# BOOSTING SIMULATION PERFORMANCE OF UVM REGISTERS IN HIGH PERFORMANCE SYSTEMS

Ahmed Yehia
Mentor Graphics Corp.
Cairo, Egypt
ahmed_yehia@mentor.com

## ABSTRACT

Registers and memory blocks are key parts of any design; a typical design could hold hundreds of them. Verifying the behavior of registers and memory blocks is an essential part in the verification process. There are many techniques for modeling and verifying hardware registers and memory blocks. In this paper, we focus on verifying hardware registers using register packages.

Verification and modeling of hardware registers and memory blocks via register packages is not a new trend. Many register packages from various vendors, written in different languages, currently exist and used in the industry. The Accellera VIP-TSC committee has made a significant progress in releasing the Universal Verification Methodology (UVM) [1] defining standards for creation, integration, and extension of verification environments. The UVM register library is an open source library, being part of the UVM library, which allows modeling and verification of hardware registers and memory blocks. Yet, the way the UVM register library is currently designed to layer registers and memories transactions on top of bus transactions, well suits low speed buses. On the other hand, it may not be efficient for high performance buses introducing undesired simulation performance degradation.

In the paper, we give a quick overview of the UVM register library on how it could be used to model and verify hardware registers and memory blocks, showing the simulation performance bottlenecks observed when performing on high-speed buses. We then present an efficient overlay layer that can be easily integrated on top of the UVM register library, making the library suitable for high as well as low performance systems. Then we show how an efficient, yet powerful, registers to AMBA AXI bus transactions adapter would look like in this case. Finally, we provide a cost-benefit analysis on current and proposed implementations.

## 1. INTRODUCTION

Naming some of the UVM register library features:

- Address management.
- Modeling registers and memory blocks.
- Front door and back-door access to Device under Verification (DUV).
- Implicit and explicit prediction of registers and memory blocks values.
- Coverage model API.

Usually the integration of a register library in a testbench environment requires four abstract steps: (1) building the register database, (2) writing registers and memory blocks test sequences, (3) configuring registers coverage as needed, and (4) integrating registers models and test sequences to the testbench verification components.

### 1.1 Building the Register Database

Although register models could be built-up manually, typically register models are automatically generated using register model generators, which prevent manual coding errors. There are a number of commercial UVM register generator tools that can capture register specification from spreadsheet, IP-XACT, and XML inputs. Below we provide a quick overview of the different UVM classes used to build your register database.

#### *1.1.1 Field*

A group of bits providing specific functionality in a hardware register. It is modeled in the UVM register library using the *uvm_reg_field* class, and configured using the *uvm_reg_field::configure()* method.

#### *1.1.2 Register*

A hardware register model grouping fields at different offsets within the register. It is modeled in the UVM register library by extending the *uvm_reg* base class adding *rand* objects of *uvm_reg_field* type, and configured using the *uvm_reg::configure()* method.

#### *1.1.3 Memory*

A memory block with well-defined address range. It is modeled in the UVM register library by extending the *uvm_mem* base class defining the memory block specifications inside the constructor *new()*, and configured using the *uvm_mem::configure()* method.

#### *1.1.4 Block*

Groups registers, memories and sub-blocks. It is modeled in the UVM register library by extending the *uvm_reg_block* base class, then instantiating and configuring registers, memories and sub-blocks inside its *build()* method.

#### *1.1.5 Map*

Locates the address offset of registers, memories and sub-blocks within a block. It is modeled in the UVM register library by instantiating an object of *uvm_reg_map* class in a block. Registers and memories are added to the address map using *uvm_reg_map::add_reg()* and *uvm_reg_map::add_mem()* respectively.

### 1.2 Writing Register and Memory Sequences

Usually a verification engineer would like to access registers and memories writing or reading their contents. The UVM register library provides *write()*, *read()*, *burst_write()* and *burst_read()* APIs for accessing registers and memories. These methods get called in the sequence *body()* method reflecting the test scenario of a user. The map will locate the address of the register (or the memory) being accessed, then handover the register transaction to an adapter which will convert them to a corresponding transaction of the bus lying underneath; this is what is called the front-door access. The advantage of this flow, is that registers and memories test sequences can be written in a generic way independent of the bus architecture lying underneath. This enables verification environment reuse and portability. The UVM package comes with a library of automatic sequences doing basic registers and memories tests that can be reused when necessary.

### 1.3 Configuring Registers Coverage

The UVM register library does not come with coverage models for registers; however, they provide the necessary API to control the instantiation and sampling of coverage models built by the user. Coverage models will not be covered by this paper as they are out of its scope, please refer to the registers section in the UVM user manual for more details[1].

### 1.4 Integrating UVM Registers in the testbench environment

Typically, UVM register models are integrated in the testbench environment by doing the following steps:

- Build *register database* by constructing the register blocks in the test and pass their handles to testbench components via configuration objects.
- Build a register *adaption layer*; a component to translate register transactions to bus transactions and vice versa. This can be achieved by extending the *uvm_reg_adapter* base class and providing an implementation for *reg2bus()* and *bus2reg()* methods.
- Construct the register adapter object in the testbench environment and connect it, as well as the agent sequencer, to the register map via the *set_sequencer()* method.
- Build a *predictor* component acting as a listener on the bus by extending the *uvm_reg_predictor* class, implementing its *write()* method. The predictor is used to convert bus transactions to register transactions then update the corresponding register model, or if desired compare the register model value to the actual hardware register value.
- Construct the *predictor* object in the testbench environment, and connect it to the bus agent monitor analysis port using normal UVM Transaction Level Modeling (TLM) analysis port connections.

The following figure represents a verification environment with UVM registers integrated[4].

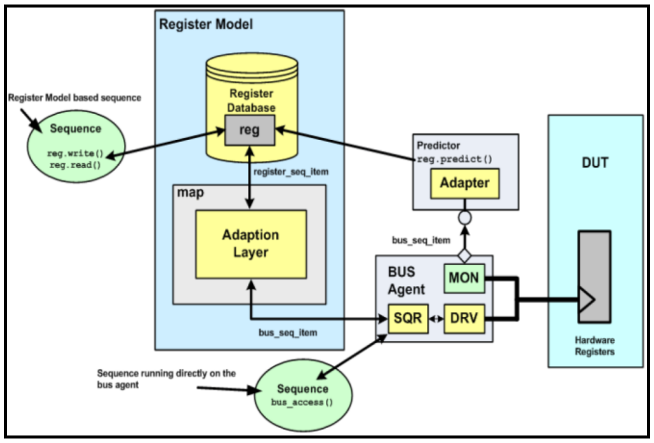

Figure 1. Verification Environment with UVM Registers Integrated

## 2. CURRNET UVM REGISTERS FRONT-DOOR ACCESS IMPLEMENTATION

The way the UVM register library front-door access is currently designed is illustrated as follows: (1) a field, register or memory *write()*/*read()* call shall invoke a field, register or memory *do_write()*/*do_read()*, (2) a field, register or memory *do_write()*/*do_read()* will invoke a map *do_write()*/do_read(), (3) a map *do_write()*/*do_read()* will invoke a map *do_bus_write()*/*do_bus_read()*, which will invoke the adapter *reg2bus()* for converting a register transaction to the corresponding bus transaction to be executed afterwards.

The current SystemVerilog [5] implementation of the map *do_bus_write()* (and similarly *do_bus_read()*) method in the UVM register library is as follows:

```
task uvm_reg_map::do_bus_write (uvm_reg_item rw,
     uvm_sequencer_base sequencer,
     uvm_reg_adapter adapter);
  ...
  //Get bus and register/field/memory information
  get_bus_info(rw, map_info, n_bits, lsb, skip);
  //Extract addresses from the map_info
  addrs = map_info.addr;
  ...
  //Loop over data values array in register trans
  foreach (rw.value[val_idx]) begin
    //Calculate byte enables in case of UVM Field
    if (rw.element_kind == UVM_FIELD) begin
      ...
    end
    ...
    //For each address location
    foreach (addrs[i]) begin
      uvm_sequence_item bus_req;
      uvm_reg_bus_op rw_access;
      uvm_reg_data_logic_t data =
                   (value >> (curr_byte*8)) &
                   ((1'b1 << (bus_width * 8))-1);
      //Update rw_access struct
      //In case of UVM Field update byte enable
      if (rw.element_kind == UVM_FIELD)
        for (int z=0;z<bus_width;z++)
          rw_access.byte_en[z] =
                   byte_en[curr_byte+z];
      rw_access.kind    = rw.kind;
      rw_access.addr    = addrs[i];
      rw_access.data    = data;
      rw_access.byte_en = byte_en;
      rw_access.n_bits  = (n_bits > bus_width*8)
                        ? bus_width*8 : n_bits;
      ...
      //Convert the register item transaction to
      //the bus transaction lying underneath
      bus_req = adapter.reg2bus(rw_access);
      ...
      //Drive transaction
      bus_req.set_sequencer(sequencer);
      rw.parent.start_item(bus_req, rw.prior);
      ...
      rw.parent.finish_item(bus_req);
      ...
    end //foreach (addrs[i])
    ...
  end //foreach (rw.value[val_idx])
  ...
endtask //do_bus_write()
```

The method first captures the information of the item[1] accessing it, i.e. address(es) of the item, number of bits to update, address offset and data. It captures some additional information in case of a field access. It also captures the corresponding bus width.

The dynamic array *value*, holds the write data in case of a *write()*, or to hold the read data in case of *read()*. In case of *read()*/*write()*, *value* will be a single element array. In case of *burst_read()*/*burst_write()*, i.e. item is a memory, *value* will hold multiple elements. The queue *addrs*, is a queue holding the address(es) of an item. The number of elements in *addrs* depends on the item's width with respect to the bus width. If the item's width is smaller than or equal to the bus width, the *addrs* queue will hold single element, otherwise the number of elements in the array will be equal to the item's width divided by the bus width.

As shown in the above implementation, the *do_bus_write()* will loop over *value* elements. Then in each iteration, loop over each item addresses. In each inner iteration, the method constructs a register transaction to be converted to bus transaction via the adapter's *reg2bus()* method. This methodology may be suitable for low performance buses; however, it could be inefficient for high performance buses not utilizing bus powerful features lying underneath. Imagine the scenario where one is operating on an AMBA AXI [6] bus and wants to write to a 2KB memory block; the current implementation will send 512 different bursts on an AMBA AXI bus of 32-bit width, this looks like an inefficient way to operate on an AMBA AXI. Instead, 32 bursts (16 beats each) could be sent, or even a single burst of 512 beats if your system permits extended burst length. Each time you send an extra burst on AMBA AXI bus, you lose at least two cycles in the case of a *write* and one cycle in the case of a *read*[2]. Maximizing the number of bursts maximizes context switching in simulation, which may have severe consequences on simulation performance imagining a test performing hundreds of these operations.

[1] An item is typically a field, register or a memory location.
[2] Analysis assuming simple AMBA AXI where data phase follows address phase by at least one clock cycle.

## 3. ALTERNATIVE IMPLEMENTATION TO CURRENT UVM REGISTERS FRONT-DOOR ACCESS

As stated previously the current *do_bus_write()* implementation creates an undesired bottleneck when performing on high performance buses. In the following subsections we represent an efficient alternative to do*_bus_write()* making it suitable for high performance buses, then we show how a powerful implementation of *reg2bus()* and *bus2reg()* methods shall look like in this case for an AMBA AXI bus.

### 3.1 Efficient Alternative to Current *do_bus_write()* Implementation for High Speed Buses

```
task uvm_reg_map::do_bus_write (uvm_reg_item rw,
     uvm_sequencer_base sequencer,
     uvm_reg_adapter adapter);
  ...
  //Used to share data between
  //reg2bus() & do_bus_write()
  uvm_reg_bus_op_c uvm_reg_bus_op_c_write = new();
  //Get bus and register/field/memory information
  get_bus_info(rw, map_info, n_bits, lsb, skip);
  //Extract addresses from the map_info
  addrs = map_info.addr;
  //Calculate byte enables in case of UVM Field
  if (rw.element_kind == UVM_FIELD) begin
    ...
  end
  //Total number of bits to be sent for all items
  n_bits_total      = n_bits*rw.value.size();
  //Capture the start address
  start_address     = addrs[0];
  //Update rw_access struct
  rw_access.byte_en = byte_en;
  rw_access.kind    = rw.kind;
  rw_access.n_bits  = n_bits_total;
  //Sync mechanism for concurrent writes
  uvm_reg_bus_op_c:: reg2bus_write_sm.get(1);
  //Extract bursts when more bits to write
  while (rw_access.n_bits > 0) begin
    rw_access.addr = start_address;
    //Pass rw to adapter
    adapter.m_set_item(rw);
    //Convert the register item transaction to the
    //bus transaction lying underneath
    bus_req        = adapter.reg2bus(rw_access);
    adapter.m_set_item(null);
    //Push the bus_req to the bus requests queue
    //Workaround adapter.reg2bus() returning only
    //one sequence item limitation
    bus_req_q.push_back (bus_req);
    //Get info about consumed bits by adapter
    //bus2reg(). Workaround rw_access passed as
    //const in bus2reg() prototype
    uvm_config_db #(uvm_reg_bus_op_c)::get(null,
                       "", "rw_acc_adapt_write",
                       uvm_reg_bus_op_c_write);
    //update start address of next iteration
    //reflecting num of bits converted to bus
    //transactions
    start_address += uvm_reg_bus_op_c_write.n_bits
                     /(8*get_addr_unit_bytes());
    rw_access.n_bits -=
                    uvm_reg_bus_op_c_write.n_bits;
  end //while (rw_access.n_bits > 0)
  //Free the semaphore
  uvm_reg_bus_op_c:: reg2bus_write_sm.put(1);
  //Drive all bus transactions
  foreach (bus_req_q[i]) begin
    uvm_sequence_item bus_req = bus_req_q[i];
    bus_req.set_sequencer(sequencer);
    rw.parent.start_item(bus_req,rw.prior);
    ...
    rw.parent.finish_item(bus_req);
    ...
  end
endtask //do_bus_write()
```

The proposed implementation avoids chopping data in bus width chunks and lets the adapter's *reg2bus()* method decide the amount of data it is going to put in a bus transaction according to its bus capabilities.

We start doing the same regular tasks getting item and bus information. The total amount of bits to be accessed and the start address were then marked and passed to the register item transaction. We keep on looping conditionally until there are no more bits to access. In each iteration, the adapter *reg2bus()* would consume as many bits as it could from the register item transaction according to its bus capabilities, converting the register item transaction to a suitable transaction for the bus lying underneath. The *reg2bus()* then notifies the *do_bus_write()* with the amount of bits it consumed via the sync object *uvm_reg_bus_op_c_write*, which the *do_bus_write()* gets a handle of via the resources DB *uvm_config_db*[3].

The *uvm_reg_bus_op_c* is a very simple class used for sharing data between the *reg2bus()* adapter method and the

[3] Resource DB was introduced in UVM for data sharing. Please refer to the UVM user manual for more information.

*do_bus_write()*. The reason it is needed, is because only the *reg2bus()* knows how many bits it consumed, however the info is required by the *do_bus_write()* to know if it needs to re-iterate. The problem could have gone away if *reg2bus()* returns an array of bus transactions, however the current method prototype in the UVM library returns only one bus transaction. The hereby proposed implementation aims to be an overlay layer on top of the UVM package; no edits in UVM source are required for it to function properly, thus the proposed means for sharing data.

Since the overhead of accessing the configuration database for every bus transaction could be tangible especially for low speed buses, a better solution is to extend the UVM register library to support a register-to-bus method that returns an array of transactions. Appendix B demonstrates the extended enhancements required in the UVM register library to leverage from a register to bus *reg2bus_arr()* method that returns an array of bus transactions for handling high, as well as low, performance buses while maintaining backward compatibility with current implementation.

The implementation of *uvm_reg_bus_op_c* class is as follows:

```
class uvm_reg_bus_op_c;
  static semaphore reg2bus_read_sm;
  static semaphore reg2bus_write_sm;
  static semaphore bus2reg_read_sm;
  int n_bits;
  // Function: new
  //
  // create a new instance
  // typically it creates instances of the static
  // semaphores only once
  function new();
    if (reg2bus_write_sm == null)
      reg2bus_write_sm = new(1);
    if (reg2bus_read_sm == null)
      reg2bus_read_sm = new(1);
    if (bus2reg_read_sm == null)
      bus2reg_read_sm = new(1);
  endfunction
endclass
```

All what needs to be done is to extend the *uvm_reg_map* class and implement the *do_bus_write()* and *do_bus_read()* methods as shown above. Implementation of *do_bus_read()* would be similar to *do_bus_write()* but may be little trickier -more details in Appendix A. In the test (or when building your register block), just override *uvm_reg_map* with your new register map class name using the *set_type_override()*[4] method.

```
class my_reg_block extends uvm_reg_block;
  virtual function void build();
    uvm_reg_map::type_id::set_type_override(
                   my_reg_map::get_type(),1);
    ...
  endfunction
endclass
```

## 3.2 Efficient implementation of reg2bus() and bus2reg() for an AMBA AXI bus

### 3.2.1 *reg2bus() implementation for an AMBA AXI bus*

Typically *reg2bus()* will be working on high level bus transactions, e.g. write/read bursts in case of an AMBA AXI protocol.

A. <u>Capture the required info for optimal burst type selection</u>

```
virtual function uvm_sequence_item reg2bus
         (const ref uvm_reg_bus_op rw);
  ...
  //get the uvm_reg_item, data, burst address,
  //map, and bus width
  rw_reg_item   = get_item();
  reg_data_q    = rw_reg_item.value;
  burst_addr    = rw.addr;
  map           = rw_reg_item.local_map;
  bus_width     = map.get_n_bytes();
  //Maximum number of bytes in one AXI burst is 4K
  bytes_to_send = ((((rw.n_bits-1)/8) + 1) < 4096)
                  ?(((rw.n_bits-1)/8) + 1) : 4096;
  ...
```

In the code below, we capture different scenarios: (1) field access, (2) register or memory access with unaligned address or number of bytes smaller than bus width, and (3) register or memory access with number of bytes greater than or equal to bus width.

B. <u>Field access accomodating for fields wider than bus width by sending INCR bursts</u>

```
  if (&rw.byte_en == 0) begin
    for (int curr_byte=0;
         curr_byte < `UVM_REG_BYTENABLE_WIDTH;
         curr_byte += bus_width)
    begin
      int ones = 0;
      byte_en_chunk = (rw.byte_en >> curr_byte) &
```

[4] The set_type_override() method is used to override specific types with others using the OOP factory terminology.

```
                ((1'b1 << bus_width)-1);
    for (int i=0; i< bus_width; i++)
      if (byte_en_chunk[i] == 1)
        ones += 1;
    if (ones > 0) begin
      bytes_sent   += ones;
      burst_length += 1;
    end
  end
  //Burst Size is either bus width
  //or register width
  rg = field.get_parent();
  parent_reg_bytes = (rg.get_n_bits()-1)/8 + 1;
  burst_size = (parent_reg_bytes > bus_width)?
              bus_width : parent_reg_bytes;
  if (rw_reg_item.kind == UVM_WRITE) begin
    for (int curr_byte=0;
         curr_byte < (burst_size*burst_length);
         curr_byte += burst_size) begin
      reg_data_q[0] = reg_data_q[0] >>
                      (curr_byte*8);
      axi_data_q.push_back(reg_data_q[0]);
    end
    void'(reg_data_q.pop_front());
  end
end
```

C. Unaligned address, or bytes to send smaller than bus width: Fixed burst with single beat

```
else begin
  for(int i=2; i<= bus_width;i=i*2)
    if ((byte_addr%i==i/2)||
        (bytes_to_send<i)) begin
      burst_length = 1;
      burst_size   = i/2;
      bytes_sent   = i/2;
      //Flag a transformation complete
      transform_complete = 1;
      if ((rw_reg_item.kind == UVM_WRITE) ||
          (rw_reg_item.kind == UVM_BURST_WRITE))
      begin
        axi_data_q.push_back(reg_data_q[0] &
                ((1'b1 << (burst_size * 8))-1));
        reg_data_q[0] = reg_data_q[0] >>
                        (burst_size*8);
      end
      break;
    end
  end
```

D. Accessing register or memory with number of bytes greater than or equal to bus width.

```
if (transform_complete == 0) begin
//Maximum AXI burst length is 16, or
//4K/Bus-width in case of extended burst length
  int max_burst_length;
  if (axi_master_cfg.extended_burst_enabled)
    max_burst_length = 4096/bus_width;
  else
    max_burst_length = 16;
    burst_length=((bytes_to_send -1) /
     bus_width + 1) < max_burst_length ?
     ((bytes_to_send - 1)/bus_width + 1):
     max_burst_length;
  burst_size = bus_width;
  bytes_sent = burst_length * bus_width;
  if ((rw_reg_item.kind == UVM_WRITE) ||
      (rw_reg_item.kind == UVM_BURST_WRITE))
  begin
    for (int i=0;i<burst_length;i++) begin
      for (int curr_byte=0;
           curr_byte <= reg_or_mem_bits /8;
           curr_byte+=burst_size) begin
        reg_data_q[0] =
          reg_data_q[0]>>(curr_byte*8);
        axi_data_q.push_back (reg_data_q[0] &
          ((1'b1 << (axi_rw_item.size * 8))-1));
      end
      void'(reg_data_q.pop_front());
    end
  end
end
```

E. Populate the bus transaction

```
axi_rw_item     = axi_item_t::type_id::create();
//AXI transaction Address
axi_rw_item.addr  = burst_addr;
//AXI transaction Length
axi_rw_item.burst_length = burst_length - 1;
//AXI transaction Kind
axi_rw_item.burst = (burst_length > 1)?
                    AXI_INCR : AXI_FIXED;
//AXI transaction Size
axi_rw_item.size = burst_size;
//Direction, Strobes, Data
if ((rw_reg_item.kind == UVM_WRITE) ||
    (rw_reg_item.kind == UVM_BURST_WRITE)) begin
  //AXI transaction kind
  axi_rw_item.read_or_write = AXI_TRANS_WRITE;
  //AXI transaction kind
  axi_rw_item.data_words = axi_data_q;
  // AXI transaction write strobes
  ...
  end
  else if ((rw_reg_item.kind == UVM_READ) ||
           (rw_reg_item.kind == UVM_BURST_READ))
    axi_rw_item.read_or_write = AXI_TRANS_READ;
```

After transformation is complete, we update the uvm_reg_bus_op_c object to hold the number of bits consumed and pass it to the resource DB, then return the bus transaction.

```
  //Wrapper uvm_reg_bus_op_c object update
  uvm_reg_bus_op_c_1 = new();
  uvm_reg_bus_op_c_1.n_bits = bytes_sent*8;
  if ((rw_reg_item.kind == UVM_WRITE) ||
      (rw_reg_item.kind == UVM_BURST_WRITE))
    uvm_config_db #(uvm_reg_bus_op_c)::set(null,
            "*", "rw_access_adapter_write",
            uvm_reg_bus_op_c_1);
  else
    uvm_config_db #(uvm_reg_bus_op_c)::set(null,
           "*", "rw_access_adapter_read",
           uvm_reg_bus_op_c_1);
  return axi_rw_item;
endfunction //reg2bus()
```

In some circumstances, you may have limitations or constraints on the bus that certain kinds of bus capabilities could not be utilized, the above code can be easily extended to take bus limitations and constraints into consideration when generating the burst. All what is needed would be a handle of the bus configuration object being passed to the adapter, the *reg2bus()* would check if a chosen burst feature is not supported and re-iterate if needed.

### *3.2.2 bus2reg() for an AMBA AXI*

Typically *bus2reg()* implementation would be relatively simple since we will be listening to lower level transactions, i.e. read and write beats, and not high level bursts. Therefore, it will convert lower level bus beats transactions to register transactions. It may need to listen to higher level read AXI bursts to get the whole burst data for the *do_bus_read()* method as shown below.

```
virtual function void bus2reg (uvm_sequence_item
        bus_item, ref uvm_reg_bus_op rw);
if (!$cast(axi_rw_beat, bus_item)) begin
    if ($cast(axi_rw_burst, bus_item))
      //This part is needed to fulfill needs of
      //do_bus_read() to get data out of the burst
      //in one shot
      if (axi_rw_burst.read_or_write ==
          AXI_TRANS_READ) begin
         typedef uvm_reg_data_t data_q_t [$];
         data_q_t read_data_q;
         foreach (axi_rw_burst.data_words[i])
           read_data_q.push_back
                     (axi_rw_burst.data_words[i]);
           uvm_config_db #(data_q_t)::set(null,
           "*", "bus2reg_read_data_q",
           read_data_q);
      end
    else
      `uvm_info ("RegMem",{"adapter
            [",this.get_name(),"] bus2reg()
            Casting failed!"}, UVM_FULL)
    return;
  end
  rw.kind = (axi_rw_beat.read_or_write ==
           AXI_TRANS_WRITE) ? UVM_WRITE:UVM_READ;
  rw.addr     = axi_rw_beat.addr;
  rw.data     = axi_rw_beat.data;
  rw.status   = UVM_IS_OK;
  //calculate byte enable from beat size
  ...
endfunction
```

## 4. COST-BENEFIT ANALYSIS AND EXPERIMENTAL RESULTS

### 4.1 Lines of code and complexity

As demonstrated above when taking a look at the proposed implementation of *do_bus_write()/do_bus_read()* w.r.t. the current implementation, you shall notice that lines of code and complexity are relatively close. While lines of code of proposed implementation may be smaller (since some of the overhead to chop data into bus bursts is moved to the *reg2bus()*and *bus2reg()* methods), yet using the resource DB and semaphores to share data between do_bus_write()/do_bus_read() and reg2bus()/bus2reg() methods and to support concurrent writes, or reads (working around *reg2bus()* prototype limitation) add some complexities. These complexities shall be eliminated if the *reg2bus()* prototype returns an array of bus transactions.

On the other hand the implementation of *bus2reg()/reg2bus()* methods in the proposed implementation would be more complex if one wants to benefit from a high performance bus powerful features. Or rather, stick with simple implementation of these methods for a trade off with simulation performance.

### 4.2 Simulation Performance

The number of simulation cycles depends on the number of transactions executed; the greater the number of executed transactions, the greater the simulation cycles and hence longer the simulation time will be. The proposed implementation attempts to send the smallest amount of bus transactions possible. On the other hand, the current implementation sends the maximum amount. Simulation performance is affected by the amount of context switching in your code. Each time you generate and execute an extra transaction, context is switched from the UVM registers context to the bus driver context, thus maximizing the number of transaction executed which would hurt simulation performance. This makes the proposed implementation suitable for high performance buses as it just moves the bottleneck from the UVM registers to the adapter implementation and the corresponding bus architecture. On Low performance buses, the overhead of accessing the resource DB for every bus transaction could be tangible, resulting in a slight performance degradation w.r.t. current implementation. An ultimate resolution is to extend the UVM register library to support a register to bus method that generates an array of bus transactions. Appendix B demonstrates the required extensions in detail.

### 4.3 Experimental Results

The following figure describes how simulation performance is affected when executing unnecessary bursts, attempting to perform a write to a 2KB memory on an AMBA AXI bus of 32-bit width. As shown below, simulation performance of the proposed implementation can be five times better than the current implementation. The current implementation will send 512 different AXI bursts; point “A” on the graph represents normalized simulation CPU time and cycles for the current implementation. The proposed implementation can send one burst with 512 different beats, reducing context switching and eliminating extra cycles; reducing simulation time and cycles to point “B” on the graph.

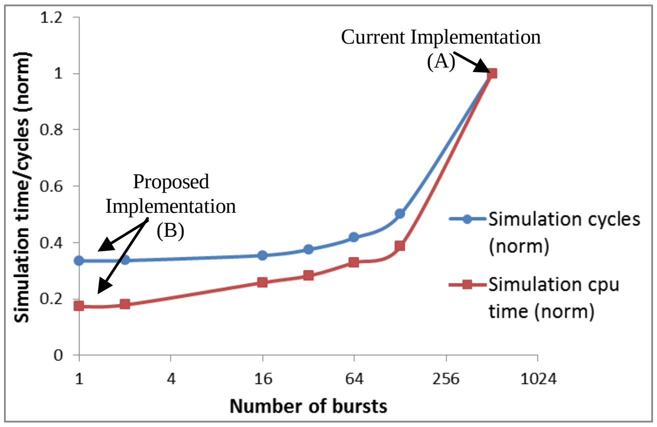

Figure 2. Effect of Executing Unnecessary AMBA AXI Bursts on Simulation Performance accessing a 2KB memory block

## 5. CONCLUSION

The paper has shed some light on current UVM registers front-door access implementation. In summary, the current implementation inserts undesired bottleneck when performing on high performance buses, by chopping data in bus-width chunks, generating a simple transaction for each chunk, and avoiding making use of bus powerful features when found. As a result the amount of transactions generated is maximized, which in turn would maximize context switching during simulation that would badly affect simulation performance.

We presented an alternative implementation that avoids chopping data, passing the decision making to the adapter translating from register transactions to bus transactions letting it decide how many of the data it could consume in one transaction. This way the bottleneck is moved to the adapter and the corresponding bus architecture. The current prototype of the *reg2bus()* in the UVM library allows returning only one transaction, although it would have been much better if the method returns array of transactions.

A cost-benefit analysis showed that in terms of lines of code count, code complexity, and use model, current and proposed implementations are similar. However when it comes to simulation performance, the proposed implementation can super exceed the current implementation on high performance buses.

## 6. REFERENCES

[1] UVM User Manual, uvmworld.org.
[2] UVM Reference, uvmworld.org.
[3] UVM Open Source Kit, uvmworld.org.
[4] UVM/OVM Cookbook, verificationacademy.com/uvm-ovm.
[5] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009.
[6] AMBA AXI reference, infocenter.arm.com.

## APPENDIX A. – OVERLAY REGISTER MAP *DO_BUS_READ()* IMPLEMENTATION

```
// $Id: uvm_reg_map_ext.svh,v 1.12 2011/06/03 00:00:00 ayehia Exp $
//----------------------------------------------------------------------
// Ahmed Yehia ahmed_yehia@mentor.com
// Copyright 2005-2013 Mentor Graphics Corporation
// All Rights Reserved Worldwide
//
// Licensed under the Apache License, Version 2.0 (the
// "License"); you may not use this file except in
// compliance with the License. You may obtain a copy of
// the License at
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in
// writing, software distributed under the License is
// distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See
// the License for the specific language governing
// permissions and limitations under the License.
//----------------------------------------------------------------------
task do_bus_read (uvm_reg_item rw, uvm_sequencer_base sequencer, uvm_reg_adapter adapter);
    //Read data queue
    typedef uvm_reg_data_t data_q_t [$];
    data_q_t read_data_q;

    //bus_width in bytes (important to know your limits). Default (when UVM_HIER is passed)
    //is to get narrowest bus. Bus write depending on bus type and configuration
    int unsigned bus_width      = get_n_bytes();
    int unsigned bus_width_bits = bus_width * 8;

    //Byte enable, applicable only for UVM_FIELD.
    //Initialize to all ones to avoid confusion otherwise
    uvm_reg_byte_en_t  byte_en  = '1;

    //Address related fields
    uvm_reg_map_info map_info;

    //Array of addresses. For a register its size depends on the bus_width;
    //if bus_width is smaller than the register width then the size of the addr,
    //array will simply be register_width/bus_width, otherwise the addr will hold only 1 location
    //For memory each memory location will be treated as one register
    uvm_reg_addr_t     start_address, addrs[$];

    //Specification variables
    int lsb, addr_skip, n_bits, n_bits_total, extra_bits;
    uvm_sequence_item bus_req, bus_req_q[$];
    uvm_reg_bus_op rw_access;
    uvm_reg_bus_op_c uvm_reg_bus_op_c_read = new();

    //Get Addresses and specs for all kinds (fields/registers/mem)
    Xget_bus_infoX(rw, map_info, n_bits, lsb, addr_skip);
    //Macro needed for IUS limitation with assignment compatibility of dynamic arrays to queues
    `UVM_DA_TO_QUEUE(addrs,map_info.addr)

    //UVM_FIELD is only picked up if one configures his fields to be accessed individually
    if (rw.element_kind == UVM_FIELD) begin
      //Excess bits on bus width boundaries. Flags the start of the field w.r.t
      //the bus width boundary, this is useful when calculating byte enable
      //and when shifting the data to align with the start of the field
      extra_bits = (lsb % bus_width_bits);
```

```
  //Calculate byte enables if adapter supports byte enable.
  if (adapter.supports_byte_enable) begin
    int idx        = extra_bits / 8;         // Initialize index to start of bytes locations
    // Total size of the field to be accessed in bits (size of a field)
    int access_bits = extra_bits % 8 + n_bits;
    byte_en        = '0;                     //Initialization of byte_en all zeros
    while(access_bits > 0) begin
      byte_en[idx++] = 1'b1;
      access_bits -= 8;
    end
  end

  //Skip addresses un-needed to access the fields (addresses of the rest of the reg),
  //byte_enable should take care of the footer addresses
  for (int i=0; i<addr_skip; i++)
    void'(addrs.pop_front());
end

//Total number of bits to be sent. For memory, n_bits come for one memory location
//and so I am multiplying by the value size to reflect for the memory
n_bits_total = n_bits*rw.value.size();
start_address = addrs[0];             //Capture the start address

//Assing byte_en (useful only for UVM_FIELD), kind and n_bits
rw_access.byte_en = byte_en;
rw_access.kind    = rw.kind;
rw_access.n_bits  = n_bits_total;

//Lock the semaphore, needed if concurrent writes will be supported
uvm_reg_bus_op_c::reg2bus_read_sm.get(1);

//Loop for bursts
while (rw_access.n_bits > 0) begin //As long as there is data to send
  rw_access.addr    = start_address;

  adapter.m_set_item(rw);
  bus_req = adapter.reg2bus(rw_access);

  //Something wrong with the  adapter.reg2bus(), trigger a UVM_FATAL
  if (bus_req == null)
    `uvm_fatal("RegMem",{"adapter [",adapter.get_name(),"] didnt return a bus transaction"});

  //Push the bus_req to the queue of bus_requests. This is needed due to the limitation
  //in the adapter.reg2bus() prototype returning only one sequence item
  bus_req_q.push_back (bus_req);

  //A means for communication between the adapter.reg2bus() and the do_bus_write().
  //This is to workaround the limitation of rw_access being passed as const
  uvm_config_db #(uvm_reg_bus_op_c)::get(null, "",
                                        "rw_access_adapter_read", uvm_reg_bus_op_c_read);

  //Something wrong with the  adapter.reg2bus(), trigger a UVM_FATAL
  if (uvm_reg_bus_op_c_read == null)
    `uvm_fatal("RegMem",{"adapter [",adapter.get_name(),"] Means of communication is broken
                        between adapter.reg2bus() and reg_mem_map.do_bus_write()!"});
  if (uvm_reg_bus_op_c_read.n_bits == 0)
    `uvm_fatal("RegMem",{"adapter [",adapter.get_name(),"] Adapter returned n_bits of zero,
                         this could result in an infinite loop!"});

  //update start_addr to reflect number of bits that has been already sent out
  start_address    += uvm_reg_bus_op_c_read.n_bits/(8*get_addr_unit_bytes());
  rw_access.n_bits -= uvm_reg_bus_op_c_read.n_bits;
end //while (rw_access.n_bits > 0)

//Free the semaphore
uvm_reg_bus_op_c::reg2bus_read_sm.put(1);

//Drive read transactions to driver
foreach (bus_req_q[i])
begin
  uvm_sequence_item bus_req = bus_req_q[i];
  bus_req.set_sequencer(sequencer);
  rw.parent.start_item(bus_req,rw.prior);

  if (rw.parent != null && rw_access.addr == addrs[0]) begin
    rw.parent.pre_do(1);
    rw.parent.mid_do(rw);
  end
  rw.parent.finish_item(bus_req);
  bus_req.end_event.wait_on();
  uvm_reg_bus_op_c::bus2reg_read_sm.get(1);

  if (adapter.provides_responses) begin
    uvm_sequence_item bus_rsp;
    uvm_access_e op;
    rw.parent.get_base_response(bus_rsp);
    adapter.bus2reg(bus_rsp,rw_access);
  end
  else begin
    adapter.bus2reg(bus_req,rw_access);
  end

         //Get the read data from the bus and update the item
  uvm_config_db #(data_q_t)::get(null, "*", "bus2reg_read_data_q", read_data_q);
  uvm_reg_bus_op_c::bus2reg_read_sm.put(1);
  rw.status = rw_access.status;
  for (int j = 0, k = 0; j < read_data_q.size(); j++, k++) begin
    rw.value[k]    = 0;
    for (int curr_byte=0; curr_byte < `UVM_REG_BYTENABLE_WIDTH; curr_byte+=bus_width) begin
      if (curr_byte > 0) begin
        j++;
        if (j >= read_data_q.size())
          break;
      end
      read_data_q[j] = read_data_q[j] & ((1<<bus_width*8)-1);
      rw.value[k]   |= read_data_q[j] << curr_byte*8;
      if (rw.element_kind == UVM_FIELD)
        rw.value[k] = (rw.value[k] >> extra_bits) & ((1<<n_bits)-1);
      if ((rw.status == UVM_IS_OK) && ((^read_data_q[j]) === 1'bx))
        rw.status = UVM_HAS_X;
      end
    end

    if (rw.parent != null && rw_access.addr == addrs[addrs.size()-1])
      rw.parent.post_do(rw);
  end //foreach (bus_req_q[i])
endtask //do_bus_read()
```

## APPENDIX B. – DEMONSTRATION OF SOME OF THE REQUIRED MODIFICATIONS IN THE UVM REGISTERS LIBRARY FOR HANDLING HIGH SPEED BUSES EFFICIENTLY

```
// $Id: uvm_reg_map_ext.svh,v 1.12 2011/07/20 00:00:00 ayehia Exp $
//----------------------------------------------------------------------
// Ahmed Yehia ahmed_yehia@mentor.com
// Copyright 2005-2013 Mentor Graphics Corporation
// All Rights Reserved Worldwide
//
// Licensed under the Apache License, Version 2.0 (the
// "License"); you may not use this file except in
// compliance with the License. You may obtain a copy of
// the License at
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in
// writing, software distributed under the License is
// distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See
// the License for the specific language governing
// permissions and limitations under the License.

typedef uvm_sequence_item uvm_sequence_item_q [$];

virtual class uvm_reg_adapter extends uvm_object;
  ...
  virtual function uvm_sequence_item_q reg2bus_arr(const ref uvm_reg_bus_op rw);
  endfunction
endclass


task uvm_reg_map::do_bus_write (uvm_reg_item rw, uvm_sequencer_base sequencer, uvm_reg_adapter adapter);
  uvm_reg_addr_t     addrs[$];
  uvm_reg_map        system_map = get_root_map();
  int unsigned       bus_width  = get_n_bytes();
  int unsigned bus_width_bits = bus_width * 8;
  uvm_reg_byte_en_t  byte_en    = -1;
  uvm_reg_map_info   map_info;
  int                n_bits;
  int                lsb, skip, addr_skip, n_bits_total, extra_bits;
  int unsigned       curr_byte;
  int                n_access_extra, n_access;
  uvm_sequence_item bus_req, bus_req_q[$];
  uvm_reg_bus_op rw_access;
  Xget_bus_infoX(rw, map_info, n_bits, lsb, skip);
  `UVM_DA_TO_QUEUE(addrs,map_info.addr)
  // if a memory, adjust addresses based on offset
  if (rw.element_kind == UVM_MEM)
    foreach (addrs[i])
      addrs[i] = addrs[i] + map_info.mem_range.stride * rw.offset;
  if (rw.element_kind == UVM_FIELD) begin
    //Excess bits on bus width boundaries. Flags the start of the field w.r.t the bus width boundary,
    //this is useful when calculating byte enable and when shifting the data to align with the start of
    //the field
    extra_bits = (lsb % bus_width_bits);
    //Calculate byte enables if adapter supports byte enable.
    if (adapter.supports_byte_enable) begin
      int idx         = extra_bits / 8; //Initialize index to start of bytes locations
      //Total size of the field to be accessed in bits (size of a field)
      int access_bits = extra_bits % 8 + n_bits;
      byte_en         = '0; //Initialization of byte_en all zeros
      while(access_bits > 0) begin
        byte_en[idx++] = 1'b1;
        access_bits -= 8;
      end
    end
    //Skip addresses un-needed to access the fields (addresses of the rest of the reg),
    //byte_enable should take care of the footer addresses
    for (int i=0; i<addr_skip; i++)
      void'(addrs.pop_front());
    //Update value to align on the field.
    //No need to do a foreach cause for a field value size is 1 anyways.
    foreach (rw.value[val_idx])
      rw.value[val_idx] = rw.value[val_idx] << extra_bits;
  end
  //Total number of bits to be sent. For memory, n_bits come for 1 memory location and so I am
  //multiplying by the value size to reflect for the whole memory
  n_bits_total = n_bits*rw.value.size();
  //byte_en (useful only for UVM_FIELD), kind and n_bits
  rw_access.byte_en = byte_en;
  rw_access.kind    = rw.kind;
  rw_access.n_bits  = n_bits_total;
  rw_access.addr    = addrs[0];
  adapter.m_set_item(rw);
  bus_req_q = adapter.reg2bus_arr(rw_access);
  adapter.m_set_item(null);
  if (bus_req_q.size() == 0) begin
    //Method reg2bus_arr was not implemented, revert to old behavior for backward compatibility
    foreach (rw.value[val_idx]) begin: foreach_value
      uvm_reg_data_t value = rw.value[val_idx];
      foreach(addrs[i]) begin: foreach_addr
        uvm_reg_data_t data;
        data = (value >> (curr_byte*8)) & ((1'b1 << (bus_width * 8))-1);
        `uvm_info(get_type_name(), $sformatf("Writing 'h%0h at 'h%0h via map \"%s\"...",
                  data, addrs[i], rw.map.get_full_name()), UVM_FULL);
        if (rw.element_kind == UVM_FIELD) begin
          for (int z=0;z<bus_width;z++)
            rw_access.byte_en[z] = byte_en[curr_byte+z];
        end
        rw_access.kind    = rw.kind;
        rw_access.addr    = addrs[i];
        rw_access.data    = data;
        rw_access.n_bits  = (n_bits > bus_width*8) ? bus_width*8 : n_bits;
        rw_access.byte_en = byte_en;
        adapter.m_set_item(rw);
        bus_req = adapter.reg2bus(rw_access);
        adapter.m_set_item(null);
        if (bus_req == null)
          `uvm_fatal("RegMem",{"adapter [",adapter.get_name(),"] didnt return a bus transaction"});
        bus_req_q.push_back(bus_req);
        curr_byte += bus_width;
        n_bits -= bus_width * 8;
      end: foreach_addr
      foreach (addrs[i])
        addrs[i] = addrs[i] + map_info.mem_range.stride;
    end: foreach_value
  end
  foreach (bus_req_q[i])begin
    uvm_sequence_item bus_req = bus_req_q[i];
    bus_req.set_sequencer(sequencer);
    rw.parent.start_item(bus_req,rw.prior);
    if (rw.parent != null && rw_access.addr == addrs[0])
      rw.parent.mid_do(rw);
    rw.parent.finish_item(bus_req);
    bus_req.end_event.wait_on();
    if (adapter.provides_responses) begin
      uvm_sequence_item bus_rsp;
      uvm_access_e op;
      // TODO: need to test for right trans type, if not put back in q
      rw.parent.get_base_response(bus_rsp);
      adapter.bus2reg(bus_rsp,rw_access);
    end
    else begin
      adapter.bus2reg(bus_req,rw_access);
    end
    if (rw.parent != null && rw_access.addr == addrs[addrs.size()-1])
      rw.parent.post_do(rw);

    rw.status = rw_access.status;

    if (rw.status == UVM_NOT_OK)
         break;
  end
endtask: do_bus_write
```