Problem Statement

- Users prefer an API that clearly highlight the feature of the device being programmed.
  - UVM: Design verification
  - GUI: Lab evaluation
  - Software: Firmware and customer
  - Specifying product features as fields in registers hides implementation details of the product features.
- The IP-XACT specification, and the UVM environment present a hardware centric view of a device.

Solution – Product feature based API

- API is independent of the implementation of product features as field(s) within register(s).
  - Obviates user errors and improves readability and portability.
  - The API coexists with the existing UVM model.
- Software features are implemented as BitFields ('adi_bitfield' objects).
- All BitFields are held in a container class ('adi_bitfield_block')
  - The object for this class is created at the root of the UVM memory map for easy access to product features.

- The Class implements a product feature (BitField)
  - Can have arbitrary length, not restricted by register widths
  - Contains methods similar to uvm_reg_field
    - Set: get, update, write, read
    - Contains an array of 'slices' that maps portions of the product feature to uvm_reg_field(s) across uvm_reg_object(s).
    - Each member of the 'slice' array is a handle to a uvm_reg_field, a MSB position, and a LSB position that defines the mapping

Example of register based environment

- Device contains three product features - pll_f[10:0], pll_pd, and ch_enable[1:0] in three 8-bit registers.

Programming the device in UVM environment

- Programming Impediments
  - The user needs to have a detailed knowledge of the implementation details of the product features.
  - Complicated coding style: User needs to traverse the register layer hierarchy of the device to access a feature.
  - Manual overhead: Configuration/Read-back of product features that span multiple registers is a manual process (splitting feature into multiple fields/merging multiple fields into a feature) that is very prone to user errors.
  - Reduced Portability: Code is not easily portable as it is susceptible to changes in field width, position, or moving the field to a different register.

BitField based environment

- 'adi_bitfield' class
  - The class implements a product feature (BitField)
  - Can have arbitrary length, not restricted by register widths
  - Contains methods similar to uvm_reg_field
    - Set: get, update, write, read
    - Contains an array of 'slices' that maps portions of the product feature to uvm_reg_field(s) across uvm_reg_object(s).
    - Each member of the 'slice' array is a handle to a uvm_reg_field, a MSB position, and a LSB position that defines the mapping

- 'adi_bitfield_block' class
  - The class is a container for all 'adi_bitfield' objects
    - Instance of this class is created at the root of the UVM memory map to be easily accessible to the user API.
  - This class contains a non-iterative implementation of the 'update' method.
    - The method updates only the 'adi_bitfield' objects that have been modified using the 'adi_bitfield's set method'.
    - When an 'adi_bitfield's set method is invoked it adds itself to the 'modified_fields' array. The 'update' method in this class only iterates through this array, and not through all BitFields.

- Programming using the BitField API
  - API accesses product features directly
    - Code is independent of implementation.
    - No manual overhead splitting/merging features to fields.
    - Portable and understandable code

Conclusion

- BitField based API has several advantages over Register based UVM environment
  - Integrated and co-existing with existing UVM environment
  - Fully automated process to generate BitField environment
  - Test patterns using the BitField API are easy to code, better documented, and immune to changes in the BitField address, width or position during the course of the project.