Designing PSS Environment Integration for Maximum Reuse

Matthew Ballance
Mentor, A Siemens Business
8005 SW Boeckman Rd, Wilsonville, OR, 97070

***Abstract*- The Accellera PSS language standard enables users to capture a model of test intent that is portable across verification levels and execution platforms. Challenges in reusing the integration between the PSS model and the environment severely impacts the overall reuse benefits of applying PSS. This paper highlights the challenges in productively integrating PSS with the environment, and covers key criteria for a PSS/environment integration that maximizes reuse of both the environment elements and the PSS description. It describes key elements of a framework that maximizes PSS and test realization reuse across UVM-based testbench environments, and between UVM and embedded-software environments.**

I. INTRODUCTION

The Accellera PSS language standard [1] enables users to capture a model of test intent that is portable across verification levels and execution platforms. The PSS language captures test intent – the high-level design of what should be tested – while the task of carrying out the test intent is handled by test realization code in the UVM or embedded software environment in which the tests created from a PSS model execute.

A PSS model is logically separated into two pieces – one for modeling test intent and the other (test realization) for carrying out that test intent, as shown in Figure 1.

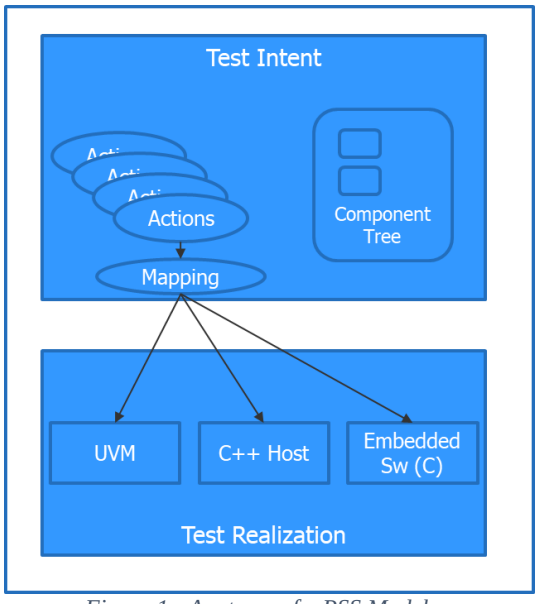

Figure 1 - Anatomy of a PSS Model

The PSS language features used to model test intent are heavily constraint-based, and capture value relationships that model a test space. The PSS language provides two mechanisms for connecting this high-level constraint-driven model to test realization code: procedure calls, and code templates (target templates). Both of these mechanisms are captured using the *exec blocks* construct.

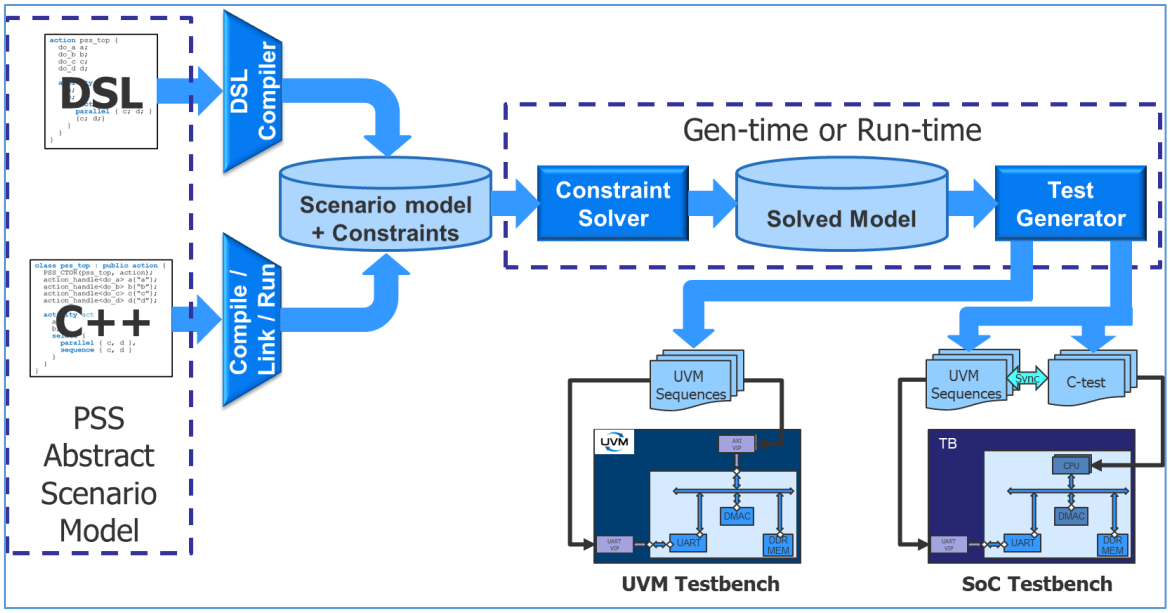

Figure 2 - Generating Tests from a PSS Model

The goal of capturing the PSS model and mapping to test realization is, of course, to create tests to run in the target environment. PSS supports either pre-generating these tests or generating them on-the-fly as the target environment runs.

A typical application of pre-generated tests are SoC platforms where the test created from the PSS model must be an embedded-software program expressed in either C or assembly language. Due to the restrictions of the bare-metal software-execution environments supported by an SoC test environment, the PSS test can have relatively few external dependencies. It certainly cannot depend on an external solver that evaluates the PSS model as the SoC executes.

A typical application of tests generated on-the-fly is when integrating into a UVM testbench or a SystemC environment. These environments are typically already designed with constrained-random techniques in mind. In other words, these environments are typically run in regression mode with different seeds and the assumption that different stimulus will result from using a different seed. When using on-the-fly test generation, the behavior of a PSS model integrated into these environments matches those assumptions.

This paper will use a simple DMA engine as the basis for examples. The DMA engine has 8 channels that can perform a variety of transfer types.

The most fundamental operation the DMA can perform is to transfer data from one region of memory to another.

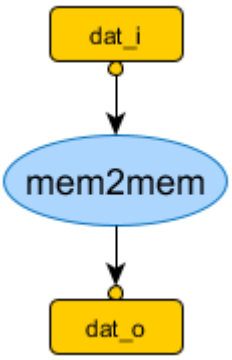

Figure 3 - Diagram of a Memory-to-Memory Transfer

```
action wb_dma_xfer_a : wb_dma_a {
  // The channel this transfer runs on
  rand bit[3]                   channel;

  // Total transfers to perform
  rand bit[16]                  tot_sz;

  // Bytes to transfer at a time (1, 2, 4)
  rand bit[4]in [1,2,4]   trn_sz;
}

action mem2mem_a : wb_dma_xfer_a {
  input data_mem_b              dat_i;
  output data_mem_b             dat_o;

  constraint {
    dat_i.sz == (tot_sz * trn_sz);
    dat_i.sz == dat_o.sz;
  }
}
```

Figure 4 - Memory-to-Memory Action

Figure 4 shows a simple atomic action for controlling the DMA engine to transfer data between two memory regions. The atomic action models the relationships needed to model the intent of a DMA transfer, without specifying how the DMA engine will be programmed to carry out the DMA transfer. We have two primary options for connecting this *test intent* to *test realization* within the testbench that will actually program the DMA engine to carry out the intended transfer.

The PSS Target Template exec block allows the user to specify literal snippets of code that the PSS processing tool. When we integrate into a SystemVerilog environment, our PSS model will produce a UVM virtual sequence. We can choose to implement the DMA memory-to-memory transfer in the same way that we would when writing a UVM sequence by hand: by setting up a UVM sequence item and calling start_item/finish_item as shown in Figure 5.

```
extend action wb_dma_c::mem2mem_a {

	exec body SV = """
		begin
		wb_dma_descriptor		desc =
			wb_dma_descriptor::type_id::create();
		desc.channel = {{channel}};
		desc.mode = 0;
		desc.inc_src = 1;
		desc.inc_dst = 1;
		desc.src_sel = 0;
		desc.dst_sel = 1;
		desc.tot_sz = {{tot_sz}};
		desc.trn_sz = {{trn_sz}};
		desc.chk_sz = 16;
		desc.src_addr = {{dat_i.addr}};
		desc.dst_addr = {{dat_o.addr}};

		start_item(desc);
		finish_item(desc);
		end
	""";
}
```

Figure 5 - SystemVerilog Target-Template Implementation

Using the target-template approach to specifying test realization has some weaknesses. First, it requires us to pre-generate tests, which means that the user needs to manage many generated test files instead of simply running simulation with different seeds. Secondly, writing in-line code like that shown in Figure 5 can be error-prone. Any errors will be caught when compiling the generated test, instead of being able to be identified by the PSS processing tool.

The PSS Procedural Interface (PI) has similarities to the SystemVerilog Direct Procedure Interface (DPI) and foreign-language interfaces provided by other programming languages. In the case of PSS, as well as these other languages, the user specifies a prototype within the 'native' language (eg PSS) that captures the signature of a function implemented in the 'foreign' language (eg C). Doing so allows tools that process PSS descriptions to perform better checking up front. For example, checking that the methods are being called with the correct number and correct type of arguments.

```
function void mem2mem(
       bit[31:0] channel,
       bit[31:0] src,
       bit[31:0] dst,
       bit[31:0] tot_sz,
       bit[31:0] trn_sz);
import target function mem2mem;

extend action wb_dma_c::mem2mem_a {

       exec body {
              mem2mem(channel, dat_i.addr, dat_o.addr, tot_sz, trn_sz);
       }
}
```

Figure 6 - Procedural-Interface Implementation

Figure 6 shows an implementation of the memory-to-memory transfer using the PSS procedural interface. Figure 7 shows the code within the sequence that implements the API. Note that the underlying code is very similar to what we placed in the target-template exec block, but by using the procedural interface we allow the PSS tool to provide better error checking. And, the stimulus can be generated on-the-fly as the simulation runs instead of needing to be pre-generated.

```
task mem2mem(
  bit[31:0]         channel,
  bit[31:0]         src,
  bit[31:0]         dst,
  bit[31:0]         tot_sz,
  bit[31:0]         trn_sz);
    wb_dma_descriptor  desc = wb_dma_descriptor::type_id::create();

    desc.channel = channel;
    desc.mode = 0;
    desc.inc_src = 1;
    desc.inc_dst = 1;
    desc.src_sel = 0;
    desc.dst_sel = 1;
    desc.tot_sz = tot_sz;
    desc.trn_sz = trn_sz;
    desc.chk_sz = 16;
    desc.src_addr = src;
    desc.dst_addr = dst;

    start_item(desc);
    finish_item(desc);

endtask
```

Figure 7 - SystemVerilog implementation of the mem2mem task

Due to the advantages of using the procedural interface, the first recommendation for creating reusable test realization is to define APIs and use the PSS procedural interface to connect the PSS test intent to that test realization.

II. ANATOMY OF TEST REALIZATION

In understanding how best to structure test realization code to maximize its reuse potential, it's important to consider the key elements of test realization. The details of how each of these elements are captured depends a bit on the environment, but they're always present.

A. API

The API might be the most-recognizable aspect of test realization. The test realization API provides access to the functionality of the test realization code.

```
void wb_dma_dev_mem2mem(
             wb_dma_dev_t              *dev,
             uint32_t                  channel,
             uint32_t                  src,
             uint32_t                  dst,
             uint32_t                  sz,
             uint32_t                  trn_sz);
```

Figure 8 - Embedded-Software Test Realization API

Figure 8 shows an example of a C embedded-software API for performing a memory-to-memory transfer using our DMA engine. Note that a handle to the device's context data is passed as the first parameter, and the remaining parameters specify the details about the transfer itself.

```
task mem2mem(
       int unsigned              channel,
       int unsigned              src,
       int unsigned              dst,
       int unsigned              sz,
       int unsigned              trn_sz);
```

Figure 9 - UVM Test Realization API

Figure 9 shows an example of a UVM API for performing a memory-to-memory transfer. Note that in this case the mem2mem task is a class member task, and so the context data is implied (ie it's the current class instance).

B. Configuration and Context Data

Test realization for all but the most-trivial of devices will require some element of context and configuration data. Context data is often found in two forms: state information about the device being managed, and data objects used to interact with the environment.

Configuration data may be as simple as the base address of the device being managed in the case of embedded-software test realization. In the case of UVM test realization, configuration data often consists of one or more UVM objects (eg register model) used to access the device registers.

```
typedef struct wb_dma_dev_s {
	wb_dma_regs_t		*regs;

	// Status flags for state of channels
	uint32_t		status[8];

	// Notification objects for interacting with IRQ
	pvm_event_t		xfer_ev[8];
} wb_dma_dev_t;
```

Figure 10 - Embedded-Software Context Data

Figure 10 shows an example of context and configuration data for an embedded-software version of test realization for the DMA engine. In this case, configuration data is in the form of the *regs* field that holds a pointer to the base address of the DMA device. Context data is in the form of flags to monitor the state of each channel, and objects used for interacting with the interrupt-management system.

```
class wb_dma_dev extends pvm_dev;
	`uvm_object_utils(wb_dma_dev);
	wb_dma_reg_block		m_regs;
	bit				m_active[];
	semaphore			m_sem[];
```

Figure 11 - UVM Test Realization Context Data

Figure 11 shows an example of context and configuration data for a UVM version of test realization for the DMA engine. in this case, configuration data is in the form of the *m_regs* field that is a handle to the UVM register model used to access the DMA device registers. Conext data is in the form of the *m_active* and *m_sem* fields that hold information about the state of DMA channels and allow

C. Events

Here again, test realization for all but the most-trivial of devices will need to respond to events – most often in the form of interrupts. In the case of our DMA engine, our test realization can become aware that a channel has completed a transfer either by continuously polling its channel-status register (CSR) or by receiving an interrupt. Determining channel completion by polling the CSR is functional in an environment that only executes a single transfer on a single DMA engine instance at a time, using interrupts is generally required to do anything more-complex.

```
static void wb_dma_dev_irq(pvm_dev_t *devh) {
	wb_dma_dev_t *dev = (wb_dma_dev_t *)devh;
	uint32_t i;
	uint32_t src_a;

	src_a = pvm_ioread32(&dev->regs->int_src_a);

	// Need to spin through the channels to determine
	// which channel to activate
	for (i=0; i<8; i++) {
		if (src_a & (1 << i)) {
			// Read the CSR to clear the interrupt
			uint32_t csr = pvm_ioread32(&dev->regs->channels[i].csr);
			dev->status[i] = 0;
			pvm_event_signal(&dev->xfer_ev[i]);
		}
	}
}
```

Figure 12 - Embedded-Software Test Realization Event Code

The test realization for our DMA has an interrupt-handler callback function (shown in Figure 12) that is called by the bare-metal software environment. This function is responsible for both clearing the interrupt condition and for notifying the test realization code that the DMA transfer is complete.

```
virtual task irq(int unsigned id);
  uvm_status_e status;
  uvm_reg_data_t value;

  m_regs.int_src_a.read(status, value);
  `uvm_info(get_name(), $sformatf("Received IRQ SRC='h%08h", value), UVM_LOW);

  for (int i=0; i<8; i++) begin
    if (value[i]) begin
      `uvm_info(get_name(), $sformatf("Channel %0d active", i), UVM_LOW);
      if (m_active[i]) begin
        uvm_status_e status_t;
        uvm_reg_data_t value_t;
        wb_dma_ch ch = m_regs.ch[i];

        // Read the CSR to clear the interrupt
        ch.CSR.read(status_t, value_t);
        m_sem[i].put(1);

      end else begin
        `uvm_fatal(get_name(), $sformatf("Interrupt on inactive channel %0d",
i));
      end
    end
  end
endtask
```

Figure 13 - UVM Test Realization Event Code

Figure 13 shows an example UVM test realization task that is triggered when the interrupt signal on the DMA is activated. Note that the behavior is nearly identical to the embedded-software version, except that the UVM register model is used to access registers.

III. Test Realization Reuse Requirements

It is, of course, important to be clear on the requirements for reusable test realization. Three key requirements must be considered when designing a reuse scheme for test realization.

A. Compose Test Realization

The PSS language enables elements of test intent to easily be composed into larger models – for example to combine test intent for several IPs into test intent for a subsystem containing those IPs. It is important that test realization parallel this composability of test intent. Otherwise, the productivity boost obtained by applying portable stimulus is undercut by the cost of assembling the test realization layer.

Test intent must cooperate in order to support composition. This typically means that the code should use a common API to allow each element of test realization to manage concurrency and events. Test realization will often need to wait for events, such as interrupts, and it must be possible to allow other test realization to execute in the meantime.

B. Support Multiple Instances

In addition to needing to support composition of test realization for multiple IPs, it is important to support multiple instances of test realization for the same IP type. In practical terms, this means that each module of test realization must be able to maintain context data. It must also be possible to configure each module of test realization with different resources. For example, in a UVM environment, two instances of test realization will need to be configured with different register-model instances.

C. Support Addressing Instances

Finally, and perhaps most critical from a Portable Stimulus perspective, it must be possible to address the individual elements of test realization from a PSS model. The challenge comes from the fact that the addressing schemes typically used by embedded-software and UVM rely on the client of test-realization having pointers or handles to the modules of test realization code. In contrast, PSS mostly operates on integral quantities.

IV. Test Realization Reuse Best Practices

A. Abstract Up

PSS encourages abstracting up, such that test intent deals with the high level of what is being exercised in the design instead of the details. It is ideal to match the abstraction level of the test realization API to that of the PSS test intent. What this typically means is that the test-realization API is matched to that of the leaf-level actions in the PSS model.

B. Use the Procedural Interface

PSS target-template exec blocks allow literal code in non-procedural languages (eg assembly language) to easily be generated. However, the PSS processing tool is limited in how much checking can be done. Target-template exec blocks also do not support on-the-fly test generation. Procedural-interface exec blocks enable the PSS processing tool to check argument count and parameter types being passed to external methods, and also support both on-the-fly and pre-generated tests. When interfacing to any procedural language, use of the procedural interface is strongly preferred.

C. *Interact via Scalar Values and minimize data exchange*

All foreign-language interfaces impose some level of restriction around the data types to be exchanged, and manner in which data is exchanged with the outside world. PSS is no different, and has similar best practices to SystemVerilog DPI and Java JNI.

It's best to have PSS call the environment and pass data out, while minimizing how much data is returned to the PSS model. In general, pre-generated PSS tests are not reactive, so only passing data from the test intent to the test realization is the safest and most-flexible approach.

Since PSS is a constraint-driven description centered on integral quantities, using an integral quantity to address the test realization instances works well. Using a small integer quantity to identify device instances will be familiar to anyone who has interacted with devices using the Unix file I/O interface [2]. In the case of PSS, using integral quantities to refer to different test-realization instances allows PSS to work with scalar quantities, while providing a mapping to pointers or handles that are friendly to the embedded-software or UVM environment.

The primary challenge in this approach is keeping the IDs used on the PSS side in sync with the context-data pointers or handles on the environment side.

D. *Delegate Whatever Possible to the Environment*

As stated before, test-realization code involves context data, configuration data, and events. To the extent possible, all of these aspects of test realization should be managed within the environment (ie UVM or embedded software), keeping the PSS test realization independent of the implementation details.

An outcome of this best practice is a core assumption that environment initializes itself and the relevant test-realization code before PSS test-generation code executes. This ensures that the PSS test can assume devices are initialized, that interrupts have been configured, and that it can simply call the provided test realization APIs.

V. Test Realization Reuse Framework

The best practices described in the paper can be implemented in many different ways, and with tweaks to existing environments. This section describes key elements of a test realization reuse framework that can either be used as-is or incorporated into other environments. The framework is composed of PSS elements, and elements for each target environment – currently embedded software and UVM.

A. *PSS*

The core data types on the PSS side are deceptively simple. Specifically, a base component that contains an integral field for addressing the appropriate test realization instance.

```
component pvm_dev_c {
      int                devid;

}
```

Figure 14 - Base Component Type

In the case of our DMA engine, our DMA component inherits from this base component and, consequently, contains a device-id field to use in addressing its associated test realization instance.

```
component wb_dma_c : pvm_dev_c {
  import pvm_types_pkg::*;

}
```

Figure 15 - DMA PSS Component

When specifying implementation for DMA actions, the component instance-specific devid value is passed as the first parameter of the test-realization function to uniquely identify the instance.

```
function void wb_dma_dev_mem2mem_d(
	bit[31:0] devid,
	bit[31:0] channel,
	bit[31:0] src,
	bit[31:0] dst,
	bit[31:0] tot_sz,
	bit[31:0] trn_sz);
import target function mem2mem;

extend action wb_dma_c::mem2mem_a {

	exec body {
		wb_dma_dev_mem2mem_d(comp.devid, channel, dat_i.addr,
			dat_o.addr, tot_sz, trn_sz
		);
	}
}
```

Figure 16 - Passing Device ID to Test Realization

B. UVM

UVM is an object-oriented framework implemented in terms of object-oriented languages – primarily SystemVerilog. In a UVM environment, it is most natural to encapsulate content in SystemVerilog classes and reference instances of these classes via handles.

The first framework element on the UVM side is a base class from which a test realization class must inherit. The pvm_dev class is, itself, a UVM component and provides the same mechanisms for propagating configuration data that any other UVM component-derived class does.

```
/**
 * Class: wb_dma_dev
 *
 * Implements PSS test-realization for the wb_dma IP
 */
class wb_dma_dev extends pvm_dev;
	`uvm_object_utils(wb_dma_dev);
	wb_dma_reg_block		m_regs;
	bit				m_active[];
	semaphore			m_sem[];

	function new(string name="wb_dma_dev");
		super.new(name);

		m_active = new[8];
		m_sem = new[8];

		foreach (m_sem[i]) begin
			m_sem[i] = new(0);
		end
	endfunction

// . . .

endclass
```

Figure 17 - UVM Test Realization

Specific API calls are implemented as class methods within the device class, as shown below. These tasks will use environment resources, such as the UVM register model, to interact with the device.

```
task mem2mem(
	int unsigned		channel,
	int unsigned		src,
	int unsigned		dst,
	int unsigned		sz,
	int unsigned		trn_sz);

	`uvm_info(get_name(),
		$sformatf("--> mem2mem channel=%0d src='h%08h dst='h%08h sz=%0d",
		channel, src, dst, sz), UVM_LOW);

	init_single_transfer(channel, 0, src, 1, dst, 1, sz, trn_sz);

	wait_complete_irq(channel);

	`uvm_info(get_name(),
		$sformatf("<-- mem2mem channel=%0d src='h%08h dst='h%08h sz=%0d",
		channel, src, dst, sz), UVM_LOW);
endtask
```

Figure 18 - UVM Implementation of mem2mem

Note that the name of the task shown in Figure 18 is different from the name of the API referenced by the PSS model (Figure 16). Also, note that the API shown above doesn't have a 'devid' parameter. The reuse framework

provides a mechanism for mapping from a global SystemVerilog task call and a class method in a specific instance of a device class.

```
`pvm_dev_task_decl_5(wb_dma_dev, mem2mem, uint32_t, uint32_t, uint32_t,
uint32_t, uint32_t)

`pvm_dev_task_decl_5(wb_dma_dev, mem2dev, uint32_t, uint32_t, uint32_t,
uint32_t, uint32_t)

`pvm_dev_task_decl_5(wb_dma_dev, dev2mem, uint32_t, uint32_t, uint32_t,
uint32_t, uint32_t)
```

Figure 19 - Global Task Declaration Macro

Figure 19 shows the use of macros provided by the reuse framework to declare global tasks that accept a device id as the first parameter, obtain a handle to the appropriate instance of the wb_dma_dev class, and call the appropriate class method. The first macro call (for mem2mem) is roughly equivalent to the code shown in Figure 20.

```
task automatic wb_dma_dev_mem2mem_d(
      uint32_t devid,
      uint32_t p1,
      uint32_t p2,
      uint32_t p3,
      uint32_t p4,
      uint32_t p5);
      wb_dma_dev dev_inst;
      $cast(dev_inst, pvm_get_device(devid));

      dev_inst.mem2mem(p1, p2, p3, p4, p5);
endtask
```

Figure 20 - Generated Global Task

C. Embedded Software

The test-realization reuse framework also provides infrastructure for embedded software. Embedded software is typically written in C, and is often very resource-constrained. Many of the same object-oriented patterns can still be expressed, though.

```
typedef struct wb_dma_dev_s {
      pvm_dev_t                 dev;
      wb_dma_regs_t             *regs;
      uint32_t                  status[8];

      pvm_event_t               xfer_ev[8];
} wb_dma_dev_t;
```

Figure 21 - Embedded SW Implementation of DMA Device

Note that the framework data type (pvm_dev_t) is listed first in the user-defined data structure for the DMA engine. This provides a simple way of implementing inheritance in C. DMA-specific fields are also defined in the core data structure.

```
void wb_dma_dev_mem2mem(
             wb_dma_dev_t       *drv,
             uint32_t           channel,
             uint32_t           src,
             uint32_t           dst,
             uint32_t           sz,
             uint32_t           trn_sz) {
      uint32_t csr, sz_v;

      // Disable the channel
      csr = pvm_ioread32(&drv->regs->channels[channel].csr);
      csr &= ~(1);
      pvm_iowrite32(csr, &drv->regs->channels[channel].csr);

      // Program channel registers
      csr = pvm_ioread32(&drv->regs->channels[channel].csr);

      csr &= ~(1 << 19); // interrupt on chunk done
      csr |= (1 << 18); // interrupt on done
      csr |= (1 << 17); // interrupt on error
      csr &= ~(1 << 5);  // Don't use hardware handshake
      csr |= (1 << 4); // inc src
      csr |= (1 << 3); // inc dst

      csr &= ~(1 << 2); // use interface 0 for source
      csr |= (1 << 1); // use interface 1 for destination

      csr |= (1 << 0); // enable channel

      // Setup source and destination addresses
      pvm_iowrite32((src&0x0FFFFFFF),
                    &drv->regs->channels[channel].src);
      pvm_iowrite32((dst&0x0FFFFFFF),
                    &drv->regs->channels[channel].dst);

// . . .

}
```

Figure 22 - Embedded Software mem2mem Implementation

Figure 22 shows the beginning of an embedded-software implementation of the memory-to-memory transfer operation. Note that a handle to the device data structure provides the context for the function.

```
void wb_dma_dev_mem2mem(
             wb_dma_dev_t              *dev,
             uint32_t                  channel,
             uint32_t                  src,
             uint32_t                  dst,
             uint32_t                  sz,
             uint32_t                  trn_sz);
pvm_devid_api_decl_5(wb_dma_dev, mem2mem,
             uint32_t, uint32_t, uint32_t, uint32_t, uint32_t);
```

Figure 23 - Embedded Software API Registration

The embedded-software implementation of the framework provides macros to define wrapper functions that convert device-id to the appropriate instance of the device data structure, just as the UVM implementation does. In the case of embedded software, this macro declares a static-inline function that will, in most cases, be collapsed to inline code by the compiler. An example of the macro expansion is shown in Figure 24.

```
static inline void wb_dma_dev_mem2mem_d(
             uint32_t                        devid,
             uint32_t                        p1,
             uint32_t                        p2,
             uint32_t                        p3,
             uint32_t                        p4,
             uint32_t                        p5) {
      wb_dma_dev_mem2mem((wb_dma_dev_t *)pvm_get_dev(devid),
                  p1, p2, p3, p4, p5);
}
```

Figure 24 - Expansion of the device-id macro

CONCLUSION

The Accellera PSS language enables test intent to easily be combined, making test intent portable across a range of execution platforms and reusable across a range of levels of verification. None of this is practically possible without modules of test realization that can be similarly composed and reused. Designing test realization code to be well-encapsulated and modular, and easily accessed from PSS enables test intent and test realization to scale in a similar way across platforms and environments, enabling the key results that portable stimulus promises to deliver.

REFERENCES

[1] Accellera Portable Test and Stimulus Specification 1.0, http://accellera.org/downloads/standards/portable-stimulus

[2] File descriptor, https://en.wikipedia.org/wiki/File_descriptor