Using OVP with Specman/ISX

Posedge Software

This application note presents an example implementation of an OVP MIPS processor model integrated into a Specman testbench using ISX. A baremetal MIPS platform, modeled with OVP, is used along with a simple target application that reads and writes internal variables. In addition to details of this particular implementation, other suggestions for using OVP and ISX are given.

Posedge Software, Inc.

Web: http://posedgesoft.com
Email: info@posedgesoft.com
Phone: (612) 644-0156
Fax: (763) 428-1884

Introduction

Specman Elite and its e language are typically used to create testbenches to verify a hardware design. This target hardware is usually simulated with a Verilog (or other HDL) simulator, with Specman generating stimulus and checking results. In a system-level verification environment (VE), the target software is often included, and is executed on a processor model running in the hardware simulator. While this can accurately simulate the software execution, it can be extremely resource intensive and greatly increase simulation times. Often, only a small part of the software is used to keep simulation times reasonable.

Open Virtual Platforms (OVP), and in particular the M* tools from Imperas provide a fast software execution environment for a variety of processors and peripherals. In addition to high performance, these tools provide mechanisms for debugging the software and monitoring the system with minimal impact on the performance or the state of the target system.

By combining the above tools, it is possible to implement a powerful constrained-random testbench that can be used for the verification of both hardware and software.

Intended Audience

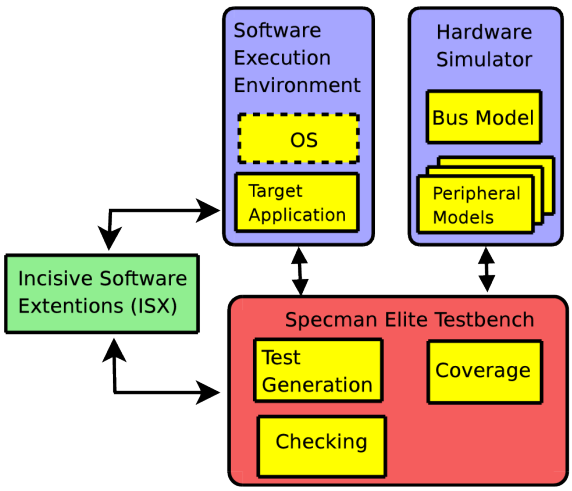
The intended audience for this application note is developers looking to create a software verification environment using virtualization technology. While further knowledge of OVP, the Imperas tools, and Specman would be needed to carry out the suggested implementation, this document provides a starting point.

Overview of a Specman/ISX Environment

While Specman has many features for stimulus generation, coverage, and checking, it is not generally used on its own, but with hardware simulators such as NC-Sim or VCS. Support for these hardware simulators is integrated into Specman, but it can also be used effectively with other types of simulation environments and tools.

Specman can communicate with external tools using a variety of mechanisms. Two commonly used ones are the Co-verification link (CVL) and the C interface. Either of these mechanisms allow for C functions to be called from the e language used by Specman, or allow e methods to be called from C. They are typically used to provide additional control of the simulation environment, or access other simulation models such as memory.

On top of this, other features such as the Incisive Software Extensions (ISX) can be used. ISX is a toolkit which provides mechanisms for treating software similar to other hardware components, and therefor use Specman to test software. For example, instead of the testbench driving a value onto a port of a hardware model, a function can be called in the software. ISX works by extracting debug info from the target software, and then reading and writing the memory space in which the target software is running.



System Level Verification Environment

ISX is typically used to perform Co-verification, where the software is tested at the same time as the hardware. In addition to testing software earlier in the design cycle, co-verification allows testing of hardware-software interaction and other end cases that would be difficult to test. Otherwise, this testing (if even possible) is not done until real hardware is available.

When performing Co-verification, or using Specman just for software verification, the target software should ideally be run unmodified, while keeping it synchronized with the testbench and other hardware models. While Verilog processor model can be used for this, they significantly increase the simulation time. Instead, the OVP models and simulators can be used for this task, as they provide a fast, accurate environment for executing software. This combination of Specman, ISX, and OVP can be used for system-level verification, or to create a powerful environment specifically for testing software.

Building the OVP Platform

In most of the OVP examples, the platform is a standalone executable, implemented as a C source file with a single main routine. To more easily integrate with Specman, it is instead implemented as a shared library with separate initialize, simulate, and exit functions.

```
int platform_init(void * env, char * appName, unsigned int mailbox_addr) {

    icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC, 0,0);
    const char *mips32Model    = icmGetVlnvString(NULL, "imperas.com", "processor",
"mips32", "1.0", "model");
    const char *mips32Semihost = icmGetVlnvString(NULL, "imperas.com", "semihosting",
"mips32SDE", "1.0", "model");

    // create a processor instance
    processor = icmNewProcessor(
        "CPU1",             // CPU name
        "mips32",           // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        mips32Model,        // model file
        "modelAttrs",       // morpher attributes
        ICM_ATTR_DEBUG,          // attributes
        0,                  // user-defined attributes
        mips32Semihost,     // semi-hosting file
        "modelAttrs"        // semi-hosting attributes
    );

    // load the application executable file into processor memory space
    if(!icmLoadProcessorMemory(processor, appName, False, False)) {
        return -1;
    }

    ...

}


int platform_simulate(int instrs) {
    ...
}


void platform_exit()
{
    // free the processor
    icmFreeProcessor(processor);
}
```

In addition to separating the main routine, the processor object or other variables should declarded as global, so they can be used by all the functions.

The platform should then be compiled as a shared library. An existing makefile or other build system can be used to perform this, or the following commands can be used.

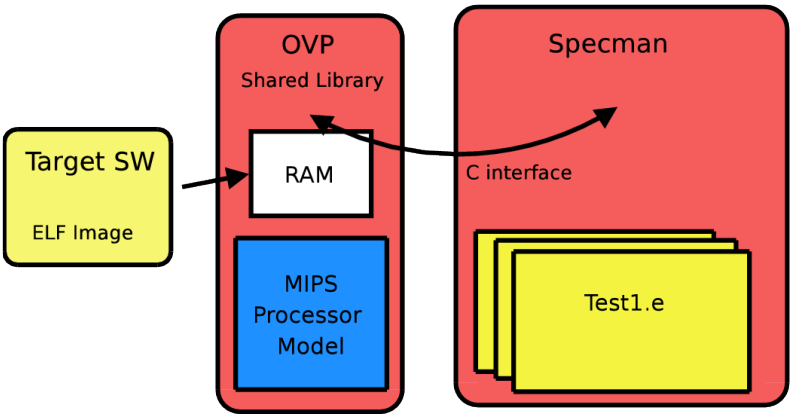
```
>gcc -fPIC -c platform.c -I${IMPERAS_HOME}/ImpPublic/include/host

>ld -E -shared -lpthread ${IMPERAS_HOME}/bin/Linux/libOVPsim.so -o ovpplatform.so
platform.o
```

Specman C Interface

Two commonly used mechanisms for interfacing Specman with external tools are the Co-verification link (CVL) and the C interface. CVL makes use of network sockets, and can be used by Specman to communicate with a separate program running on either the same or a different computer. It is the preferred mechanism when the other tool needs to be run as a separate executable, or possibly run on a different platform or operating system.

The Specman C interface works with shared libraries that are linked statically or dynamically with Specman, so therefore the tool must run on the same platform as Specman. When possible, this mechanism is preferable, as there is significantly less overhead and complexity than a CVL connection.

The C interface is used in this application note to communicate with the M* tools from Imperas, as this allows for the greatest performance and most flexibility. However, a CVL connection could potentially be used to communicate with the Windows version of OVPsim. If this was the case, the platform should still be split into different functions, but instead linked with a CVL library and built as a standalone executable.



Specman to OVP integration

Once the platform library is created, it needs to be made accessible from the e testbench. A method (prototype) is declared in the e code using the **`dynamic C routine`** keywords. The name of a shared library can also be specified, where Specman will search the LD_LIBRARY_PATH for the given library name.

```
platform_init(s1:string, mbox_addr:uint): int is dynamic C routine ovpplatform:;

platform_simulate(p1:int, p2:int): int is foreign dynamic C routine ovpplatform:;

platform_exit() is foreign dynamic C routine ovpplatform:;
```

Note that in the above code the **`platform_simulate()`** and **`platform_exit()`** methods are declared as **`foreign`** while the **`platform_init()`** is not. When the **`foreign`** keyword is used, the C function is assumed to be a normal function and is not aware it is being called from e. When not used, an extra parameter will be passed as the first argument to the C function which provides information about the calling e method's environment. This environmental info is needed for the C function to make calls back into e.

After the methods have been declared, they can be used like any other e method. Also note that these are not e Time-Consuming Methods (TCMs), so no Specman simulation time will elapse while these methods are being called.

Backdoor Memory Access

One of the primary components of ISX is the Generic Software Adapter (GSA). The GSA allows for monitoring and control of the target software by using a software mailbox located in the target memory. Specman must therefore be able to read and write to this memory location. While this could be implemented by communicating with the target process using network sockets or similar methods, this is not desirable as it requires significantly more modifications to your target software and may have unintended consequences. It is much preferable to have backdoor memory access, where the target memory can be read and written independently of the target process. While this could be difficult to do with the real target hardware, it can be accomplished relatively easy with most virtual or simulated systems. With a Verilog processor model, the memory is accessible and often modeled in the hardware simulator. Using OVP, this backdoor memory access is also trivial, as API functions exist for reading and writing to memory.

Backdoor memory access functions should be implemented in the platform shared library. Besides calling the relevant API functions, endian conversion can be performed in these functions.

```
unsigned int bd_read_int(unsigned int address)
{
   unsigned int value;
   sn_mode = TRUE;
   icmReadProcessorMemory(processor, address, &value, 4);
   // Endian conversion
   value = (value & 0x000000FF) << 24 | (value & 0x0000FF00) << 8 | (value & 0x00FF0000)
>> 8 | (value & 0xFF000000) >> 24;
   //icmPrintf("bd_read_int(0x%X,0x%X)\n",address,value);
   sn_mode = FALSE;
   return value;
}

void bd_write_int(unsigned int address, unsigned int value)
{
   sn_mode = TRUE;
   // Endian conversion
   value = (value & 0x000000FF) << 24 | (value & 0x0000FF00) << 8 | (value & 0x00FF0000)
>> 8 | (value & 0xFF000000) >> 24;
   icmWriteProcessorMemory(processor, address, &value, 4);
   //icmPrintf("bd_write_int(0x%X,0x%X)\n",address,value);
   sn_mode = FALSE;
}
```

Also note in the above code, that an sn_mode flag that was created is set before the memory access API functions are called. When these API functions are called, the act as if the target processor performed the access, and therefore would trigger any memory callbacks that were enabled. This sn_mode flag can be used by those callbacks to check if it is a real access by the target process, or backdoor functions.

These backdoor functions also need to be made available in Specman using the C interface.

```
bd_read_int(address: uint) : uint is foreign dynamic C routine ovpplatform:;
bd_write_int(address: uint, value: uint) is foreign dynamic C routine ovpplatform:;
```

At a minimum, these two function for reading and writing 32-bit values need to be implemented. Backdoor access functions for different sizes, such as bytes or strings, can also be implemented and will be used by ISX when available to minimize the number of backdoor memory accesses that need to be made.

Position	Decription	Example Values
0x0	Signature	0x5f677361
0x1	Checksum	0xaada8f1e
0x3	Status	GS_MB_READY(0x1)
0x6	Message Flag	&msg_flag
0x8	Message String	&msg_str[0]
0x10	Function Select	0x8
0x11	Argument 1	0x1
0x12	Return Value	0x0
...	...	...

Example GSA Mailbox

Simulation Control and Synchronization

Once communication is implemented between Specman and OVP, some mechanism needs to be implemented to keep them synchronized. In general, the OVP simulation should run for one timeslice (a specified number of cycles), or until some event occurs, and then hand control back to Specman to process any pending events in the testbench. After Specman processes these events, which should not take any OVP simulation time, it hands control back to OVP.

One option is to simply implement the scheduler in Specman, such that after a fixed number of processor cycles, Specman will regain control. The difficulty with this is that in order to have very fine resolution of events that occurred, you need to run for a smaller number of cycles in each processor timeslice, which greatly impacts performance of OVP.

Another approach is to essentially let the simulated processor run free and have Specman just respond to events such as memory accesses. This would be desirable, but there may also be cases where actions should be triggered by other events occurring within the testbench.

A third approach can be taken, where the scheduler is implemented in Specman, but if a particular event occurs within the target system that requires interaction by Specman, it yields, or temporarily halts the simulation, which causes control to be passed back to Specman. After Specman processes any pending events the simulation is restarted. This approach is used in this application note.

The **`platform_simulate()`** function in the platform shared library should first be implemented to run the simulation for a specified number of instructions. This value could be be hard-coded, or a value which is specified later in the e testbench.

```
int platform_simulate(int instrs)
{
    // simulate the platform
    int status = icmSimulate(processor, instrs);
    // was simulation interrupted or did it complete
    return status;
}
```

A scheduler loop can then be implemented in e as a TCM. In e, a TCM is a method that is associated with an event or other temporal expression. Once started, this TCM will begin executing when the specified event occurs, and will execute until finished or until a wait (or other temporal action) occurs.

```
ovp_scheduler() @timeslice is {
    var status : int;
```

```
    out("Run first simulator timeslice\n");

    while TRUE {

          status = platform_simulate(10000);
          if (status == 3)
          {
                out("SW Exiting!");
                stop_run();
          };
          emit wakeup_sn$;
          wait cycle;
          out("Run next simulator timeslice\n");
   };

};
```

The above scheduler is implemented as an infinite loop which begins execution when the timeslice event occurs, it then calls the platform_simulate() method, and then waits until the next cycle of the specified event (timeslice) before repeating.

The **`emit wakeup_sn$`** action is also called after each simulated timeslice to indicate to ISX that the software execution has advanced and that it should check if there are any pending operation that should be performed.

While the above implements a basic scheduler, the more optimal solution is to halt execution and pass control to Specman and ISX whenever the target software is waiting for data from the testbench. Without this, the target software will busy-wait until the end of the simulator timeslice. This change can be implemented with several modifications to the platform shared library.

When using ISX, once the target software has finished calling the desired function, or performing the previously specified operation, it will continually check the status value in the mailbox. By monitoring this location in memory yit can be determined when the target software is waiting for further instructions. This can be done by using a callback mechanism in the OVP models.

```
icmAddReadCallback(
       processor,       // processor
       mailbox_addr,      // low address
       mailbox_addr + 0x100,         // high address
       mailboxReadCB,       // callback to invoke
       "" // user data passed to callback
);
```

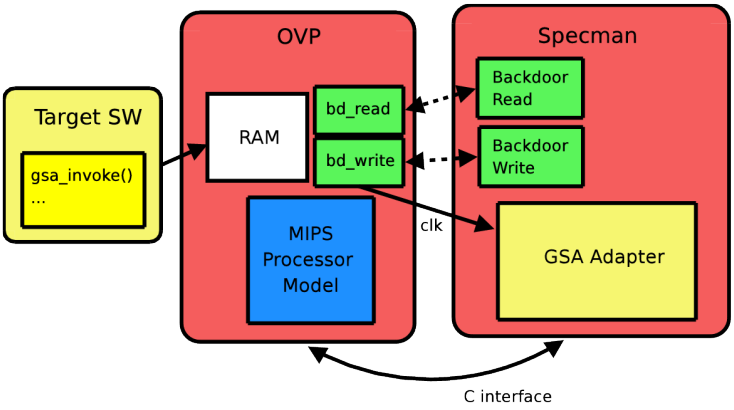
The above function call is added to the **`platform_init()`** function, and will add a callback whenever any location in the mailbox is read by the processor. Note that the mailbox address was passed as an argument to the **`platform_init()`** function, and is determine by other ISX components.

```
static ICM_MEM_READ_FN(mailboxReadCB) {
   if (!sn_mode && address == mbox_addr + 12)
   {
          icmYield(processor);
   }
}
```

The above code defines the callback that will be executed. In this example, while the callback is specified over the entire mailbox, there is only a particular address within it (the fourth 32-bit value, or starting at the 12th byte) that is of interest. Also note the sn_mode flag to determine if it is a real processor read or a backdoor access.

GSA and Software UVC

A GSA mini-adapter needs to be created to interface with the target software. This consists of code for both the e environment and target application needed to implement the mailbox and communication between the two. This could be created manually by calling various e macros. In addition to declaring and naming the adapter, e ports need to be declared corresponding to functions or variables in the target software that are of interest. This approach is documented in the ISX User Guide shipped with Specman.



Basic GSA Interfacing

Another approach is to use the ISX Builder tool. This tool will read debugging information from the target software and provide a graphical interface where functions and variables of interest can be selected. A template for the GSA code along with an example verification environment will then be generated. Note that while this code will not work with OVP as is, and will require some modifications, it is an effective approach for users new to ISX. More documentation on using ISX Builder is included with it, but a summary of the needed steps follows.

ISX Builder is a plugin for the Incisive Verification Builder (IVB). Once IVB is started, a new project should be created. This project defines the complete verification environment including the OVP platform, the target application, and any other hardware models that would be included. ISX Builder can then be started by right-clicking in it under the “Create UVCs” category, and selecting “Activate Wizard”. In the dialog that appears, an option to chose between “Simulator” or “Host Machine” is given. The “Host Machine” option can be used, as the “Simulator” option generates extra code for syncing with a hardware simulation clock which is not relevant with the OVP simulators. Further information should then be entered about the target software, including the executable name, and functions and variables to create ports for. Once the wizard is complete, the code can be generated by right-clicking on the created UVC and selecting “Generate Wizard Code”.

A library of code will have been generated by the ISX Builder. This contains a Universal Verification Component (UVC) for the OVP platform. This will be specific to the OVP platform that was described earlier in the application note, but is not specific to the target software and could be reused in other projects. An example configuration using this software UVC to test the target software was also generated. Several changes need to be made to both the UVC and configuration before they can be used.

The generated UVC and configuration files assume CVL is being used to communicate with the target, so support needs to be added for communicating with the OVP platform using the Specman C interface. First, a new subtype of the sw_uvc_mem_intf_u should be created.

```
extend sw_uvc_mem_intf_t : [OVP];

extend OVP sw_uvc_mem_intf_u {
   sw_interface : ovp_sw_interface is instance;

   init_adapter() is empty;

   backdoor_write_int(address:uint, data : int) is {
      sw_interface.bd_write_int(address, data);
   };

   backdoor_read_int(address:uint) : uint is {
      var data: uint;
      sw_interface.bd_read_int(address, data);
      result = data;
   };

   backdoor_get_mbox() : uint is {
      result = sw_interface.mailbox_address;
   };

};
```

This defines the required methods for the GSA mini-adapter. A new unit should then be created for the remaining interface code. This unit should include the C interface methods and OVP scheduler which were presented earlier.

```
unit  ovp_sw_interface {

   !mailbox_address: uint;
   !application_name: string;
   event timeslice is {@timeslice; cycle};

   bd_read_int(address: uint) : uint is foreign dynamic C routine ovpplatform:;
   bd_write_int(address: uint, value: uint) is foreign dynamic C routine ovpplatform:;

   specman_store_mailbox(mbox: uint) @sys.any is {
      mailbox_address = mbox;
   };


   platform_init(s1:string, mbox_addr:uint): int is dynamic C routine ovpplatform:;
   ...

   ovp_scheduler() @timeslice is {
   ...
   };

   wakeup_sn : out event_port is instance;
   keep bind(env.wakeup_sn, wakeup_sn);

   run() is also {
   var status : int;
   status = platform_init(adapter.sw_executable, mailbox_address);

   start ovp_scheduler();
   emit timeslice;
  };

  quit() is also {
      platform_exit();
  };


};
```

In the above unit, the platform_init is not called until the test is run, while the scheduler is started immediately after. This should be all the necessary changes to the software UVC. A configuration is

needed to use this UVC. An example of this, but using CVL, was generated by ISX Builder and includes the GSA port declarations. This should be modified to use OVP.

```
// The miniadapter provides backdoor access to memories in the design.
extend OVP sw_mailbox_adapter {

   mbox_addr : uint;
   keep pack_options() == packing.low_big_endian;

  // Constrain the software executable to point to the elf file
  keep soft sw_executable ==
"/home/hvonbank/work/ovp/sandbox/baremetal/c_intf/target/simple.elf";

  -- User may set the mailbox pointer here. Following is an example mechanism
  -- using objdump. however any mechanism may be used. If this method remains
  -- unextended then GSA will search for the mailbox in defined ranges.
  get_specman_mbox_ptr(): uint is also {
    var addr: list of string;

    -- Extract mailbox address from executable file
    addr = str_split(output_from("objdump -t
/home/hvonbank/work/ovp/sandbox/baremetal/c_intf/target/simple.elf | grep -m 1
gsa_mailbox")[0]," ");
    result = append("0x", addr[0]).as_a(uint);
  };


  -- User may set the size of the mailbox here. If left undefined GSA
  -- will determine the required size of the mailbox
  get_specman_mbox_size() : uint is  {
   result = 256;
  };
};

extend sw_uvc_env {

  keep soft adapter.kind == OVP;
  keep soft agent() == "sw_mailbox";

};
```

Target Software

The target software used in this example is primarily composed of several functions that read or modify a state and a data variable in memory. While this a very basic example, it illustrates a model that could be used in more complex applications.

```
void simple_initialize()
{
    printf("simple_initialize() \n");
    current_state = init;
    data_value = 0xffffffff;
}

void simple_write(unsigned long val)
{
    if (val != check_value)
        current_state = error;
    else
        current_state = write_enabled;

    data_value = val;

    printf("simple_write() value: %x\n",data_value);
}
```

```
unsigned long simple_read()
{
    current_state = read_enabled;

    printf("simple_read() value: %x\n",data_value);

    return(data_value);
}
```

After the GSA mini-adapter is created, the target software must be modified to make use of it. This is accomplished by generating a C stub files based on the mini-adapter, and compiling it into the target application, along with several other changes to the target source code. The stub files for the mini-adapter are generated by Specman using the write stubs command along with the name of the mini-adapter (in this example sw_mailbox).

```
>specman -c "load ex_ovp_pkg/main_sve/ex_ovp_simple_pkg_main_sve; write stubs -sw_mailbox"
```

A C source file and header with starting with the name of the mini-adapter will be generated (i.e. sw_mailbox_specman.*).

In many application, the control loop is implemented in the main() function. When using ISX, this control loop is moved into the e testbench, such that it can be generated dynamically for each test without needing to modify the source code. This is done by replacing the main function with calls to GSA functions that are located in the stub files.

```
int main() {
  sn_gsa_init();
  while (1) {
    sn_gsa_wait();
  }
}
```

This new main function initializes the GSA mailbox, and then begins calling the sn_gsa_wait() function. This function will check the status flag in the mailbox for commands from the testbench. If a command is pending, it will call the desired function in the target application, and write the return values back to the mailbox.

These target source files should then be recompiled and linked to create a new application image.

Writing Tests

Once the rest of the VE is in place, individual tests need to be written to exercise the target software. Functions within the target software are represented in the testbench as e sequences. A test can be written by adding sequence actions to the body of the **`MAIN`** sequence. A sequence is typically performed, or run, by using the **`do`** keyword.

```
  body() @driver.clock is only {
      do simple_initialize;

      for i from 1 to 100 do {
              var ret : uint;

          do simple_write keeping { .val == i };
          do simple_read;
          ret = simple_read.return_val;

          if ret != i then {     //verify the result
              outf("Test FAILED.\n");
              dut_error("Data mismatch %x",i,ret);
```

```
         }
         else {
            outf("OK.....\n");
         };
   };
   //do simple_exit;

    outf("\n\nTest PASSED.\n\n");
};
```

The above example implements a simple directed test that writes and then reads values 1 to 100, while checking that the result matches. Specman also has powerful features for constrained-random generating. The above test can be easily modified to make use of these.

```
body() @driver.clock is only {
   do simple_initialize;

   for i from 1 to 100 do {
      var d1 : uint;
      var ret : uint;
      gen d1 keeping {it < 10000};

      outf("Generated value: %x\n",d1);

      do simple_write keeping { .val == d1 };
      do simple_read;
      ret = simple_read.return_val;

      outf("Read Value: %x\n",ret);
      if d1 != ret then {     //verify the result
         outf("Test FAILED.\n");
         dut_error("Data mismatch %x",d1,ret);
      }
      else {
         outf("OK.....\n");
      };
   };
   //do simple_exit;

   outf("\n\nTest PASSED.\n\n");
};
```

Instead of using the index value for writing and reading, a new random value is generated. Constraints can be added to this, such as keeping its value under 10000.

Summary

The combination of OVP models and the Imperas M* tools, along with Specman and ISX creates a powerful verification environment. Specman provides powerful test generation and automation features, while OVP and the Imperas tools provide fast and flexible simulation of software execution. This environment could be used for system-level verification, or strictly for software verification.

Some of the steps needed to integrate these tools involve creating a simulation platform a communication mechanism with Specman, providing backdoor memory access and simulation control, and create a software verification component and writing tests.

These same basic steps could be followed to create a more powerful verification environment for other tasks such as using a multi-threaded or multi-core processor, targets running Linux or another operating system, and testing of a kernel, kernel drivers, or user-space applications.

Other Resources

Posedge Software:
http://www.posedgesoft.com
OVPworld:
http://www.ovpworld.org
Imperas:
http://ww.imperas.com
Cadence:
http://www.cadence.com

Posedge Software, Inc.

Website: http://posedgesoft.com
Email: info@posedgesoft.com
Phone: (612) 644-0156
Fax: (763) 428-1884