Hardcore Abstraction

Replacing Hardcoded Register Values in VIPs

Alex Melikian
Hilmar Van Der Kooij
Verilab Inc.

October 1st, 2015
Ottawa, Canada

Agenda

Describing the Problem

Overview of UVM Register Model

Guidelines & Examples of VIP Register Abstraction

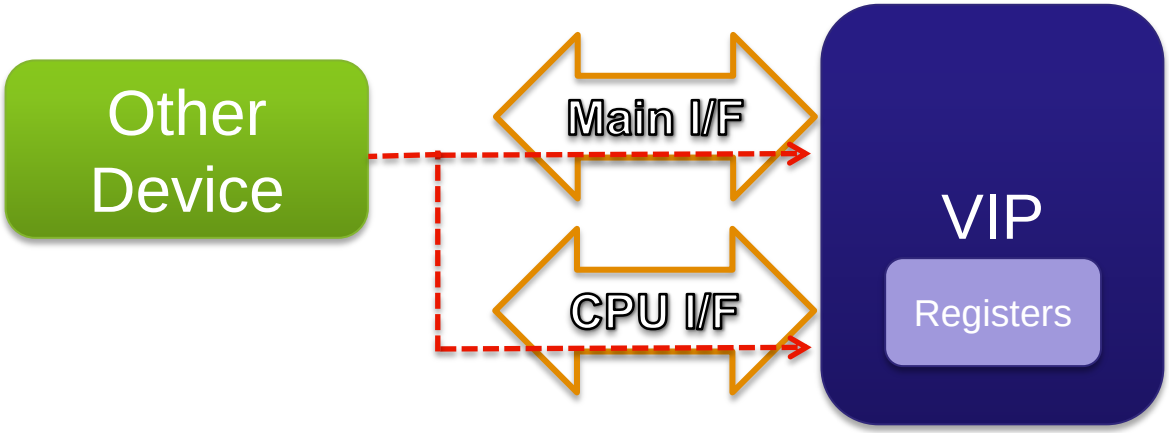
Takeaways

Describing the Problem

Why are there hardcoded register values in VIPs?

Protocols Include Registers

- Many of today's protocols include register sets
  - Serial Protocols: MIPI M-PHY, SoundWire, High Definition Audio
  - Non-Serial Protocols: DDR4 SDRAM
- Registers control or configure operational parameters
- Creates the need for VIPs implementing these protocols to model the register set and their functionality



Implementation Using Hardcoding

- (Unfortunately) VIP code to implement register features may contain hardcoded values

```
// if address and data is for 0x0100 register & trigger field
if (virtual_if.addr==16'h0100) && (virtual_if.data[5]==1'b1)
   do_something();
```

- Quick to code, but clearly invites problems:
  - Not robust to any register attribute changes
  - Likely to introduce hard-to-find run time bugs
  - `define help, but still problematic especially if manually updated
  - A lot of time can be lost over the course of a project

Solution: Using Abstraction

Synopsys Users Group

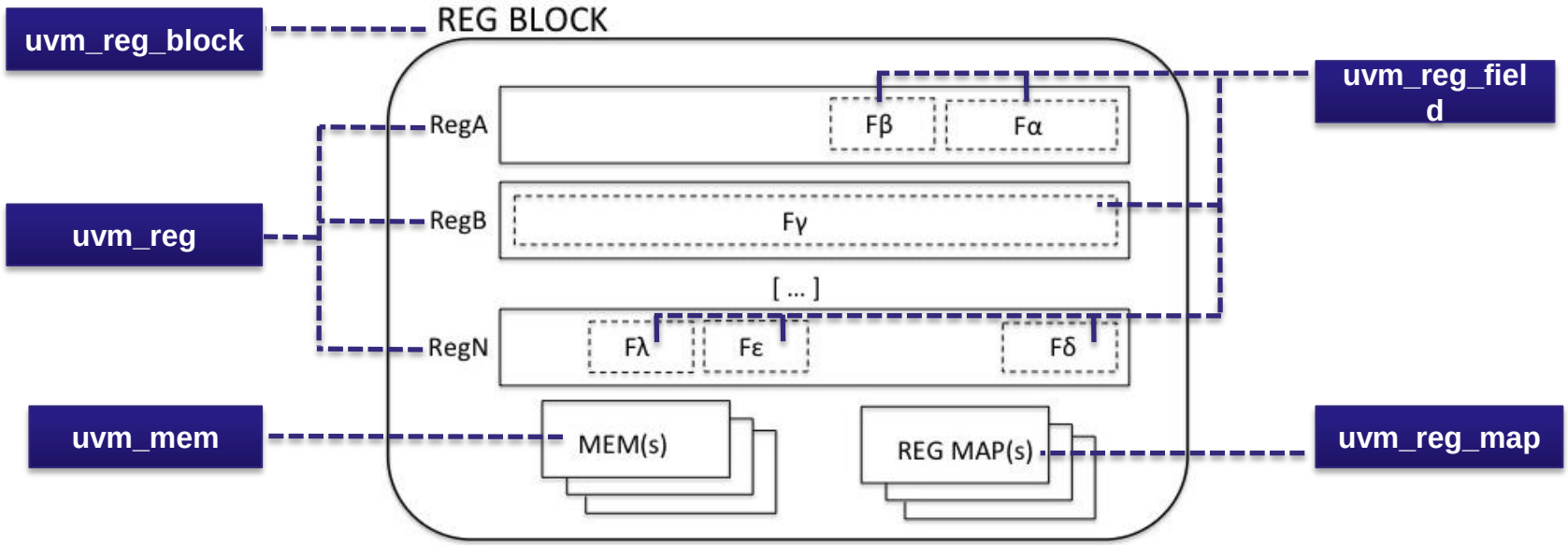
uvm_reg	uvm_reg_field	uvm_mem	uvm_reg_block
new()	new()	new()	new()
configure()	configure()	configure()	configure()
set_offset()	get_name()	set_offset()	create_map()
get_name()	get_lsb_pos()	get_name()	get_name()
get_address()	get_n_bits()	get_size()	get_blocks()
set()	set()	write()	get_reg_by_name()
get()	get()	read()	get_reg_by_addr()

- Leveraging off the UVM Register Model classes
  - Encapsulate and structure register related definitions
  - Rich set of access, introspection and query methods
  - Proven library with additional EDA tools available (code auto-generation, eg RALGEN)

Overview of UVM Register Model

Conventional and Proposed Use Cases

UVM Register Classes



- **`uvm_reg_field`**: base class for an atomic register fields
- **`uvm_reg`**: base class for an individual register
- **`uvm_mem`**: base class for contiguous storage locations
- **`uvm_reg_map`**: utility class representing address map
- **`uvm_reg_block`**: base class representing hierarchy

UVM Register Classes Code

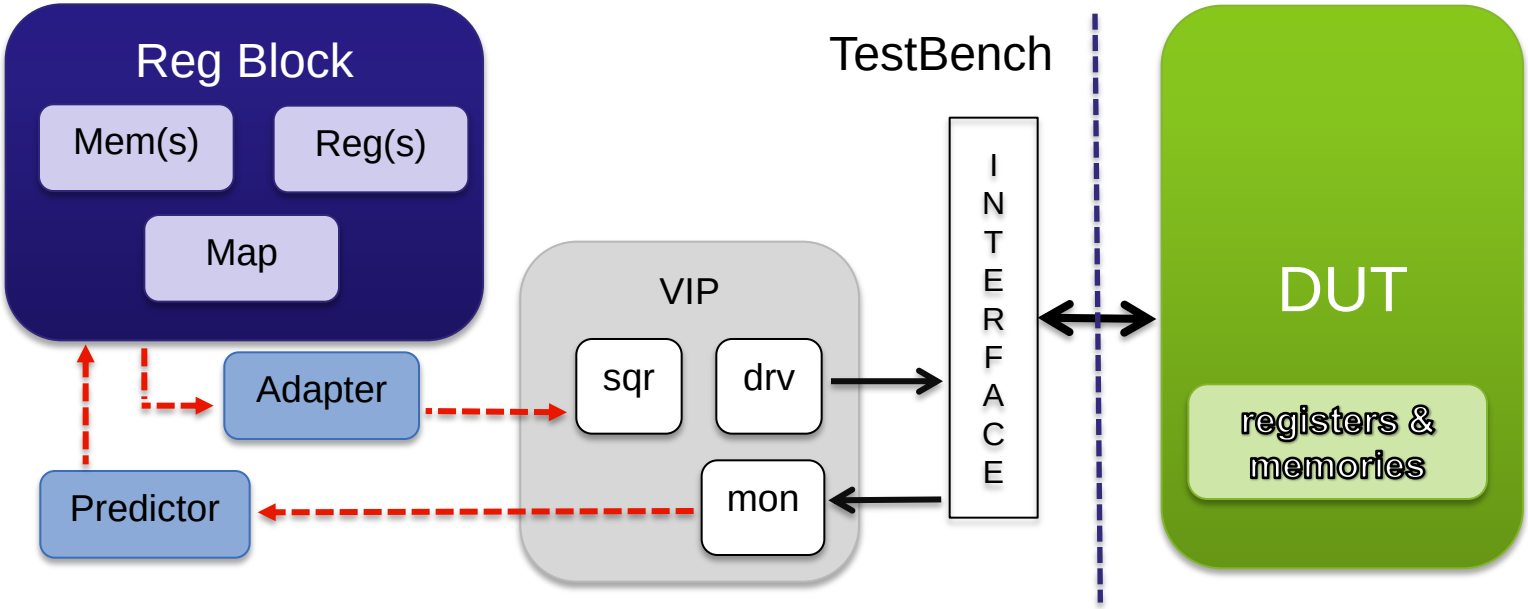
```
class my_vip_ref_regblock extends uvm_reg_block;
[…]
  protocol_reg1 protocol_reg1_reg;
  protocol_reg2 protocol_reg2_reg;
  protocol_reg3 protocol_reg3_reg;

  virtual function void build();
    protocol_reg1_reg = protocol_reg1::type_id::create("...");
    protocol_reg1_reg.configure(this);
    protocol_reg1_reg.build();

    protocol_reg2_reg = protocol_reg2::type_id::create("...");
    [...]
    protocol_reg3_reg = protocol_reg3::type_id::create("...");
    [...]

    default_map.add_reg(protocol_reg1_reg, 'h100, "RW");
    default_map.add_reg(protocol_reg2_reg, 'h104, "RW");
    default_map.add_reg(protocol_reg3_reg, 'h108, "RW");
  endfunction
endclass
```

uvm_reg_block derived class

uvm_reg derived classes - contain uvm_reg_field(s)

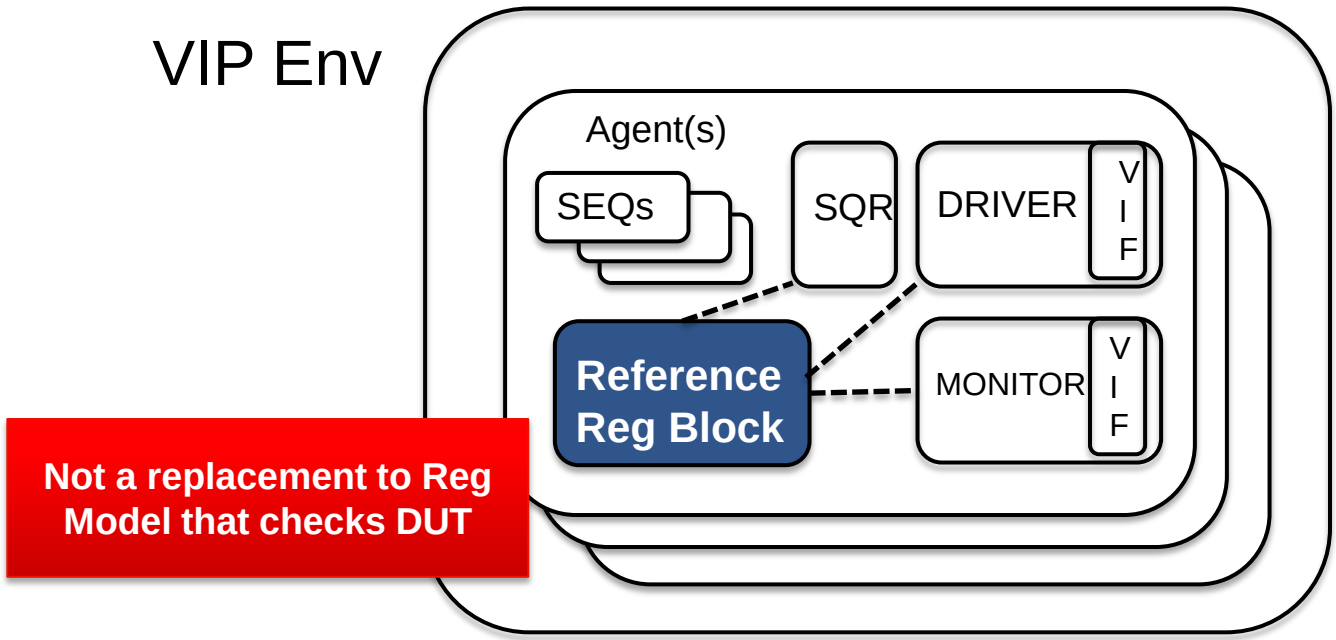
default_map (uvm_reg_map) to create address map

Conventional Use Case in TB



- Register model (register block) included in TB
- **`uvm_reg_predictor`** & **`uvm_reg_adaptor`** classes are used to quickly integrate with TB components
- Register model can now automatically execute and check DUT register functionality

Proposed New Use Case in VIP



- Code of Reference Register Block similar to TB use case
- Represents protocol register set the VIP must model
- Creates a layer of abstraction between VIP operations and low level register details

Synopsys Users Group

VIP Register Abstraction

Guidelines and Examples

Abstraction of Register Access

- Take the original example with hardcoded values …

```
// if address and data is for 0x0100 register & trigger field
if (vif.addr==16'h0100) && (vif.data[5]==1'b1)
   do_something();
```

- Replace with abstract methods in reference reg block

```
my_vip_ref_regblock  ref_regblock;

ref_regblock = my_vip_ref_regblock::type_id::create("...");
if (vif.addr == ref_regblock.protocol_reg1.get_offset())&&
   (vif.data[ref_regblock.protocol_reg1.trigger.get_lsb_pos()]==1))
  do_something()
```

UVM register class library query methods used here

Reference register block (regs/fields members) used here

Advantages of Abstraction

- A bit more code, but …
  - Far more robust, will automatically adapt to any changes in addresses or offsets
  - Reference register block and its attributes are defined in a single centralized file
  - Eliminates time costly run-time bugs due to hard-coded values
  - Trivial to fix compilation errors if registers radically change

```
my_vip_ref_regblock  ref_regblock;

ref_regblock = my_vip_ref_regblock::type_id::create("...");
if (vif.addr == ref_regblock.protocol_reg1.get_offset())&&
   (vif.data[ref_regblock.protocol_reg1.trigger.get_lsb_pos()]==1))
  do_something()
```

Abstraction Using Trans Item

- No dedicated i/f for registers access in some scenarios (serial i/f protocols)
- VIP's transaction item can be used instead

```
trans = my_vip_transaction::type_id::create("");

 [ … ]

 if ( vip_detected_write_operation()) begin
   trans.register_operation = WRITE;
   trans.addr = vip_decoded_write_address( );
 end

 [ … ]

 if (trans.register_operation==WRITE) &&
    (trans.addr == ref_regblock.protocol_reg1.get_offset()) &&
    (trans.data[ref_regblock.protocol_reg1.trigger.get_lsb_pos()]==1)
    do_something()
```

VIP's transaction item

VIP decodes operation into trans item

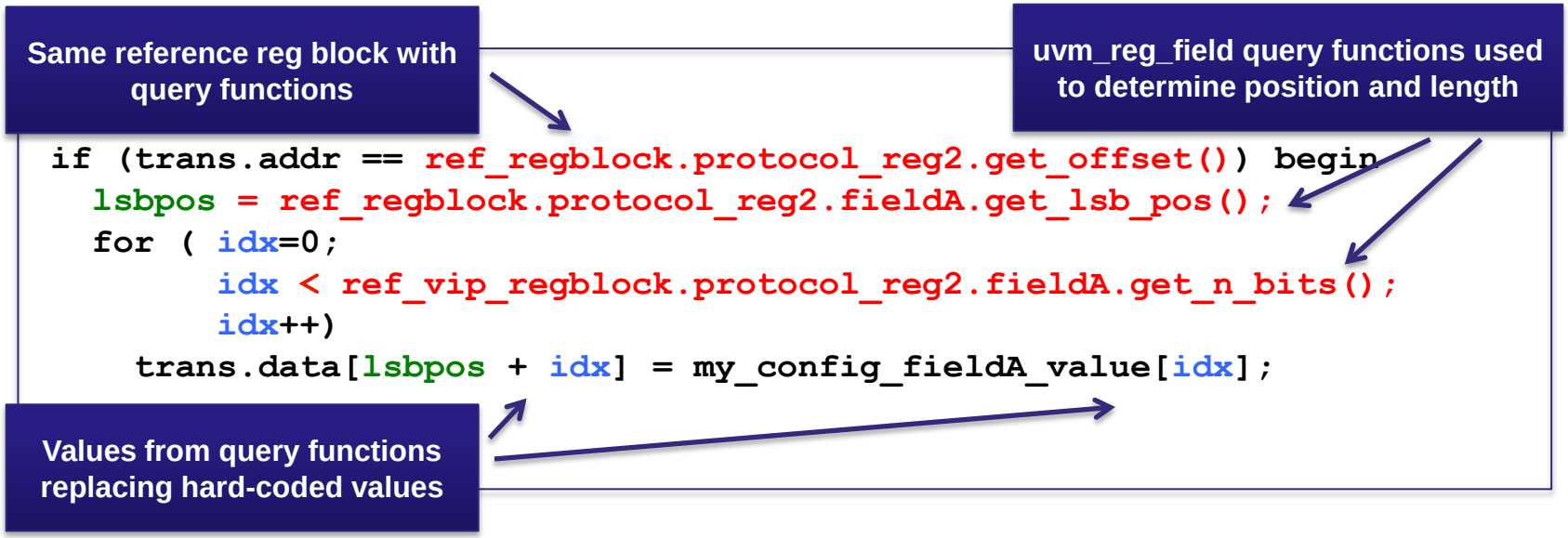
Trans item instead of 'vif' used to determine register access

Register Value Retrieval

- A hardcoded example likely to cause trouble

```
if (vif.addr == 16'h0100)
  vif.data[7:5] = my_config_fieldA_value;
```

- Same operation using abstracted method



Further Leverage of Register Block

- VIP code can use reference register block to also hold and report values
- Use UVM register class access methods:
  - uvm_reg::get( ) & uvm_reg_field::get( )
  - uvm_reg::set( ) & uvm_reg_field::set( )

```
if (trans.addr == ref_regblock.protocol_reg2.get_offset()) begin
  lsbpos = ref_regblock.protocol_reg2.fieldA.get_lsb_pos();
  for ( idx=0;
        idx < ref_vip_regblock.protocol_reg2.fieldA.get_n_bits();
        idx++)
    trans.data[lsbpos + idx] = my_config_fieldA_value[

      ref_regblock.protocol_reg2.set( trans.data );

  […]= ref_regblock.protocol_reg2.fieldA.get();
  […]= ref_regblock.protocol_reg2.fieldB.get();
```

Data set( ) with register

Parsed / sliced automatic on fields, retrieve individually with get( )

Determining Register in Operation

- Address used to determine which register is in operation
- Access to specific register may trigger consequential action (clear interrupts, counters, initiate/shut-down)
- Example with hardcoded values

```
case (trans.addr)
   16'h0100 : consequential_action1();
   16'h0101 : consequential_action2();
   16'h0102 : consequential_action3();
endcase
```

- Same vulnerabilities and problems
  - Code will break if addresses change
  - Can produce difficult to find run time bugs in VIP

Determining Register in Operation

- Example using abstraction

```
m_reg = m_ref_regblock.default_map.get_reg_by_offset(my_trans.addr)
if ( m_reg != null ) begin
  m_reg_str = m_reg.get_name( );
end

case (m_reg_str)
    "reg1_reg" : consequential_action1();
    "reg2_reg" : consequential_action2();
         ...
    "regN_reg" : consequential_action3();
endcase
```

'default map' query method returns pointer to register

get name string from reg

work with string to determine register in operation

These strings are like hard-coded values!!!

- Advantages
  - Robust if addresses change
  - But what if the register names change???

Determining Register in Operation

- Abstraction allows self-validating code

```
const string c_reg1_str = "reg1_reg";
const string c_regN_str = "reg2_reg";
string c_register_table[] = { c_reg1_str , […], c_regN_str };

virtual function build_phase(uvm_phase phase);
  foreach(c_register_table[r])
     if (null == m_ref_regblock.get_reg_by_name(c_register_table[r]) )
       `uvm_fatal( … )
endfunction

[…]
  case (m_reg_str)
     c_reg1_str : consequential_actions1(…);
     […]
     c_regN_str : consequential_actions_regN(…);
  endcase
```

Group into array of 'const string'

Go through array contents confirming they are in reference register block

Use 'const string' in case statement

- Another advantage over hardcoded values and `defines

Takeaways

Takeaways

- Use a UVM register block in a VIP to create a layer of abstraction
- Abstraction allows the following:
  - Synchronize with a central register attributes file (register block)
  - Code to be robust and automatically adapt to register changes
  - Eliminate hard-to-find run time bugs
  - Trivial to fix compilation errors, Create self-validating code
- Reduce time lost due to changes in register attributes over the course of a project

verilab

snug
Synopsys Users Group

Thank You

Follow us: *verilab.com*

@verilab