Extending UVM Register Abstraction Layer for Verification of Register Access via Serial Bus Interface

Darko M. Tomušilović
Elsys Eastern Europe d.o.o.
Agenda

- Key requirements
- SPI Universal Verification Component
- SPI Communication Format
- Register Types
- Extension Object
- Power Management
- Summary
Key requirements

• Full device register access verification using UVM Register Abstraction Layer
• SPI used for register access
• Bit-resolution access level
• `supports_byte_enable` of `uvm_reg_adapter` class? Not adequate!
• Complex buffer access mechanism? Not straightforward to implement!
• Reusability for passive operation
SPI UVC

• Master/slave architecture
• Standard 3-wire/4-wire communication
• Configurable clock period, duty cycle, clock polarity, clock phase
• Configurable timings between signals
• Configurable transaction length
• SystemVerilog assertions
uvm_callback - developer implementation

```plaintext
// Abstract callback class
virtual class spi_callback extends uvm_callback;
   `uvm_object_utils(spi_callback)

   // new
   function new (string name = "spi_callback");
      super.new(name);
      endfunction : new

   // Pre-processing - empty task
   virtual task pre_processing (spi_driver driver, spi_transfer trans);
      endtask : pre_processing

   // Post-processing - empty task
   virtual task post_processing(spi_driver driver, spi_transfer trans);
      endtask : post_processing

   endclass : spi_callback

   // Registration of a callback class within a driver
   `uvm_register_cb(spi_driver, spi_callback)

   // Execution of a callback method within a driver
   `uvm_do_callbacks(spi_driver, spi_callback, post_processing(this, trans))
```
SPI Communication Format

- The Header byte
  - Read/write bit + 7 address bits
- Data bits
  - An arbitrary number of data bits can be transmitted
  - UVM documentation recommends byte-level granularity
Register types (1)

• Configuration registers
  – Can be updated by writing bit by bit

• Control registers
  – Must be written completely for changes to take effect
  – Partial writes completely ignored

```c
// Prediction of configuration and control registers
temp_reg = uvm_reg_ext'(reg_model.default_map.get_reg_by_offset(transfer.addr));
temp_data = temp_reg.get_mirrored_value();
if (transfer.kind == UVM_WRITE)
    begin
        if ((temp_reg.reg_type == CONF) ||
            ((temp_reg.reg_type == CTRL) && (DATA_SIZE == temp_reg.get_n_bits())))
            for (int i=0; i<DATA_SIZE; i++)
                begin
                    temp_data[DATA_UPDATE_LSB+i] = transfer.transmit_data[i];
                end
        end
```
Register types (2)

• TX buffer (SPI-to-COM communication)
  – Variable buffer size
  – Write the Header byte
  – Write BUFF_NBITS byte
  – Write data bits
  – Total length calculated in adapter *reg2bus* function
  – Overflow and underflow sequences
Register types (3)

• RX buffer (COM-to-SPI communication)
  – Variable buffer size
  – Write the Header byte
  – Read BUFF_NBITS byte
  – Read data bits
  – Total length not known at the start of a transaction
  – By default, 16 SPI clock cycles (8 - Header, 8 - BUFF_NBITS)
  – The additional clock cycles generated using a callback
  – Overflow and underflow sequences
uvm_callback - user implementation

```verilog
// Concrete callback class
class concrete_spi_callback extends spi_callback;

`uvm_object_utils(concrete_spi_callback)

// new
function new (string name = "concrete_spi_callback");
  super.new(name);
endfunction : new

// Pre-processing - empty task
virtual task pre_processing (spi_driver driver, spi_transfer trans);
endtask : pre_processing

// Post-processing - implementation
virtual task post_processing(spi_driver driver, spi_transfer trans);
// RX buffer read - drive extra bits
...
if ((address == `RXBUFF_0) && (kind == SPI_READ))
begin
  driver.drive_transfer(trans, buff_nbits);
end
...
endtask : post_processing
endclass : concrete_spi_callback
```
uvm_callback - user implementation

```c++
// A concrete callback class object declaration
concrete_spi_callback concrete_spi_cb;

// A concrete callback class object creation
concrete_spi_cb = concrete_spi_callback::type_id::create("concrete_spi_cb");

// Registration of a concrete callback class with the uVC driver
uvm_callbacks #(spi_driver, spi_callback)::add(spi0.agents_i[0].spi_driver_i, concrete_spi_cb);
```
Extension object

- Default or user-modified transaction

```cpp
// Class reg_obj
class reg_obj extends uvm_object;

rand bit use_default_length = 1;
rand bit use_default_timing = 1;

rand int length = 8;

'uvm_object_utils_begin(reg_obj)
...  
'uvm_object_utils_end

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

// Within a register sequence - underflow scenario
// obj.use_default_length = 0;
void'(obj.randomize(length) with
    { length inside [0 : regs[i].get_n_bits() ]; });
regs[i].write(status, data, .parent(this), .extension(obj));

// Within the adapter
function uvm_sequence_item reg2bus (const ref uvm_reg_bus_op rw);
    spi_transfer transfer;
    reg_obj obj;
    uvm_reg_item reg_item;

    transfer = spi_transfer::type_id::create("transfer");
    reg_item = this.get_item();
    $cast(obj, reg_item.extension);

    ...

    if ((obj == null) ||
        ((obj != null) && (obj.use_default_length == 1)))
        // use default
    else
        transfer.num_bits = obj.length;

    ...
endfunction
```
Power management

• Multiple power domains
• Global power monitor
• Register power supply - locking field
• Software reset - \textit{post\_predict()}
• Software power switch - locking field + \textit{post\_predict()}
Results

• All critical scenarios covered
  – Overflow, underflow
  – Power management
  – Reset injection
  – Priority scenarios

• Extensible

• Well encapsulated

• Seamless integration expected
Summary

• UVM_REG beneficial for serial register access
  – Built-in checking mechanism
  – Abstract reusable stimulus
  – Coverage model
• Partial register access
• uvm_callback
• Extension object
• Power supply modeling
• Passive operation
Acknowledgment

The author would like to thank the entire Texas Instruments Freising Security design and management team and Elsys Eastern Europe d.o.o. Belgrade verification and management team for their support during the implementation of aforementioned solutions.
Questions?

Thanks!