Mission Systems

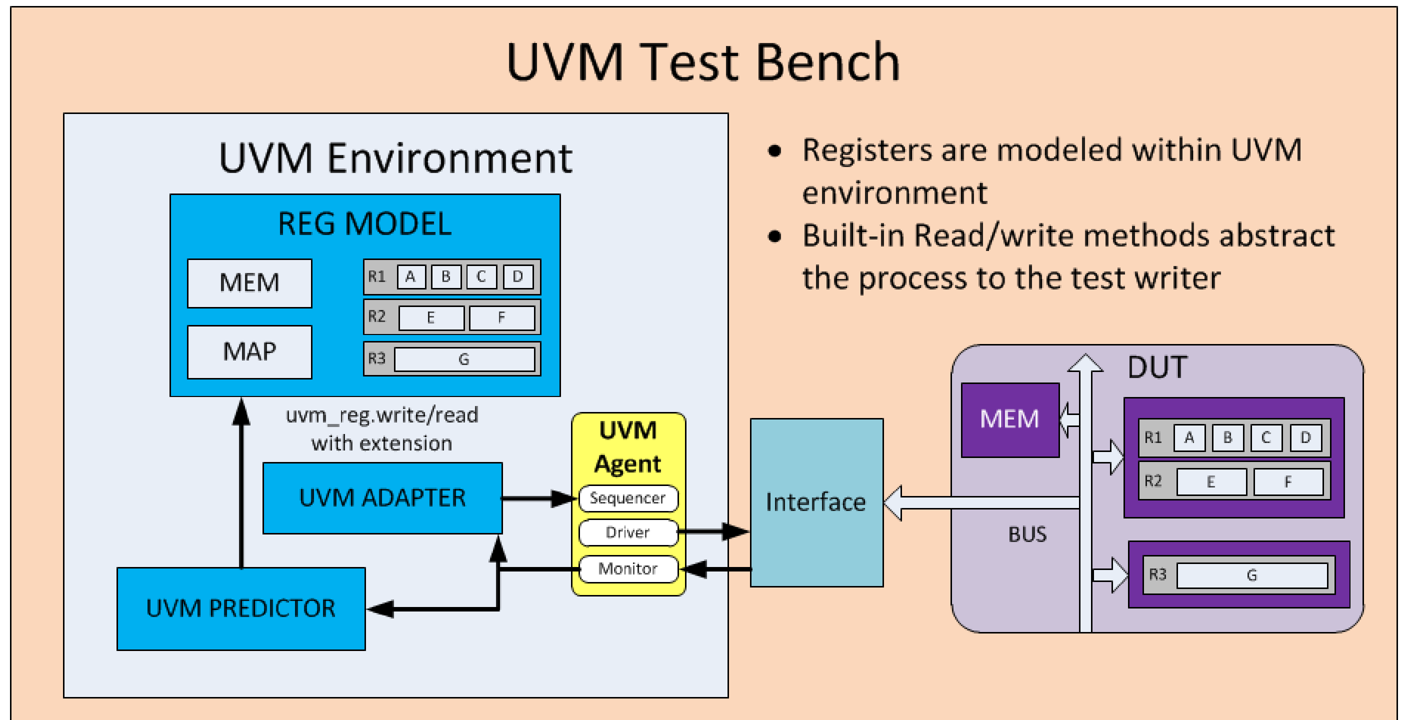
Adapting the UVM Register Abstraction Layer for Burst Access

Mark Villalpando

UVM Register Abstraction Layer

- Standardized method to model DUT registers and memories
- Passive and active mirroring
- Powerful API
- Coverage Capabilities

uvm_reg
- *write()*
- *read()*
- *set()*
- *get()*
- *mirror()*
- *predict()*



The Single Access Problem

Front door access is limited by the *uvm_reg.read()* and *uvm_reg.write()* methods

```
uvm_status_e status;
uvm_reg_data_t wdata;
wdata = 16'h55;
m_reg_block.reg_0.write(status, wdata);
```

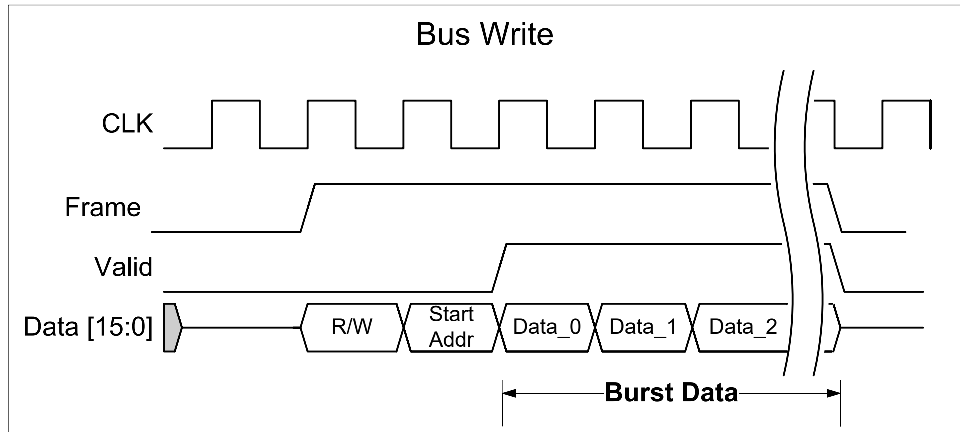
Only supports single access to the design!

Multiple accesses require multiple write commands

```
m_reg_block.reg_0.write(status, wdata0);
m_reg_block.reg_1.write(status, wdata1);
m_reg_block.reg_2.write(status, wdata2);
```

The Single Access Problem

The example method is based on this protocol



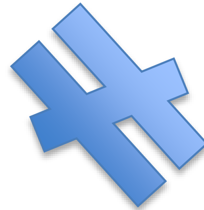
What if we used *uvm_mem*?

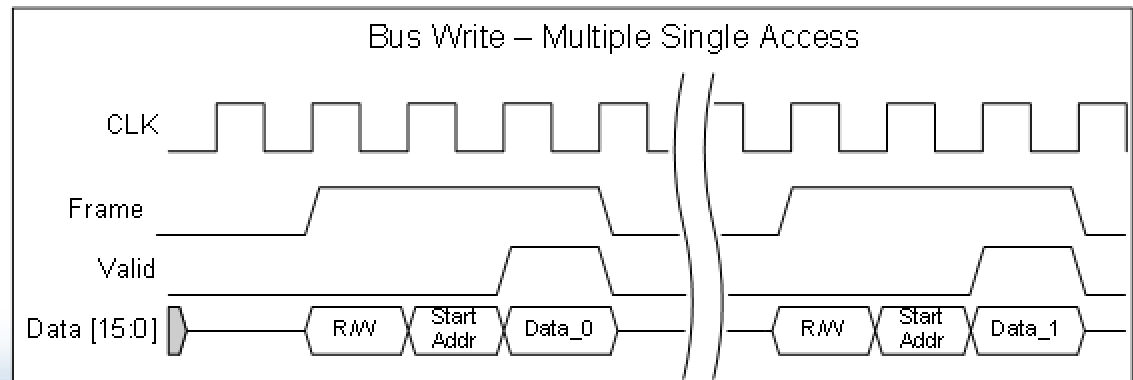
- ✖ *uvm_mem* does not mirror!

We can adapt!

```
m_reg_block.reg_0.write(.extension());
```

```
m_reg_block.reg_0.write(status, wdata0);
m_reg_block.reg_1.write(status, wdata1);
m_reg_block.reg_2.write(status, wdata2);
```





Extension Object

- Custom class to store burst data and other miscellaneous information
- Transmitted data is stored within a queue
- Can pass any type of information to the Register Adapter
- Accepted input into *uvm_reg.write/read()*

```
class cfg_info extends uvm_object;
  `uvm_object_utils(cfg_info)

  rand int unsigned  transmit_delay;
  rand logic  [ 5:0] device_id;

  rand logic  [15:0] burst_data [$];

 constraint c_transmit_delay  {
    transmit_delay  > 4;
    transmit_delay <= 10;
  }

 function new (string name = "");
    super.new(name);
 endfunction
endclass
```

Register Adapter

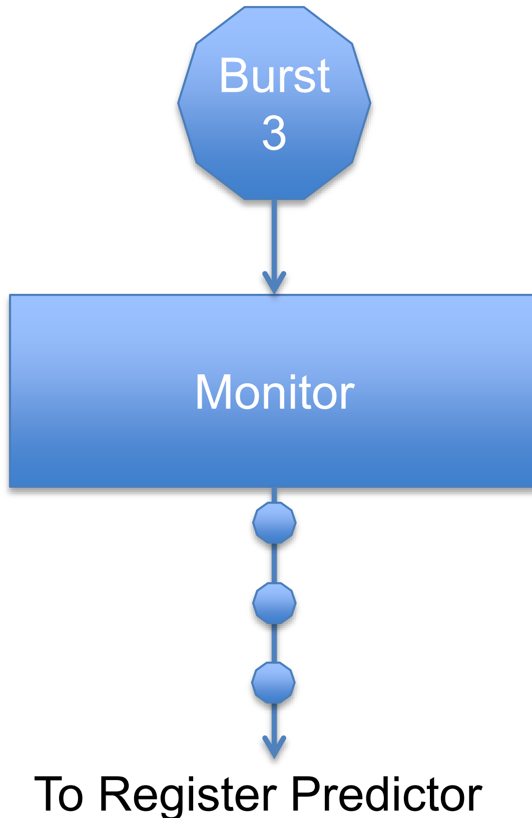
- Transcribes Extension Object information to the Sequence Item in *reg2bus()*

```
virtual function uvm_sequence_item reg2bus( const ref uvm_reg_bus_op rw );

  uvm_reg_item item       = get_item();

  cfg_seq_item cfg_trans  = cfg_seq_item::type_id::create("cfg_trans");
  cfg_info     cfg_ext    = cfg_info::type_id::create("cfg_ext");

  cfg_trans.dir           = (rw.kind == UVM_READ) ? READ : WRITE;
  cfg_trans.address[15:0] = rw.addr[15:0];

  if (item.extension != null) begin
    $cast(cfg_ext, item.extension);
    cfg_trans.transmit_delay        = cfg_ext.transmit_delay;
    cfg_trans.device_id             = cfg_ext.device_id;
  end

  if (rw.kind == UVM_WRITE) begin
     cfg_trans.wr_data          = cfg_ext.burst_data;
  end
  return cfg_trans;
endfunction: reg2bus
```

We're done, right?

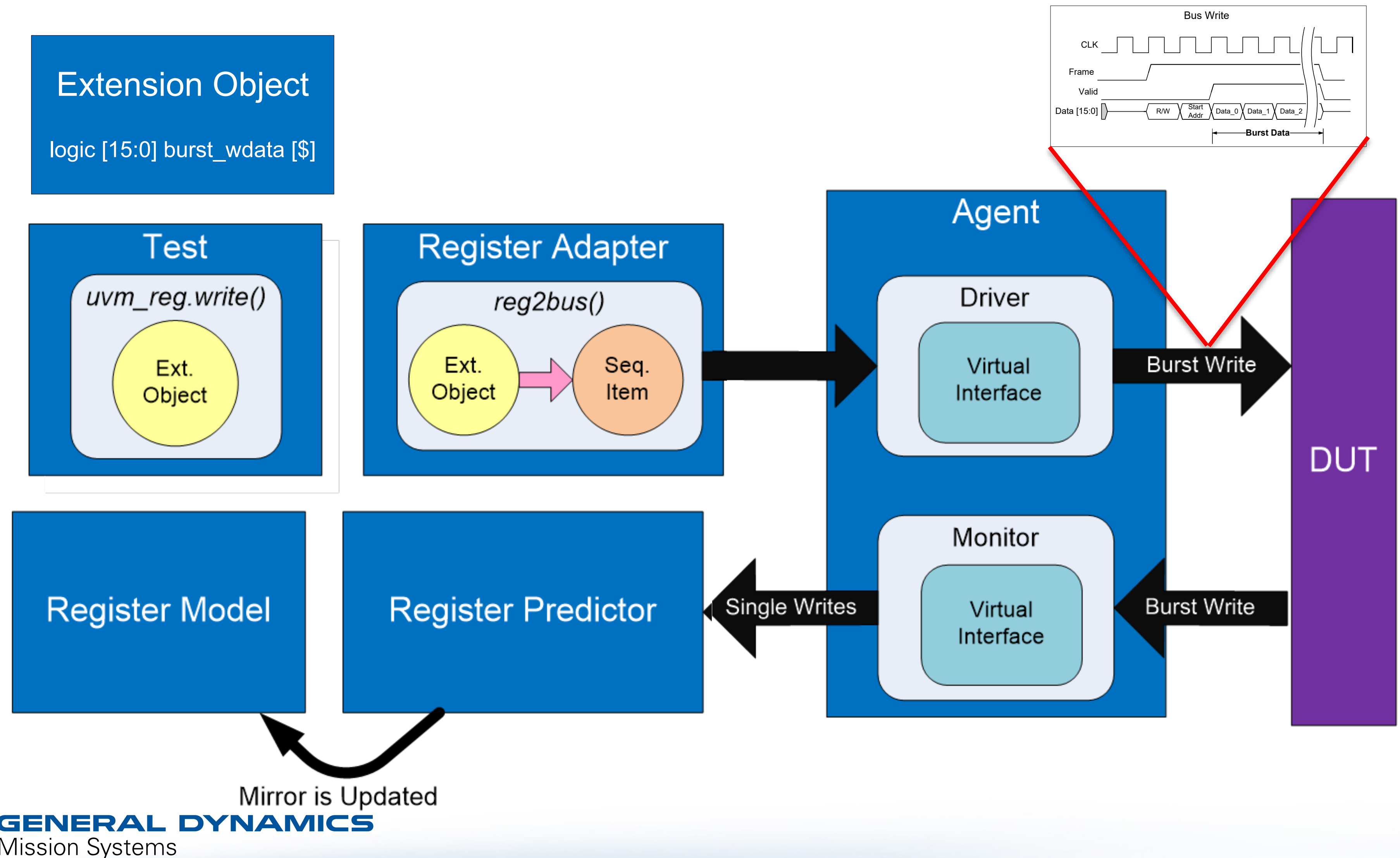
Monitor

- Divides burst transactions into single access transactions




```
virtual protected task collect_transactions();
    cfg_transfer transfer;
    cfg_transfer trans_predictor;

   forever begin
      transfer = new();
      lcb_config.v_lcb_if.collect_transaction(transfer);
      $cast(trans_predictor, transfer.clone);

      while (transfer.length > 1) begin
         $cast(trans_predictor, transfer.clone);
         ap_cfg_transfer.write(trans_predictor);
            if (transfer.dir == READ)
              transfer.rd_data.pop_front();
            else
              transfer.wr_data.pop_front();
         transfer.address++;
         transfer.length--;
       end
       ap_cfg_transfer.write(trans_predictor);
    end

  endtask : collect_transactions
```

Bringing it All Together – The *uvm_reg* Burst Write



Test Case

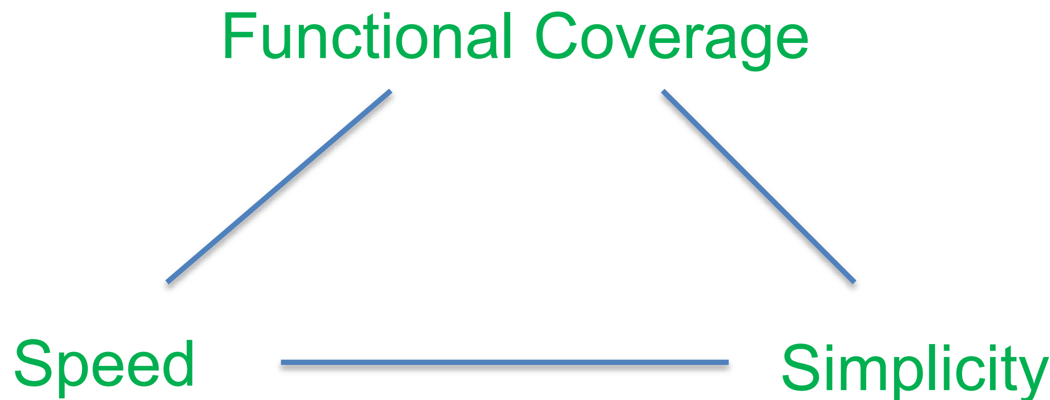
```
class burst_reg_test extends uvm_test;

...
  task run_phase( uvm_phase phase );
    super.run();

     ...
    m_cfg_ext                  = cfg_info::type_id::create(“m_cfg_ext");
    m_cfg_ext.device_id        = 1;
    m_cfg_ext.transmit_delay = 2;
    m_cfg_ext.burst_data       = {16’h3, 16’h4, 16’h5};

    m_reg_block.reg0.write(status, 16’h00, .extension(m_cfg_ext));
    m_reg_block.reg0.read(status, 16’h00, .extension(m_cfg_ext));

    ...
  endtask: run_phase

endclass: burst_reg_test
```

Limitations

- Will require a custom Extension Object for the bus protocol
  - Must be used for each burst operation
- Burst read data will not be returned to the test/sequence
  - Cannot “return” an extension object from the bus
- *uvm_reg.mirror()* front door method is not supported
  - Combination of *uvm_reg.read()* and *uvm_reg.predict()* with *UVM_CHECK*

Summary

- Consistent and easy method for exercising the burst protocol of a bus
- No custom methods needed for updating the register model, passive mirroring is maintained for every register
- No changes to UVM Library components
- Normal method operation is unaffected
- Extension Object can support a multitude of purposes



Recommendations

- Add a true burst access method to the *uvm_reg_map* class
- This method shouldn't produce multiple single access operations (already exists!)
- Rather, pass and return multiple data words from a single operation
- *uvm_reg_predictor* will require updates

Questions ?