2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Leveraging the UVM Register Abstraction Layer for Memory Sub-System Verification

Implementing Memory Sequence Reuse Across Multiple Underlying Bus Protocols

Tudor Timisescu, Infineon Technologies, Munich, Germany (*tudor.timisescu@infineon.com*)

Uwe Simm, Cadence Design Systems, Munich, Germany (*uwes@cadence.com*)

***Abstract*—Memory sub-systems are a ubiquitous part of any SoC design. While the mechanics of how to model and stimulate registers are well documented, the topic of memory verification has lagged behind. This paper will demonstrate how, by using the UVM register abstraction layer (UVM_REG), sequences written for one block can be vertically reused. By using the "frontdoor" mechanism to convert from abstract memory operations to physical bus transfers, the same memory sequences can be plugged into multiple verification environments.**

Keywords—Universal Verification Methodology; UVM; register; memory; UVM_REG; frontdoor; vertical reuse

I. Introduction

With verification taking up more and more of project schedules, there is no way for teams to reinvent the wheel with each new product they validate. Reuse has become the corner stone that enables better productivity and higher quality. After a few iterations of vendor specific verification methodologies (like AVM, URM and VMM) and cross-vendor efforts (OVM), the industry has finally settled on the Universal Verification Methodology (UVM) as the first truly vendor-agnostic set of guidelines and best practices. While the UVM is available in multiple languages, the *SystemVerilog* implementation is the one that enjoys the most widespread use and will be the focus of this paper.

As registers and memories are ubiquitous in semiconductor design, they also play a central role in verification. Register packages were developed for all major verification libraries to abstract operations done on registers and memories. Thinking in terms of such abstract operations increases the potential of reuse from block level to subsystem and system level. Such a register package was notably missing from OVM, but the gap was filled by third party packages from Mentor Graphics and Cadence. A standardized register packaged was added to UVM, its implementation being based on the VMM RAL, and was affectionately named UVM REG.

The tasks of modeling and stimulating registers are rather well understood. There are excellent papers that explore the topic, such as [1] and [2]. There aren't many resources dealing with how to work with memories though. This paper will try to fill a small part of that gap.

II. Problem Statement

A. *Memory sub-system verification*

A topic that comes up in any System-on-Chip (SoC) design is that of memory sub-system verification. The memory sub-system could be composed of multiple smaller blocks that span multiple levels of hierarchy, each of them being verified with their own testbench. These block-level testbenches are usually reused vertically as part of the sub-system verification environment.

Figure 1 shows an example of a simple memory sub-system. It is centered on a cache module with a proprietary bus interface that can process both data and instructions. This cache is integrated together two protocol converters, one for each stream, to form a memory sub-system that can be used with an AHB based processor.

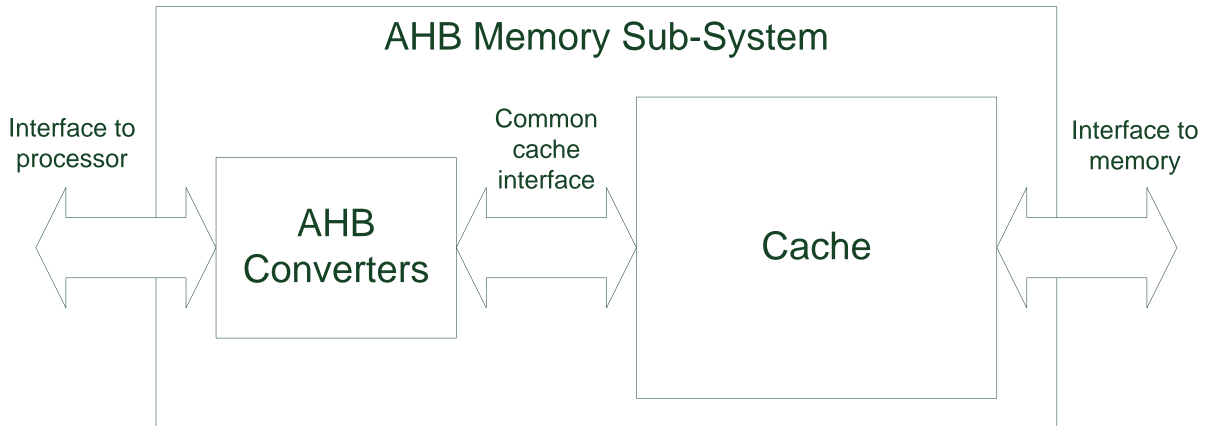

Figure 1 AHB Memory Sub-System block diagram

The cache communicates on the processor side via a Common Cache Interface (CCI), architected for maximum throughput. The block’s reasonably high complexity warrants its own standalone verification environment. With the proper set of protocol converters, this cache can be used alongside any processor core. To allow it to work with an ARM core, for example, AHB-to-CCI converters are required. The sub-system testbench can leverage the cache verification effort by instantiating the block-level testbench, providing it with a reference model and checks.

B. Vertical reuse

The mechanics of vertical reuse for testbenches are pretty well understood, but the topic of reusing stimulus from lower levels to higher ones is a hot topic at the time of writing. Accellera has started the Portable Stimulus Working Group to explore new standards and tools that can do this.

The type of stimulus that both the cache and the memory sub-system are expected to consume is conceptually similar. From a simplistic point of view, both are supposed to handle sequences of memory accesses. There are two seemingly contradicting requirements, however:

1. it is required to stress the bus protocol to be able to fully verify the specific DUT
2. it is necessary to write abstract sequences that can be reused on subsystem level

This paper will explore how to use the UVM register package to achieve both of these requirements when accessing memories.

III. Proposed Solution

It is noted in [3] that one of the main objectives of the UVM register layer is to allow "*the migration of verification environments and tests from block to system levels without any modifications*". This is done by decoupling physical accesses (in this case CCI or AHB transfers) from the logical operations they represent (memory accesses). What is required is a mechanism to convert from abstract memory operations to bus accesses.

A. UVM Register Layer basics

Writing sequences in terms of registers and memories presents a number of advantages. From a maintainability point of view, it makes it easier to change the address map layout, because all information about it is stored in one place instead of being distributed throughout multiple bus sequences as magic numbers. It also makes it possible to more quickly change between bus protocols, for example for new iterations of the product. This way of writing sequences is also closer to the way an embedded software programmer thinks. Moving upward in abstraction reduces the amount of code the test writer needs to write to achieve the desired outcome:

```
task body();
  uvm_status_e status;
  model.MEM_CTRL.ENABLE.set(1);
  model.MEM_CTRL.update(status);
endtask
```

Code sample 1 Example register write

Converting from register accesses is typically done using a *uvm_reg_adapter*. Such an adapter can convert between abstract register accesses and concrete bus transfers. The same mechanism can be used for regular memory read/writes. Regular in this case means accesses that are smaller or equal to the underlying bus width.

```
class ifx_ahb_reg_adapter extends uvm_reg_adapter;
  // …

  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    ifx_ahb_seq_item item = ifx_ahb_seq_item::type_id::create("item");
    if (!item.randomize() with {
      item.burst_kind == SINGLE;
      item.size == (rw.n_bits == 8 ? BYTE : (rw.n_bits == 16 ? HALFWORD : WORD));
      item.direction == rw.kind == UVM_WRITE ? WRITE : READ;
      item.addr == rw.addr;
      item.data == rw.data;
    })
      `uvm_error("RANDERR", "Randomization error")
    return item;
  endfunction
endclass
```

Code sample 2 AHB adapter

A register adapter for a certain protocol is reusable and could be delivered as part of that protocol's UVC.

B. Burst accesses

A memory burst access represents a stream of bytes that are to be transferred. The number of bytes can potentially be greater than the bus width. Bursts are started using the *burst_read(...)* and *burst_write(...)* tasks:

```
class access_mem_seq extends uvm_reg_sequence;
  // ...

  reg_model model;

  virtual task body();
    uvm_status_e status;
    uvm_reg_data_t data[] = new[8];
    model.mem.burst_read(status, 'h40, data, .parent(this));
  endtask
endclass
```

Code sample 3 Memory read burst

Modern bus protocols allow for different ways to move data. For AHB transactions, the bus master can specify the number of data words to send within a bus access and the width of those words ([4]). For example, a read of 32 bytes can be executed using different sequences of AHB transactions: 32 SINGLE BYTE accesses, 8 SINGLE WORD accesses, 2 INCR4 WORD accesses, combinations of the previous and more. A device must be able to cope with any of these combinations to say that it is AHB-compliant. The call to *burst_read(...)* from Code sample 3 encapsulates any of the sequences described above. Executing such a simple memory operation in multiple tests by varying the types of AHB transfers used to perform it has the potential to fill a significant amount of the protocol coverage.

The task of converting from the memory burst to the physical bus accesses required to perform it is left to the UVM register layer. For burst accesses, the register map through which the memory is accessed will chop the burst into multiple single accesses and process them through the register adapter, a process that is illustrated in Figure 2. The algorithm implements this is fixed, which isn't very good for controllability, since it means that only a subset of the possible scenarios allowed by the protocol can be reached. The UVM developers had this limitation in mind and provided a mechanism to override the default behavior: *uvm_reg_frontdoor*.

A frontdoor allows the verification engineer to customize how a memory operation is split into bus transfers. The frontdoor mechanism could also be used to access special types of registers (e.g. indexed registers), but this is out of the scope of this paper.

Figure 3 Frontdoor operation shows a conceptual picture of how the frontdoor mechanism works. The memory sequence started by the user accesses the memory through calls to *burst_read(...)* or *burst_write(...)*. These memory operations go through the address map which starts the frontdoor sequence on its associated bus sequencers.

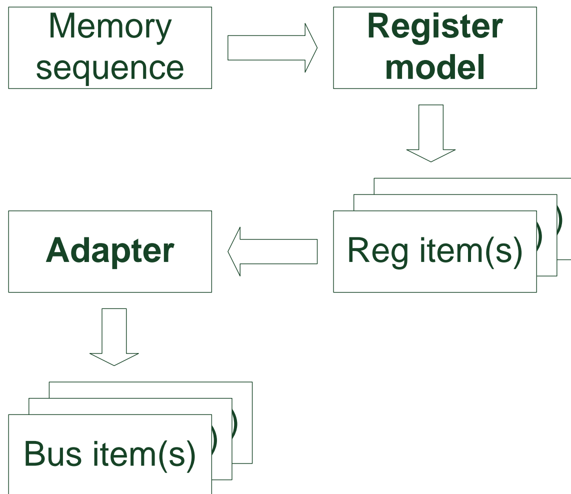

Figure 2 Adapter operation

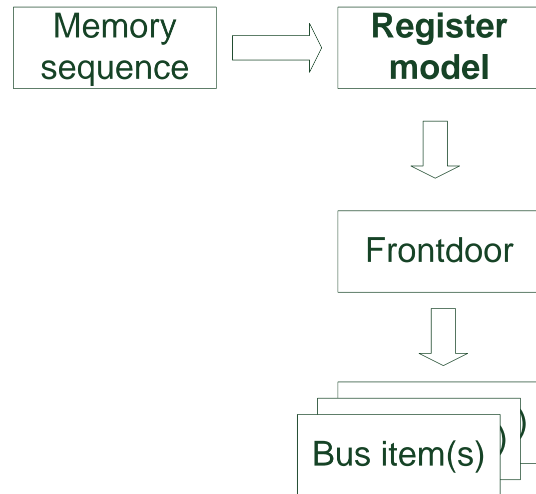

Figure 3 Frontdoor operation

C. Defining custom frontdoors

A frontdoor is nothing more than a *uvm_sequence* that contains a reference to the operation it is supposed to translate. It is essentially the glue that binds the register model to the protocol UVC. A frontdoor is associated with a specific memory by passing it in as a parameter to its *set_frontdoor(...)* function:

```
ifx_ahb_mem_frontdoor frontdoor = ifx_ahb_mem_frontdoor::type_id::create("frontdoor");
model.mem.set_frontdoor(frontdoor, model.default_map);
```

Code sample 4 Setting the frontdoor

The developer needs to implement the *body()* task to define how the operation is converted into bus accesses. Constrained randomization can help fill in the blanks on exactly which bus transfers to execute. Code sample 5 contains an example of a simple frontdoor for the CCI protocol.

```
class ifx_cci_mem_frontdoor extends uvm_reg_frontdoor;
  // ...

  virtual task body();
    ifx_cci_seq_item items[] = new[rw_info.value.size()];
    foreach (items[i]) begin
      `uvm_create(items[i])

      if (!items[i].randomize() with {
        direction == rw_info.kind inside { UVM_READ, UVM_BURST_READ } ? READ : WRITE;
        addr == rw_info.offset + i * 4;
      })
        `uvm_fatal("RANDERR", "Randomization error")

      start_item(items[i]);
      finish_item(items[i]);
    end
  endtask
endclass
```

Code sample 5 CCI frontdoor

The AHB memory sub-system testbench would contain an instance to the same register model. The same memory sequence could be reused here by passing it through a different frontdoor that can start AHB transactions on an AHB bus sequencer. Code sample 6 shows a possible implementation of an AHB frontdoor. The flexibility of using this mechanism doesn't come without a cost. In comparison to the adapter presented in Code sample 2, it can be seen that the frontdoor sequence is more complicated, but just as the former was reusable, so is the latter. An AHB UVC could come pre-packaged with a frontdoor, thereby making the task of writing it a one-off effort.

```
class ifx_ahb_mem_frontdoor extends uvm_reg_frontdoor;
  // ...

  virtual task body();
    bit[31:0] offset = rw_info.offset;
    int unsigned num_bytes = rw_info.value.size() * 4;
    while (num_bytes) begin
      ifx_ahb_seq_item burst = get_next_burst(offset, num_bytes);
      int unsigned burst_num_bytes = (2 ** burst.size) * burst.burst_len;
      start_item(burst);
      finish_item(burst);
      num_bytes -= burst_num_bytes;
      offset -= burst_num_bytes;
    end
  endtask

  protected function ifx_ahb_seq_item get_next_burst(bit[31:0] offset, int unsigned num_bytes);
    ifx_ahb_seq_item burst;
    `uvm_create(burst);
    if (!burst.randomize() with {
      addr == offset;
      direction == rw_info.kind inside { UVM_READ, UVM_BURST_READ } ? READ : WRITE;
      size inside { BYTE, HALFWORD, WORD };
      burst_kind inside { SINGLE, INCR4, INCR8 };
      (2 ** size) * burst_len <= num_bytes;
    })
      `uvm_fatal("RANDERR", "Randomization error")
    return burst;
  endfunction
endclass
```

Code sample 6 AHB frontdoor

Complex sequences that only make use of memory operations (calls to *burst_read(…)* and *burst_write(…)*) could be effortlessly reused, because there aren't any protocol aspects mixed into them. The entire set of test sequences developed when verifying the cache at the block level can be used to verify the AHB memory sub-system.

D. Extending frontdoors

Another benefit of being able to switch out frontdoors is that it allows randomization to close in on interesting protocol corner cases. This can be done by adding constraints in a sub-class. The following code sample shows an AHB frontdoor that will convert any burst into a sequence of SINGLE WORD back-to-back transfers:

```
class ahb_frontdoor_single extends ifx_ahb_mem_frontdoor;
  // ...

  virtual task body();
    ifx_ahb_seq_item items[] = new[rw_info.value.size()];
    foreach (items[i]) begin
      `uvm_create(items[i])
      if (!items[i].randomize() with {
        direction == rw_info.kind inside { UVM_READ, UVM_BURST_READ } ?
          ifx_ahb_pkg::READ : ifx_ahb_pkg::WRITE;
        addr == rw_info.offset + i * 4;
        burst_kind == SINGLE;
        size == WORD;
      })
        `uvm_fatal("RANDERR", "Randomization error")
      start_item(items[i]);
      finish_item(items[i]);
    end
  endtask
endclass
```

Code sample 7 Reduced AHB frontdoor

The UVM factory allows a test to switch out the original frontdoor with its subclass by applying a factory override:

```
class test_ahb_frontdoor_single extends test_ahb_frontdoor;
  // ...

  virtual function void build_phase(uvm_phase phase);
    factory.set_type_override_by_type(ifx_ahb_mem_frontdoor::get_type(),
      ahb_frontdoor_single::get_type());
    super.build_phase(phase);
  endfunction
endclass
```

Code sample 8 Factory override to replace frontdoor

The same effect of generating only SINGLE WORD transfers can also be achieved by adding extra constraints in the AHB item itself:

```
class single_word_ahb_seq_item extends ifx_ahb_seq_item;
  // ...

  constraint single_word {
    burst_kind == SINGLE;
    size == WORD;
  }
endclass
```

Code sample 9 Constrained AHB item

As was shown for the frontdoor, the factory is used to set a type override to the new sequence item:

```
class test_ahb_frontdoor_single2 extends test_ahb_frontdoor;
  // ...

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    ifx_ahb_seq_item::type_id::set_type_override(single_word_ahb_seq_item::get_type());
  endfunction
endclass
```

Code sample 10 Factory override to replace AHB item

This possibility to easily layer constraints enables a "write once, tweak everywhere" approach to test writing. Fewer memory sequences are needed because it is possible to vary what bus protocol combinations tests are exercising by tweaking the frontdoor and the underlying bus sequence item.

E. Achieving protocol coverage closure

While constrained randomization is powerful in covering a wide range of the state space, it probably won't reach all corner cases. Some tests will still need to be targeted to the actual bus protocol, by writing stimulus at the bus transfer level. The number of such tests that cannot be reused would be considerably reduced, though.

IV. FUTURE WORK

The current implementation of the register layer doesn't make it easy to access locations outside of the legal address map. This is a key part of error injection testing, because it is essential to ensure that the DUT can fail gracefully in case of unexpected scenarios. Further investigation of this topic is necessary.

While developing this approach, the author also tried to implement checking around the register layer, to limit the impact that changes in the underlying bus protocols would have on the verification effort. This is currently not completely possible due to some inconsistencies inside the UVM BCL. One notable example is the handling of bus errors, where *uvm_reg_item* provides a *status* field. The intention of this field was probably to deliver information on whether the access succeeded or not, but it gets overwritten inside the register predictor. Once such inconsistencies are sorted out it would be possible to shift more of the checking to rely on the register layer.

V. CONCLUSION

The paper proposed a way of leveraging the UVM register layer's frontdoor mechanism to write abstract sequences that can be reused from the block level to the sub-system level. Keeping the stimulus agnostic about bus protocols also allows for interfaces of the DUT to be swapped more easily in new product variants. Using constrained randomization keeps the translation to raw bus transfers customizable, allowing tests to cover a wide area of the protocol state space.

REFERENCES

[1] S. Holloway, "The UVM Register Layer – Introduction, Experiences and Recipes," DVClub 2012.

[2] M. Litterick, M. Harnisch, "Advanced UVM Register Modeling – There's More Than One Way to Skin a Reg," DVCon 2014.

[3] Accellera, "UVM User Guide, v1.1," www.uvmworld.org.

[4] ARM, "AMBA 3 AHB-Lite Protocol Specification", ARM IHI 0033A.