2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

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

Darko TOMUŠILOVIĆ, Elsys Eastern Europe d.o.o., Belgrade, Serbia
(darko.tomusilovic@elsys-eastern.com)

***Abstract*—UVM Register Abstraction Layer is a convenient way to model the registers within a hardware design. As the Serial Peripheral Interface (SPI) is used to access register space of the device, the support of bit-resolution access level and complex buffer access mechanism is required. These features are not covered by standard UVM documentation. The proposed solutions can be applied to some other serial bus interfaces and can be easily extended to overcome additional challenges which might be caused by serial access. All proposed solutions support passive operation, which is a critical requirement for the verification process, as it provides reusability.**

Keywords—UVM, Register-Model, SPI

I. INTRODUCTION

One of the keys to successful functional verification of designed logic is verifying proper implementation of registers. First, incorrectly implemented register access policy indicates a faulty designed feature or a missing feature. Second, an erroneous design might result in misalignment between the expected value of a status flag and the value which appears in a status register. Furthermore, a bug pertaining to register operation might reflect upon the incorrect value of output signals, such as interrupt request output. Finally, for the logic which is used to convert payload from one format to another, an error within design might result in an incorrectly converted value appearing in data register, or vice versa, data written to a data register might get corrupted and be falsely driven on the output interface.

In order to meet all the requirements defined in the functional specification of the device, a customized register space with a set of particularities is developed. The device is supposed to be integrated into a Multi-Chip Module (MCM) as a peripheral to a host microcontroller with embedded Serial Peripheral Interface (SPI). That is why SPI is chosen as the interface bus for accessing device registers. SPI provides full-duplex mode communication using master-slave architecture. The device is implemented as SPI slave.

II. VERIFICATION ENVIRONMENT

On verification side, a team of verification engineers is gathered with a goal to verify that the device is designed according to the functional specification. Verification is done using Universal Verification Methodology (UVM) implemented in SystemVerilog. The author of the paper was assigned a duty of developing register model using UVM Register Abstraction Layer (UVM_REG). Other than the register model, test environment includes several main components. For each interface (SPI, COM - custom communication channel, power management related signals), an object of dedicated UVM Verification Component (UVC) is instantiated. As the device acts as SPI-to-COM and COM-to-SPI converter, a scoreboard which compares transactions collected by dedicated interface monitors is built. The prediction of status flags set by hardware is done within register model upon information from interface monitors. Virtual sequencer is used to coordinate activity on multiple interfaces. To generate that activity, a library of virtual sequences is introduced. Finally, the necessary connections between these components are made.

During the development stage, the key is to achieve reusability, as the same environment is to be reused within MCM verification. SPI stimulus is then provided by microcontroller, rather than by active SPI UVC driver, but the passive environment part including checks and coverage collection remains the same.

A. SPI Universal Verification Component

Reusability and configurability are especially emphasized during the development of SPI UVC. As it is meant to be reused across several projects, the UVC is developed to support various SPI configurations:

- Master/slave architecture: UVC is capable to act both as SPI master and SPI slave
- Standard 3-wire/4-wire communication: Synchronous data communication is performed using SPICLK signal, provided by master. Master data are transmitted using SPISIMO signal, while the reception is performed using SPISOMI signal. Optional SPICSN signal can be used as slave selection signal
- Randomly generated timings between signals: SPICSN-to-SPICLK lead time, master output SPISIMO data setup/hold time, SPICLK-to-SPICSN end delay time, pause between consecutive transactions
- Randomly generated clock configurations: SPICLK period and clock duty cycle, configurable clock polarity and clock phase
- Configurable baud rate
- Configurable transaction length
- Automatic detection of transaction length using information received on master input SPISOMI line

The last feature is implemented using UVM callback mechanism. Pre-processing and post-processing callback methods are provided as part of an abstract callback class which is registered with UVC driver. The definition of abstract callback class is shown in **Figure 1**. To register the class with UVC driver, the code shown in **Figure 2** should be used within a driver. End user is allowed to adapt driver operation without interfering with the UVC. A callback method is invoked before and after the driver executes its basic operation on the item provided by the sequencer. An example of calling a callback method is given in **Figure 3**.

```
virtual class spi_callback extends uvm_callback;

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

  virtual task pre_processing (spi_driver driver, spi_transfer trans);
  endtask

  virtual task post_processing(spi_driver driver, spi_transfer trans);
  endtask

endclass : spi_callback
```

Figure 1: Definition of an abstract callback class

```
`uvm_register_cb(spi_driver, spi_callback)
```

Figure 2: Registration of a callback class within a driver

```
`uvm_do_callbacks(spi_driver, spi_callback, post_processing(this, trans))
```

Figure 3: Execution of a callback method

In addition to standard components, UVC also incorporates a number of configurable SystemVerilog assertions. All of the above features proved useful during the process of verification of register space of the device.

B. SPI Communication Format

The specification of the device defines a particular format of SPI low-level communication. SPI is only active if the low-active chip select signal is active (SPICSN = 0). The synchronous data communication uses a clock (SPICLK) and two data lines. The clock is provided by the master. The master output data line (SPISIMO) serves for data transfer from UVC to the device, whereas the slave output data line (SPISOMI) serves for data transfer from the device to UVC. On both data lines, particular bits of data are output on a rising edge of SPICLK and data are captured on a falling edge of SPICLK. The most significant bit is transmitted first. The first data bit is driven on the first rising edge of SPICLK after the chip select signal becomes active. The format of SPI communication is shown in **Figure 4**.

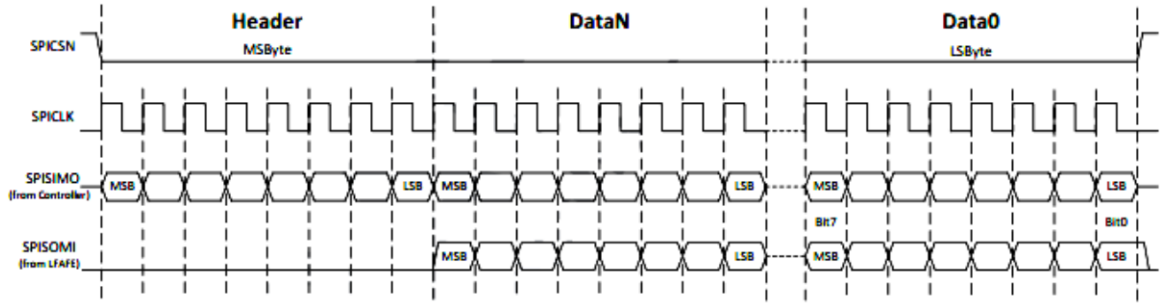

Figure 4: SPI communication format

The first byte after SPICSN falling edge is always the Header byte, which determines whether a read or a write access is being performed and which register is being accessed. Bit RWMODE determines if access is a read or a write operation (0b = read access, 1b = write access), while the remaining 7 bits determine the address, which is shown in **Table 1**.

Table 1: Header byte format

7	6	5	4	3	2	1	0
RWMODE	ADDRESS						

In case of read access, after the master sends a Header byte with selected read mode, the device will respond with the addressed register content. After Header byte reception, SPISIMO line is ignored by the device until chip select does the SPI communication reset (rising edge on SPICSN).

After the master sends a Header byte with write mode selected, the device will receive data and write them to the register being addressed. SPISOMI will be held deactivated throughout write accesses. When the write access is accomplished, chip select must be disabled (rising edge on SPICSN) before a new Header byte can be issued.

Both in case of write and read operations, an arbitrary number of data bits can be transmitted following the Header byte. That represents the most challenging task for verification side. The UVM documentation suggests the built-in field *supports_byte_enable* of *uvm_reg_adapter* class to read or write individual bytes in multi-byte bus [1]. However, in that case, the granularity within a byte is maintained, which collides with the verification requirements.

C. Register Types

Seven various types of registers are implemented and almost each of them requires an unconventional solution in order to be verified:

1. **Configuration registers** support read and write accesses. Registers can be written bit by bit, the most significant bits are written first. Depending on the fact whether a bit is written or not, a combination of

previously stored mirrored value and the value extracted from the latest transaction is used for prediction of new mirrored value. The mechanism of prediction of configuration registers is shown in **Figure 5**.

2. **Control registers** support read and write accesses. Registers must be written completely for changes to take effect, while partial writes are completely ignored. In adapter, it is checked whether the size of transaction matches register size. In case it does, new value is used for prediction, while, in the opposite case, the previously mirrored value is kept. The mechanism of prediction for control registers is shown in **Figure 5**.

```
temp_reg = uvm_reg_ext'(lfafe_rm.default_map.get_reg_by_offset(transfer.addr));

temp_data = temp_reg.get_mirrored_value();

// Partial access is possible in case of configuration register access;
// in case of access to control register, DATA_SIZE must match register size

if (transfer.kind == UVM_WRITE)
begin
  if (  (temp_reg.reg_type == CONF) ||
       ((temp_reg.reg_type == CTRL) && (DATA_SIZE == get_register_bit_size_by_offset(transfer.addr))))

  for (int i=0; i<DATA_SIZE; i++)
    begin
      temp_data[DATA_UPDATE_LSB+i]=transfer.transmit_data[i];
    end
end
```

Figure 5: Prediction of configuration and control registers upon write access

3. **Status registers** are read-only. The bit-fields within their physical implementation are directly controlled in hardware, while they are modeled based upon the information collected by interface monitors.
4. **Trigger registers** support read and write accesses. Registers can be written bit by bit, but they are cleared automatically after the access. Their role is to trigger activities. Therefore, they are modeled as read-only, but depending on the written value, a number of events are triggered in order to signalize the environment that an activity commences.
5. **The Interrupt request source register** is read access only. Its content clears upon read. It can be read bit by bit without reading the whole register, in which case only the read bits are affected. Its behavior is modeled in *post_predict* callback method by a combination of previously predicted value and a flag which indicates whether a bit field is accessed during the latest transaction. The callback method is a part of pre-defined *uvm_reg_cbs* callback class.
6. **Map registers** represent a command mapped to a register address. If the address is contained in the Header byte, the mapped function will be executed. There is no actual register content behind the address. They are modeled as unimplemented registers.
7. **Buffer registers** represent a special case of SPI communication protocol, since the size of the buffer content is not fixed. The buffers are accessed like a regular register. The Header byte contains the buffer's address, as well as the read/write bit to determine whether transmission (TX) or reception (RX) buffer is accessed.
    a. For SPI-to-COM communication, TX buffer is used. After the Header byte, the following written byte BUF_NBITS determines the number of bits to write to TX buffer. Total transaction length is calculated in adapter *reg2bus* function, based on the value programmed into BUF_NBITS field. Sequences for buffer overflow and underflow are also provided.
    b. For COM-to-SPI communication, RX buffer is used. After the Header byte, the succeeding read byte BUF_NBITS determines the number of bits to read from RX buffer. That means that at the start of SPI transaction, its total length is not known. Instead, it should be extracted from the first received byte. By default, in case of buffer read, 16 SPI clock cycles are provided, 8 for Header byte and 8 for BUF_NBITS. To generate additional number of clock cycles required to read the buffer content, the callback mechanism is used. After driver completes its operation and generates 16 SPI clock cycles, further processing is implemented as part of *post_processing* callback method. There, additional number of SPI clock cycles is calculated using the first received byte. In order to support this, it is necessary to create an object of a class derived from an abstract callback class which is a part of the UVC. The callback class used is shown in **Figure 6**. The concrete class object declaration and creation are shown in **Figure 7** and **Figure 8**, respectively. It is also required to register the concrete class with a

relevant UVC driver. The registration is shown in **Figure 9**. Sequences for buffer overflow and underflow are also provided.

```
class lfafe_spi_callback extends spi_callback;

  `uvm_object_utils(lfafe_spi_callback)

  // new - constructor
  function new (string name="lfafe_spi_callback");
    super.new(name);
  endfunction : new

  // Empty task
  virtual task pre_processing (spi_driver driver, spi_transfer trans);
  endtask

  virtual task post_processing(spi_driver driver, spi_transfer trans);
    // Automatically prolong SPI transaction in case that LFIMMOFIFO is read:
    // in that case, total number of necessary read bit cycles is determined during transaction,
    // according to second read byte: field N_BITS

    .
    .
    .

    if ((address == `LFIMMOFIFO_0) && (kind == SPI_READ))
      begin
        driver.drive_transfer(trans, buff_nbits);
      end

    .
    .
    .
  endtask

endclass : lfafe_spi_callback
```

Figure 6: Definition of a concrete callback class

```
lfafe_spi_callback    lfafe_spi_cb;
```

Figure 7: A concrete callback class object declaration

```
lfafe_spi_cb = lfafe_spi_callback::type_id::create("lfafe_spi_cb");
```

Figure 8: A concrete callback class object creation

```
uvm_callbacks #(spi_driver, spi_callback)::add(spi0.agents_i[0].spi_driver_i, lfafe_spi_cb);
```

Figure 9: Registration of a concrete callback class with UVC driver

D. Extension Object

For registers which are of type [1:5], in case of regular register write or read, the size of transaction will be calculated in adapter *reg2bus* function, using register size. However, in order to support overflow and underflow scenarios, which are critical for proper register access verification, additional mechanism is provided using UVM pre-defined extension object. The object can be added as an argument of register read and write methods, in order to support user-modified transactions. An example of an extension object is shown in **Figure 10**. Field *use_default_length* is used to signalize adapter that rather than the default value calculated using register size, the transaction size provided by the end user should be used.

```
class lfafe_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(lfafe_reg_obj)
      `uvm_field_int(use_default_length, UVM_DEFAULT)
      `uvm_field_int(use_default_timing, UVM_DEFAULT)
      `uvm_field_int(length,             UVM_DEFAULT)
  `uvm_object_utils_end

  function new (string name = "lfafe_reg_obj");
    super.new(name);

  endfunction : new

endclass : lfafe_reg_obj
```

Figure 10: Register access extension object

In the code extract shown in **Figure 11**, a random number of bits between 0 and register size is written to a register to ensure proper access policy of configuration and control registers. The extension is handled in *reg2bus* function and the corresponding SPI transfer size is configured. That is shown in **Figure 12**.

As part of the extension object, the option whether to generate transaction timings according to nominal values provided by the specification or using user-defined values is present as well, using field *use_default_timing*.

```
obj.use_default_length = 0;
void'(obj.randomize(length) with { length inside { [0 : lfafe_regs[i].get_n_bits()] }; });
lfafe_regs[i].write(status, data, .parent(this), .extension(obj));
```

Figure 11: Writing a random number of bits to a register

```
// Create SPI item according to reg access item
function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
  spi_transfer transfer = spi_transfer::type_id::create("transfer");
  lfafe_reg_obj      obj;
  uvm_reg_item  reg_item;

  reg_item = this.get_item();
  $cast(obj, reg_item.extension);

  .
  .
  .

  // Determine length of transfer
  // In case that extension is not provided, use default length
  if ((obj == null) || ((obj != null) && (obj.use_default_length == 1)))
  .
  .
  .
  else   // Extension provided: drive number of bits given in extension
    transfer.num_xfers = obj.length;

  .
  .
  .
endfunction
```

Figure 12: Handling of extension object within adapter

E. Application

The basic application algorithm is:

1. Enable the COM-to-SPI and SPI-to-COM communication using the dedicated control register
2. The device stores the data received via COM channel into the RX buffer
3. An interrupt is issued when the device detects the end of the reception - reading the Interrupt request source register clears the interrupt status flag
4. The status of the COM communication is indicated using status registers
5. The received data is read out from the RX buffer
6. Configure the SPI-to-COM communication format using the configuration registers
7. Write the TX buffer
8. Trigger the start of the SPI-to-COM communication using the dedicated trigger register
9. The device transmits the data written to the TX buffer via COM channel
10. An interrupt is issued at the end of the transmission - reading the Interrupt request source register clears the interrupt status flag

All communication scenarios require thorough register verification based on the standard or custom register type.

F. Power Management

As a means of reduction of power consumption, registers are placed across multiple power domains. Based on input signals (voltage, hardware power switch), a global power monitor predicts whether a register is under reset. In that case, the register behaves as read-only and reading the register results in read value isolation. The feature is implemented using locking field [2]. Software reset functionality is implemented using *post_predict* callback method. In that way, it is provided that writing the expected value to a dedicated field of software reset register will reset all registers within relevant functional block. Software power switch, which cuts off certain power domains by writing correct password to a dedicated register, is also modeled using locking field and *post_predict* callback method.

G. Register Sequences

The main role of register model is to facilitate development of reusable sequences that access hardware registers. Therefore, numerous sequences are provided as part of register sequences library. Some of them are oriented towards verification of register access policy (read-write sequences, overflow and partial access sequences), while the others are developed to be used as part of functional test cases (configuration sequences, buffer access sequences, sequences used to trigger activity, sequences used to check status flags). The latter not just reduce the effort of development of functional test cases, but also eliminate the redundant code, which in turn decreases the room for errors.

H. Register Coverage

In order to enhance functional coverage, UVM built-in register model coverage is included. To provide the option of conditional deferred construction, covergroups are wrapped within *uvm_object*. That also gives option to override covergroup wrapper class using the factory. The solutions given in [1] are used.

III. Preliminary Results

The main result of the solution described in this paper is thorough verification of complex register space. All the critical scenarios are covered, including those related to power management, reset injection, overflow and underflow accesses. Priority scenarios, such as triggering of an interrupt while interrupt clear register is accessed, are also handled. The solutions proved extensible, as some of the requirements given at later stage of the project were quickly implemented and required minor register model update. The whole verification process is done with high quality, meeting the expected deadline. One major advantage of the used approach is that all register related requirements are well encapsulated. Test case writers who write functional test cases are provided a simplified interface to registers through register sequences library. That makes test cases more readable, easier to write and to maintain. The next task which is under development is to reuse the verification

environment in passive context within top-level environment. Seamless integration is expected as the environment is developed with a view to achieving reusability.

IV. CONCLUSIONS AND RELEVANCE OF THE PAPER

The conclusion that can be drawn is that UVM Register Abstraction Layer is very beneficial even in case a serial bus interface is used for register access. It provides a mechanism of generating abstract and reusable stimulus through register sequences. It can be used to track the correctness of designed logic through built-in checking mechanism, while also giving contribution to functional coverage model. However, a serial access to registers brings a set of challenges which need to be overcome in order to achieve comprehensive verification. The paper proposes solutions of handling partial register access. It is shown how *uvm_callback* mechanism can be utilized to handle highly project-specific features. It is also explained how extension object can be used in order to give user option whether to use default configuration settings or to generate modified transactions. Furthermore, it is shown how some standard UVM_REG mechanisms, such as locking field and *post_predict* callback method can be used to model power supply. Finally, the paper highlights the importance of development of extensible and reusable code, suited for passive operation. While the solutions are elaborated for SPI, similar mechanisms can be applied in case some similar serial bus interface, such as I^2C, is used.

ACKNOWLEDGMENT

The author would like to thank the entire Texas Instruments Freising Security design and management team for their support during the implementation of aforementioned solutions.

REFERENCES

[1] Accellera, “UVM User Guide, v1.1”, www.uvmworld.org

[2] Mark Litterick and Marcus Harnisch, “Advanced UVM register modeling – There’s more than one way to skin a reg”, Verilab & DVCon Europe 2014.