_t() const
    {
        return read(pAddr);
    }


    // Overload & to return register address
    uint32_t* operator&()
    {
        return pAddr;
    };

};
```

This is just a simple example and other ways of achieving the same thing and of accessing bit-fields etc. are possible. However, this approach was used at one company where I was pushing for co-simulation, though I don't have access to this code and was it was implemented by an embedded software engineer, and not me, in any case. What was set up, though, were two compilation environments in the IDE and one could swap between targeting the platform or targeting the co-simulation environment.

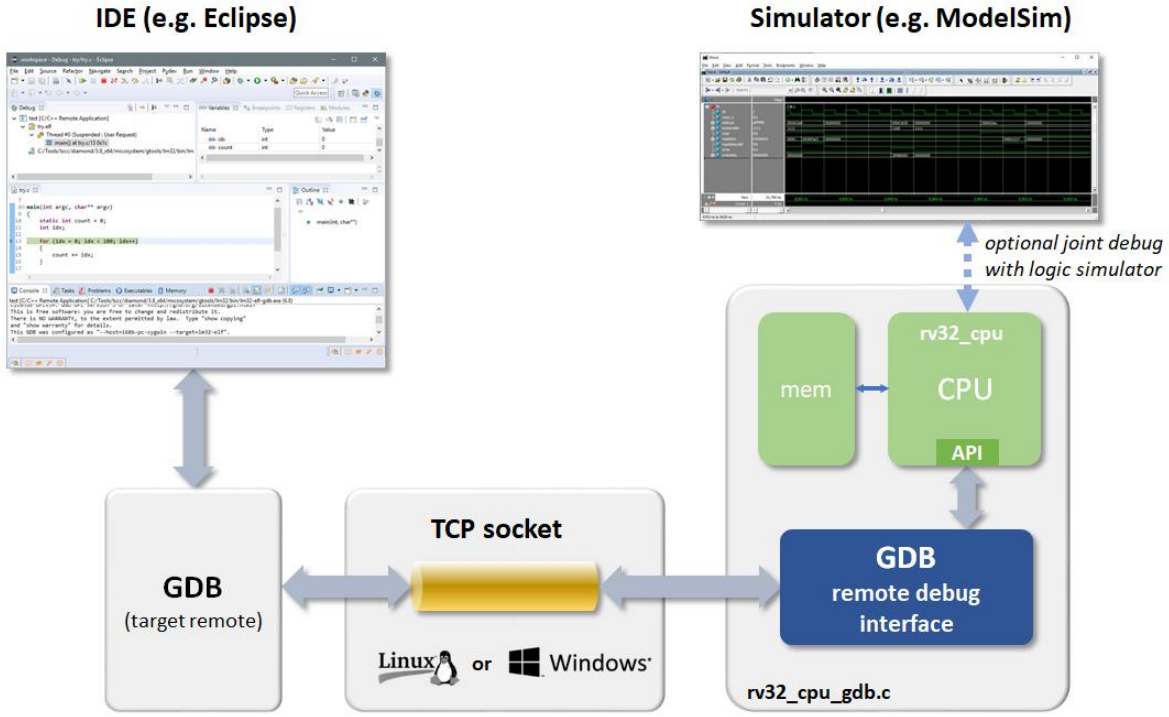
Running a Core Model

In looking at ways to interface the embedded software to the co-simulation environment there were some assumptions about the architecture that this code

would have. As mentioned before, if the software uses libraries targeting a particular process, and this is pivotal to doing any useful co-simulation, then what can we do?

Perhaps we have access to a C/C++ model of the core—such as an instruction set simulator. Since the virtual processor can 'run' native code then the memory accesses in the model can be redirected to do PLI API accesses instead. Now, if you've read the first article then you'll know that at the instigation of all this was to *avoid* having to cross-compile test code to a target processor architecture. However, that was not for co-simulation. The required code was test code to drive and test peripheral components, independent of the core development. But now we are trying to co-simulate embedded software with is counterpart logic this becomes viable. By running a core model as the software running on the virtual processor, we can run the embedded software on the core model.

This has already been done in the ISS supplied in the open-source rv32_cpu RISC-V project. The diagram below, taken from the ISS's documentation, shows the setup.



Here the RISC-V ISS is running on the VProc virtual processor. To interface it to the VProc API, use was made of the external callbacks supplied by the ISS. Functions can be registered with the ISS to call back if a memory access is made. The callback function either intercepts this access and returns a status indicating that it processed that bus access, or it returns indicating that it did not, and the ISS will then handle the access internally. (The details are all detailed in the ISS's

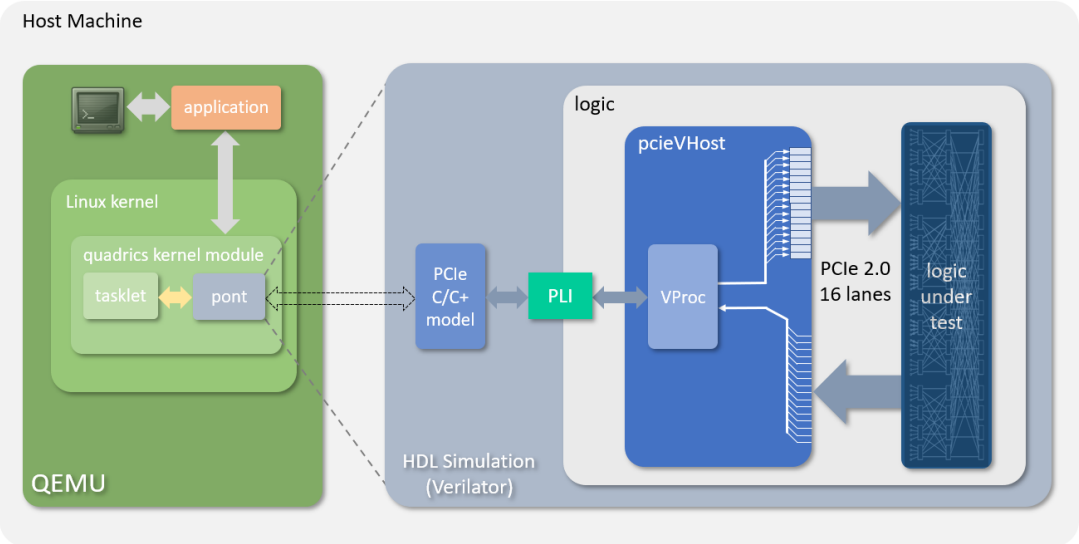
documentation.) On the software running on the virtual processor, along with the ISS, is also supplies a `gdb` interface for remote connection over a TCP/IP socket, allowing the loading and debugging of RISC-V programs onto the model. Now the embedded software is running on a model connected to a bus in the simulation. Some or all of the other SoC components may be instantiated in the logic simulation to test the driver code with the matching logic component.

Real World Use Case

I want to outline a use case in a commercial environment to demonstrate what can be done with these concepts with minimal overhead in effort. It is also a use case that uses most of the concepts we have discussed in these three articles, except the operator overloading (the embedded code is Kernel code and written in C).

Here I want to elaborate on the PCIe model mentioned in part 2. Originally, the model running on VProc was used in a 'traditional' sense, where test code was used to drive, via the model, the PCIe endpoint logic I was developing. This was done by a separate software engineer, giving better robustness to the testing (no false assumptions carried over to the test stimulus). The other parts of the software development team got wind of this setup and got interested. The issue we had was that they needed to run their kernel software to drive the network interface logic on a simulator. Previously this had been done by running it on a core implemented as synthesisable HDL running in the simulation. Not only was this slow, but the number of simulation licences was also severely restricted, with each licence running in to tens of thousands of dollars, causing a development bottle neck.

Work was done to migrate the VProc/PCIe model, along with the logic, to the open-source Verilator simulator. This was not too difficult a task and allowed each software engineer to have a development platform. The software to hook this to the rest of the software being developed was done by these engineers but, to my recollection, it looks something like the diagram below:



In summary, QEMU was used as a virtual Linux machine on which a Linux kernel was running. The Kernel code being developed was load as a module in the kernel. This code was some sort of co-operatively scheduled multi-tasking system, with each context known as a tasklet. Added to the list of tasklets was one dedicated to the simulation code and this was hooked up via a small layer developed by the software engineers, called 'pont' (or bridge) which virtualised away the details of the model. Within this pont, the rest of the PCIe model, VProc, and the simulator (with network the logic) all ran. The kernel drivers then had access to the PCIe fabric, just as they would when the ASIC arrived back from fabrication. As I understand it (and remember I am working from memory from some years ago) an application connected to terminal could access the drivers to instigate PCIe communications fast enough that one could sit and do this manually at that terminal without impractical delays.

Conclusions

So, we have looked at ways we might interface part, or all, of some-embedded software with a virtual processor, allowing access into a logic simulation environment. This allows development and testing of logic and the driver software to be done together, mitigating the risk of integrating the two only when silicon becomes available, with possible logic issues insurmountable by software changes. Even in an FPGA environment, I would argue this bring benefits. Integrating large amounts of code, after weeks or months of development will, in all likelihood, through up many issues, with some issues hiding others, making estimating a time for completion difficult. This will also be nearer the planned delivery date, with less time to catch-up for unplanned, unexpected, and complex problems. By integrating early, as development in both disciplines proceeds, not only does this mitigate the

integration risks, it is also an easier debug environment in which to work. As we have seen, one can debug both software and logic together in this environment. It is possible on the FPGA environment (ChipScope or SignalTap for the logic, and gdb/JTAG for software) but this is much harder and more time consuming, I would argue.

We have also looked at a commercial development case study of a multi-layered system all the way from simulated logic right up to an OS running on a virtual machine connected to an application and terminal. And we ***still*** haven't used any more complex features of the PLIs than that introduced in part 1 of this series. Obviously, in that development my contribution stopped at the VProc and PCIe protocol model, and the additional layering of software was added by multiple other software engineers. But the point is that, having crossed the chasm between the logic and software, the possibilities expand by orders of magnitude. Across the PLI we have built a rope-bridge, rather than the Golden Gate Bridge, by restricting the use to the most basic features of the programming interfaces, but that has not restricted, by very much, what is achievable and what has actually been done within commercial settings using open-source tools, all of which are still available. I'm sure (if you 've got this far!) you can think of new and better ways to use these concepts.