Improving the UVM Register Model: Adding Product Feature based API for Easier Test Programming

Krishnan Balakrishnan, Courtney Fricano, and Kaushal Modi
Analog Devices, Inc.
3 Technology Way
Norwood, MA 02062

Abstract – It is much easier to program a complex device if the device provides an interface that clearly highlights its feature set, be it in a Universal Verification Methodology (UVM) environment during design verification, a Graphical User Interface (GUI) during evaluation, or a software Application Programming Interface (API) for customer use. An interface that allows the user to program features of the device directly, rather than having to spend the time to figure out how the feature has been implemented in the device is extremely desirable. To be used effectively, this interface would have to be very simple, so it is preferable that the features of the device be accessed without traversing multiple layers of hierarchies within the interface.

In this paper, we describe a product feature based API integrated into the UVM register model that allows test patterns to be written more easily by accessing product features directly rather than traversing the register layer hierarchy. This API also inherently provides a simpler way of programming product features that span multiple registers. The API coexists with existing UVM register environment allowing test patterns to use a mix of feature based or register based API's. The patterns using the feature based API are easier to code, better documented and highly portable across projects.

Keywords – UVM, register, programming, sequences

I. NEED FOR PRODUCT FEATURE BASED API:

The current UVM register model specifies product features as fields (uvm_reg_field) contained within registers (uvm_reg), structured under hierarchy(s) of blocks (uvm_reg_block) within the UVM register model. This is a hardware centric view of the device. The user, programming the device, is manually required to map a product feature into bit position(s) of field(s) in register(s). If the user needs to configure a product feature that spans multiple registers, the user is required to manually split the configuration value into multiple register writes. Also, multiple register reads need to be combined manually to read back the status of a feature.

This programming methodology is cumbersome and prone to errors due to the manual effort involved. Moreover, the code being written would be quite hard to understand since register names / addresses and bit positions are used in the code instead of the feature name. The test pattern would be fragile as it becomes susceptible to failures if the number of bits used to encode the feature changes, if the bit position of the feature within a register changes, or if the feature is moved to a register with a different address. If a feature is removed, the test pattern may still compile indicating a 'false pass' if a register name based API is used in programming.

A user programming the device (through sequences / GUI / software) would prefer a feature-based API where the features of the product are accessible directly, rather than having to traverse the register hierarchy(s) to figure out the mapping from feature to fields. A feature based API with an interface that is immune to changes in the implementation of the feature will allow patterns using this API to be extremely robust. Also, if a feature is removed, using a feature based API will highlight the pattern (as a compilation error, instead of a 'false pass') that the feature was tested in.

II. PRODUCT FEATURE BASED API USING BITFIELD

The API presented in this paper resolves the hurdles and manual overhead in the existing UVM register model through a simpler usage model that is based on product features (termed 'BitField' and 'adi_bitfield' in the paper) while co-existing with the current UVM model. The product feature based API is implemented using the following new classes:

"adi_bitfield" class:

- The feature of the device (BitField) is specified through this class. The feature can be of arbitrary widths, and are not restricted by the width of a register.
  - Example: 11 bit 'pll_f'
  - Example: 1 bit 'pll_pd'
  - Example: 2 bit 'ch_enable'.
- The class contains methods like set, get, update, write, read and compare similar to the uvm_reg_field class.
- Since product features are finally implemented in silicon using fields in registers (uvm_reg_fields), the mapping from adi_bitfield to uvm_reg_field within the current UVM environment is implemented in the class.
  - An array 'slice' in the class allow each adi_bitfield to be mapped to one or more uvm_reg_fields across one or more uvm_reg objects.
  - Each array entry is a structure containing a handle to the uvm_reg_field, a Least Significant Bit (LSB) position, and a Most Significant Bit (MSB) position, to maps the feature into multiple uvm_reg_field objects.
  - The class also contains a handle back to its parent to allow a non-iterative implementation of the 'update' method used in the 'adi_bitfield_block' class.

```
class adi_bitfield extends uvm_object;

  typedef struct {                    // Structure containing pointers to the position of the bitfield slice
    uvm_reg_field ptr;                // Handle to the sub-bitfield slice
    int lsb;                          // LSB of the slice in the bitfield
    int msb;                          // MSB of the slice in the bitfield
  } slice_t;

  // Members
  slice_t slices[int];                // Collection of pointers to support bitfields spanning multiple registers
  adi_bitfield_block m_parent;        // Pointer to parent fields block

  // Methods
  new(..);
  configure_slice (..);               // Configure slice function sets the ptr, MSB, and LSB fields for a single slice
  set(..);                            // Set the desired value for this field by iterating through the slices
  uvm_reg_data_t get (..);            // Get the desired value for this field by iterating through the slices
  bit needs_update(..);               // Check if the abstract model contains different desired and mirrored values.
  update (..);                        // Update the content of the field in the design to match the desired value
  write (..);                         // Write the specified value in this field
  read (..);                          // Read the specified value in this field

endclass
```

Figure 1: adi_bitfield class

"adi_bitfield_block" class:

- The class is a container for all adi_bitfield objects. The object of this class is created at the root of the UVM register model hierarchy to be easily accessible to the user.
- This class contains a non-iterative implementation of the 'update' method.
  - The method causes register writes only for adi_bitfield objects that have been modified using the adi_bitfield's 'set' method by using an array of adi_bitfield handles. When an adi_bitfield's set method is invoked

it adds itself to this array. When the 'update' method is invoked, it will only iterate through this array (instead of iterating through all adi_bitfield objects).

- This is very efficient compared to the update method in the uvm_reg_block class that iterates through all registers (and causes a write in registers with 'volatile' fields even if the value of the register has not changed).

```
class adi_bitfield_block extends uvm_object;

  // Members
  adi_bitfield modified_fields[string];   // Array of fields modified through the 'set' method

  // Methods
  new(..);
  set_modified_field (..);     // adi_bitfield's 'set' calls this to add itself to the modified_fields array
  update (..);      // Iterates through modified_fields, and calls 'update' only for adi_bitfield objects in array

endclass
```

Figure 2: adi_bitfield_block class

This API maintains flow control at the product feature level. For a product feature that spans multiple registers, a 'write' to the feature can cause writes to its related registers to happen in any order. For finer flow control, the existing UVM register based methodology can be used.

III. BITFIELD BASED ENVIRONMENT:

The BitField based API is quite concise especially for product features that span multiple registers for methods like set, get, write, read etc. It also requires less effort to code as it obviates the need of knowing register names or how a feature is split across multiple registers. Figure 3 shows an example where three product features are distributed across three registers. The feature 'pll_f[10:0]' is split across two registers ('pll_f_config1' and 'pll_f_congif2'). The feature 'pll_pd' is also packed into the 'pll_f_config1' register. These two features are within the 'pll_settings' RegisterMap. The third feature 'ch_enable[1:0] is in register 'cfg_channel' within the 'dev_settings' RegisterMap. The existing UVM environment is marked in gray, and the BitField API in blue.

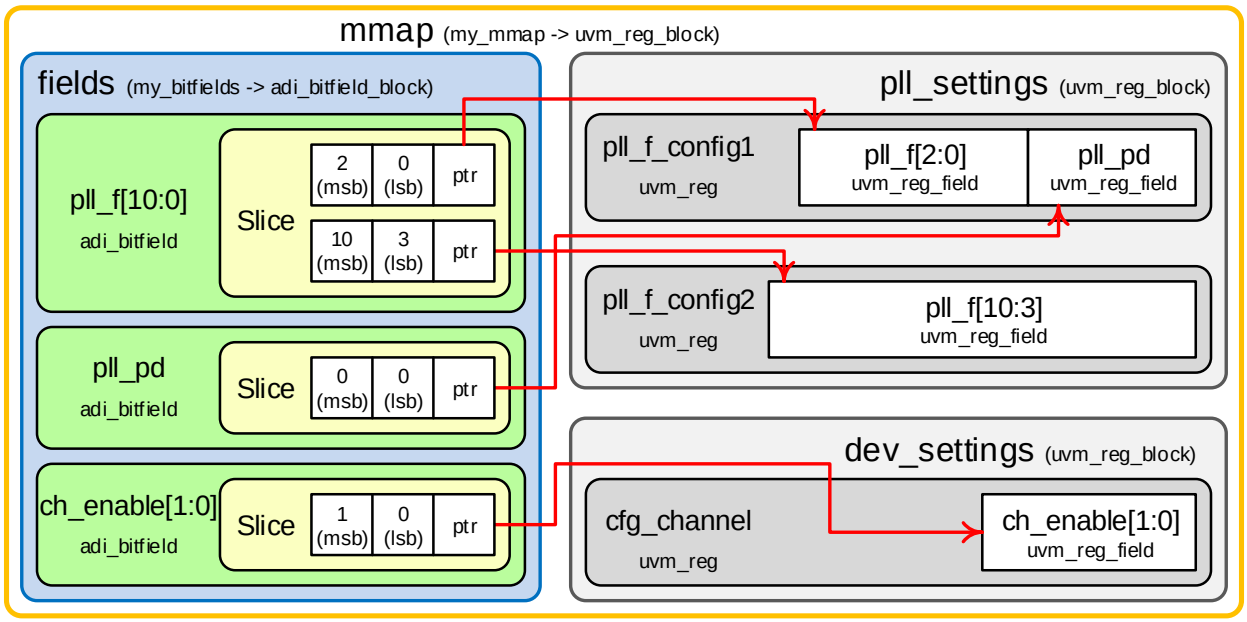

Figure 3: BitField API environment

The mappings between the adi_bitfield objects and the uvm_reg_field objects are created as shown in Figure *4* below. The my_bitfields type is a class extended from adi_bitfield_block containing the adi_bitfield objects for a specific design, and the 'configure' method is used to set the BitField mapping to each register. The my_mmap type is a class extended from uvm_reg_block that sets up the register and BitField models for the design, as shown in Figure 3.

```
// extend the adi_bitfield_block class to add the BitFields and a function to configure them
class my_bitfields extends adi_bitfield_block;

  // BitFields
  adi_bitfield ch_enable;
  adi_bitfield pll_f;
  adi_bitfield pll_pd;

  // The configure task sets up the mapping between BitField and the uvm_reg_fields
  function void configure (my_mmap blk_parent);

    // map ch_enable
    ch_enable = adi_bitfield::type_id::create("ch_enable");
    ch_enable.configure_slice(this, blk_parent.dev_settings.cfg_channel.ch_enable);
    // map pll_f
    pll_f = adi_bitfield::type_id::create("pll_f");
    pll_f.configure_slice(this, blk_parent.pll_settings.pll_f_config2.pll_f, 10, 3);
    pll_f.configure_slice(this, blk_parent.pll_settings.pll_f_config1.pll_f, 2, 0);

    // map pll_pd
    pll_pd = adi_bitfield::type_id::create("pll_pd");
    pll_pd.configure_slice(this, blk_parent.pll_settings.pll_f_config1.pll_pd);
  endfunction

endclass



// Class for the memorymap
class my_mmap extends uvm_reg_block;

  // The RegsiterMaps
  rand my_dev_settings dev_settings;
  rand my_pll_settings   pll_settings;

  // The Bitfields environment
  rand my_bitfields fields;

  virtual function void build();
    // Build the RegisterMaps as in existing UVM environment
    ...
    // Build the BitFields environment
    this.fields = my_bitfields::type_id::create("fields");
    this.fields.configure(this);
  endfunction

endclass
```

Figure 4: BitField API environment code

The code snippet in Figure 5 show an example test pattern using the register coding style for the existing UVM methodology. This requires the user to manually map features to fields and are susceptible to errors when the underlying implementation of the feature changes.

```
// pll_f = 11'b10101010101
// pll_pd = 1'b0
// ch_enable = 2'b11

uvm_status_e status;

// Set pll_pd
mmap.pll_settings.pll_f_config1.pll_pd.set      (1'b0);

// Set pll_f
// The value of pll_f has to be split manually by user into the two uvm_reg_field set methods.
mmap.pll_settings.pll_f_config1.pll_f.set       (3'b101);
mmap.pll_settings.pll_f_config2.pll_f.set       (8'b10101010);

// Set ch_enable
mmap.dev_settings.cfg_channel.ch_enable.set (2'b11);

// Update mmap. Not ideal as it iterates through all registers (and writes all registers with volatile fields)
mmap.update(status);
```

Figure 5: Existing UVM API

The code snippet in Figure 6 below uses the BitField based API. This API presents the user with easy access to product features, produces better self-documentation of the pattern, and provides extremely robust code that is immune to changes in implementation of the product feature.

```
// pll_f = 11'b10101010101
// pll_pd = 1'b0
// ch_enable = 2'b11

uvm_status_e status;

// Set pll_pd
mmap.fields.pll_pd.set       (1'b0);

// Set pll_f.
mmap.fields.pll_f.set        (11'b10101010101);

// Set ch_enable:
mmap.fields.ch_enable.set  (2'b11);

// Update all modified features.
mmap.fields.update(status);
```

Figure 6: Bitfield based API

IV. AUTOMATING BITFIELD TO UVM_REG_FIELD MAPPING

Analog Devices, Inc. (ADI) primarily uses an internally developed database for register specifications. ADI believes that the BitField is the principal element as it specifies the feature set of the device, and that the Register Abstraction Layer (RAL) is only one of several mappings to structure BitFields. Other mappings possible using the internal database include grouping BitFields in a format that can generate a GUI interface, or grouping them in a format that can generate higher level software API's, or a format that can generate internal and customer documentation. These other mappings are equally relevant as the RAL, as they improve the Time-To-Market (TTM) for ADI's products and provide great value addition for ADI's customers.

ADI believes that the IP-XACT specification, developed by the SPIRIT consortium, is not easily amenable to be able to provide the GUI and software API abstractions that benefit ADI's customers. The IP-XACT specification specifies a hardware-centric view of the device (Fields within Registers), where the features of the device are hidden under the RAL, so it does not provide a mapping between product features and fields, especially for product features (BitFields) that span multiple registers. ADI recommends using vendor extensions within IP-XACT for specifying BitFields to enable mapping between a BitField and one or more Fields within Registers, as shown in Figure 7 below.

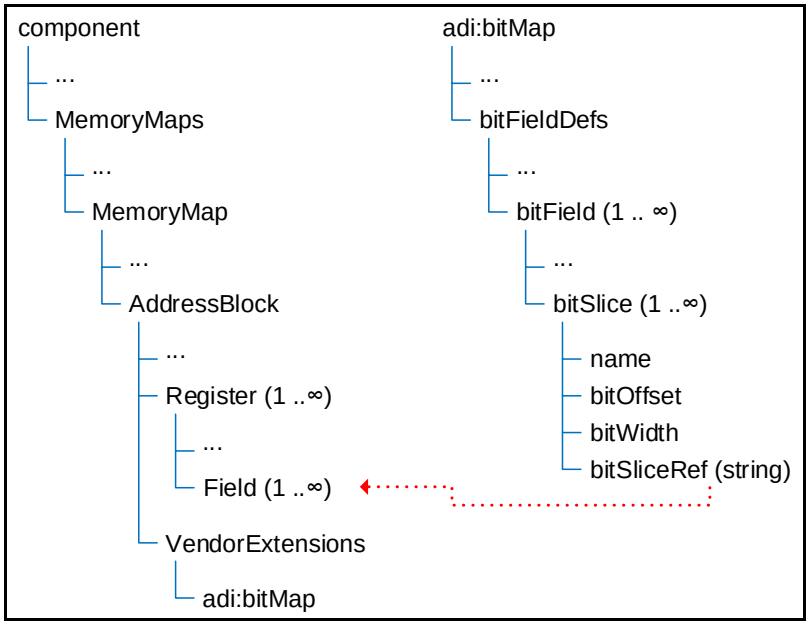

Figure 7: IP-XACT vendor extensions

ADI's internal tools are based on an Extensible Markup Language (XML) database with a XML Schema Definition (XSD) that is very similar to ADI's recommendation for IP-XACT shown above in Figure 7. A major difference from the IP-XACT specification is that all attributes of the product feature (Access type, Volatile, Enumeration, Datatype etc.) that are required for the various mappings like RAL, GUI and API reside with the BitField, and not with the field inside registers. For example, attributes for the RAL include the 'bitSlice' that links portions of the BitField to fields within registers in the Register model of the memory map. An Enumeration attribute can be used by RAL, GUI and API mappings. A User Interface Type (UItype – Checkbox / Slider / Textbox etc.) can be used by the GUI mapper. All BitFields in the schema have a global scope in the memory map.

ADI uses an internally developed GUI software overlaying the XML database that allows the user to enter product features into the database as BitFields. For a RAL mapping, the GUI provides an easy interface to slice the BitField into multiple registers, and updating the 'bitSlice' attribute within the underlying XML database. Linking from Bit-Field to a register field, rather than the other way around, ensures that the attributes of all register fields that are slices of a BitField are consistent with the attribute of the BitField.

The software also contains many generator scripts (similar to other tools used in the industry that generate code from the IP-XACT XML) that parses ADI's XML database to create Register Transfer Layer (RTL) code for the device, and the UVM setup for the BitField API environment for the verification test bench. The adi_bitfield to uvm_reg_field mapping (the my_bitfields and my_mmap classes) are automatically generated by the script during this process. ADI believes that a similar infrastructure can be developed based on the IP-XACT specification using vendor extensions, and recommends that all attributes of the product feature reside with the BitField so it can be used effectively by multiple mappers.

V. CONCLUSIONS

A BitField based programming API provides several advantages over the standard UVM Register based model for designs with feature-oriented fields. A fully-automated process has been developed to enable BitField API in a UVM testbench. The BitField API can be fully integrated into the testbench, coexisting with the current UVM environment and allowing test patterns be written using the BitField API, the legacy Register API, or a mix of the two methodologies. The test patterns using the BitField API are easier to code, better documented, and immune to the BitField being moved from one register to another during the course of the project.