2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

FLEXIBLE INDIRECT REGISTERS WITH UVM.

Uwe Simm, Cadence Design GmbH, Feldkirchen, Germany, uwes@cadence.com

***Abstract*: The paper outlines a generic extension to the UVM library simplifying the handling and definition of 'indexed' registers. Index registers are common in today's hardware especially when the address space is small and/or data structures are replicated in the address space. The outlined approach is flexible enough to accommodate custom index definitions such as wildcard indexes or patterns, it allows for custom entities providing the actual index value(s) and it also allows for a user defined 'indexable' structure such as register fields, registers or memory.**

I. INTRODUCTION

The current implementation of the UVM library [1] provides a set of bases classes that are in use by everyone plus a set of very special base classes that are only useful under special circumstances. The library does only in a very few places define a full and well defined API through which users can accomplish their extended needs (aka corner cases) via extensions. Up to UVM1.2 it is usually a question if a functionality is in the library or not – with the upcoming IEEE release it should be a question if the library provides an API so that the functionality can be accomplished using the provided API. A proper base class library should expose an API that it allows to accomplish all use models but it is not necessary that all use models have a concrete implementation in the base library.

This paper outlines that with a proper separation of concerns and structuring things get simpler, more flexible and more powerful. As an example, the paper focuses on indexed registers. UVM provides one implementation for a very special variant ('int' index derived from uvm_reg indexing an array of uvm_reg elements) instead of providing a structure that allows the user to define what an 'index' is and what the 'storage' looks like. As a result, users with slightly different 'indirect' setup must reinvent the whole infrastructure of 'indirect' again. The paper illustrates that with slightly more infrastructure in the library almost arbitrary indirect configurations can be accomplished easily.

II. STRUCTURING ENGINEER'S WISHES

When talking to engineers and asking what kind of indirect register they use or require you hear multiple different answers. To list a few:

- "My index is a register field indexing an array of registers."
- "I have a 3-dim array with 3 indicies stored in 3 uvm_reg elements."
- "The lower bits of my index are dervied from one uvm_reg_field while the upper bits of the index are stored in a different uvm_reg_field …"
- "The index is actually a mask and selects a set of registers to write…."

The list is obviously open-ended and it is impossible to agree or include support for all possibilities into a base class library.

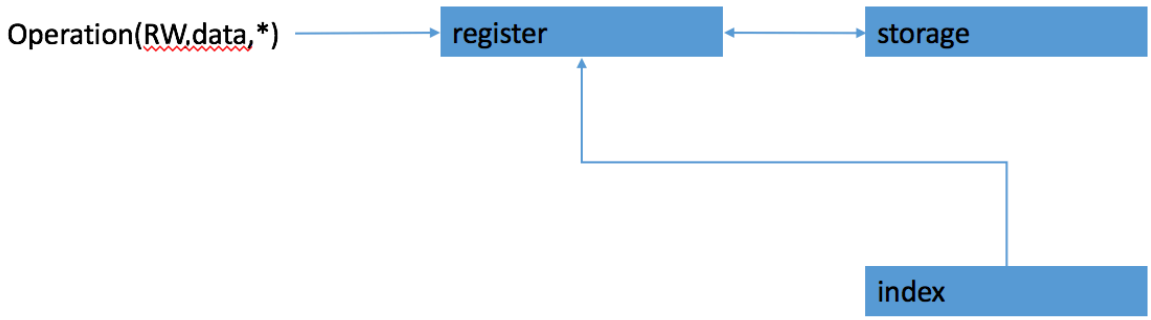

Figure 1 relation between main solution elements. The index is used by the register and the register can set or get the data from the storage.

III. FROM WISHES TO ENCAPSULATED OBJECTS

Looking carefully at the provided examples one can see that the requested properties fall into two categories:

- What is an "Index" and who provides the index?
- What is the data model or "storage" that is being indexed?

Figure 1 above depicts the indirect register approach used separating the main conceptual building blocks of the solution. The storage box in the figure provides "storage" or data model related elements while the index box in the picture depicts the "index" related elements.

Following the design pattern suggestion advice in [2] to encapsulate all "changing" or "unknown behavior" in a class of its own we encapsulate everything related to the first category into an "Index Provider" and the second category into a "Storage Provider" class. This encapsulation separates all the functionality we can define already from the unknown functionality that we cannot foresee and we want to make available to the user. At a later point in time users should be able to provide their implementation of StorageProvider and IndexProvider based on their actual hardware requirements. The general capability of the indirect register however can be encapsulated in a special GenericIndirectRegister class, which will coordinate storage and index and the normal register behavior.

IV. FROM OBJECTS TO UVM

A. IndexProvider

The IndexProviderI class translates between the abstract index and the actual hardware representation. It encapsulates the index and provides a common API to handle a concrete index definition. The following figure demonstrates the IndexProvideI base class along with a concrete implementation. The setIndex() and getIndex() pure virtual functions are invoked by the actual indirect operation. getIndex() is invoked when accessing the data register while the setIndex() is being used by the frontdoor operation. Users providing their own derived IndexProvider may derive their concrete index definition (example: user derived U(unsigned)R(egister)F(field)IndexProvider) .

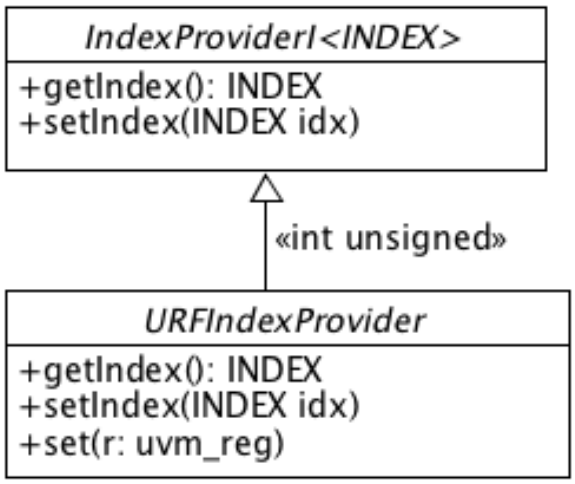

Figure 2 IndexProvider API

The URIndexProvider adds an additional set() method to the IndexProviderI API. This additional method can be used to point the index provider instance to a concrete register field instance. Figure 3 below illustrates the collaboration between the different API functions of URFIndexProvider.

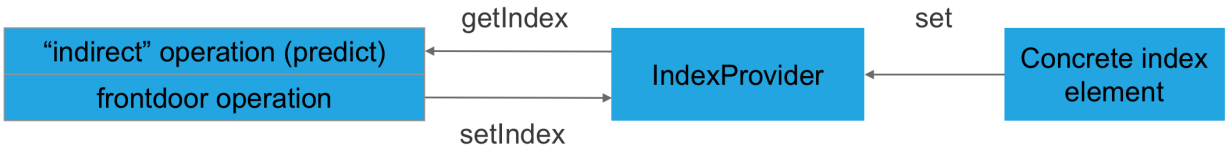

Figure 3 URFIndexProvider API collaboration

The following code example shows the implementation of the URFIndexProvider class. As an example an index can now be a register, a register field, a set of fields or auxiliary wires. In this example a register field is used.

```
// custom implementation:
// using a uvm_reg_field as index provider

class URFindexProvider extends IndexProviderI#(int unsigned);
        // the reference to the uvm_reg_field holding the numeric index
        local uvm_reg_field store;

        // setter for store
        virtual function URFindexProvider set(uvm_reg_field s);
                store=s;
                return this;
        endfunction

        virtual function INDEX getIndex();
                return store.get_mirrored_value();
```

```
        endfunction

        virtual task setIndex(INDEX idx);
                uvm_status_e status;
                store.write(status,idx);
        endtask
endclass
```

Figure 4 Sample custom implementation of IndexProviderI

B. *IndexableStorageProvider*

The IndexableStorageProviderI object mediates between the storage and the generic business logic of the indirect register (see GenericIndirectRegister) and performs the actual selection of elements from the storage. The StorageProvider also implements the inverse operation and can return an index for a particular set of storage elements. The concrete implementation may structure the storage as necessary. A user provided StorageProvider can easily utilize arrays of registers or register fields as storage (example: user derived MyIndexableStorage). Since the selection of elements from the storage is entirely in user-defined and the user can also implement N-dim indexes or 'wildcard' indexes.

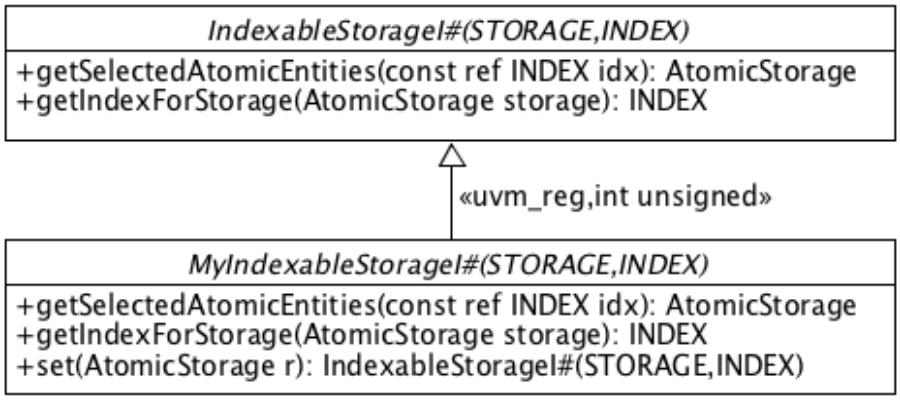

Figure 5 StorageProvider API

C. *GenericIndirectRegister*

This object is the user facing uvm_reg data register that will perform the indirect access when targeted with a RW operation. Since it adheres to uvm_reg it can be used in all places where a uvm_reg is needed. The object needs to be configured with an appropriate IndexProvider and StorageProvider to perform a desired indirect access. The core 'indirect' operation provided by the GenericIndirectRegister is as simple as the following fragment illustrates. The code shown assumes that an atomic storage element, as returned by the StorageProvider, provides a do_predict() method. The class uvm_reg and uvm_field already adhere to that API. As a result the infrastructure integrates nicely into UVM and operations can update the contents via the standard predict() call. That way the same setup can be used in active and passive mode. Users may also override the do_predict() method to provide their own implementation to retrieve the index, select the storage elements and update the storage elements.

D. Usage

To use the package the user needs to make an instance of the GenericIndirectRegister in place of the indirect data register (the register which is accessed for the data transfer). The user also needs to configure the indirect register with a storage and an index provider before the register can be operated. The following example shows an example configuration sequence.

```
// a simple/example test with
// data register @ addr =0
// index register @ addr=4
// 10 unmapped uvm_reg instances accessible via selection in the
// index register field
// 1 register holding a uvm_reg_field (_dummy) with the index
// each of the 10 uvm_reg instances have a frontdoor attached so
// that one can read/write
// to them which the frontdoor translates into access to the index
// field plus the data reg
class test extends uvm_test;
      function new (string name = "test", uvm_component parent = null);
            super.new(name, parent);
      endfunction

      `uvm_component_utils(test)

      GenericIndirectRegister#(uvm_reg,int unsigned) idatareg ;
      URFindexProvider ip;
      MyIndexableStorageI sp;
      block_B regblk;

      reg_R indexreg;
      reg_R registerset[10];

      virtual function void build_phase(uvm_phase phase);
            super.build_phase(phase);

            regblk=new("regblk");
            regblk.default_map = regblk.create_map("regmap",0,
            4,UVM_BIG_ENDIAN);
           regblk.default_map.set_auto_predict(1);

           // the reg field holding the index
           indexreg = new("indexreg");
           indexreg.build();
           indexreg.configure(regblk, null);

           // build and configure our "indirect" register
           idatareg = new("idatareg",32,UVM_NO_COVERAGE);
           idatareg.configure(regblk,null);

           foreach(registerset[idx]) begin
                  // actual registers we are indexing
                  registerset[idx] = new($sformatf("areg[%0d]",idx));
                  registerset[idx].build();
                  registerset[idx].configure(regblk, null,
            $sformatf("rgareg[%0d]",idx));
```

```
                regblk.default_map.add_reg (registerset[idx],
                0+4*idx+8,"RW",1);
          end

// data reg @ addr=0
          regblk.default_map.add_reg(idatareg,0,"RW");
// index register reg @ addr=4
          regblk.default_map.add_reg(indexreg,4,"RW");

          ip=new();     // allocate index provider
          sp=new();     // allocate storage provider

// configure index provider with actual register field
          void'(ip.set(indexreg._dummy));
// configure storage with actual storage array  =registerset[]
          void'(sp.set(registerset));

// configure indirect register properly
          void'(idatareg.setIndexProvider(ip).setStorage(sp));

          // "seal" the register model
          regblk.lock_model();

          // (optional)
          // add frontdoor to registers in order to translate
          // direct access to register into indirect access
          foreach(registerset[idx]) begin
                IregFrontdoor#(uvm_reg,int unsigned) fd;
                fd = new($sformatf("ftdr-%0d",idx));
                fd.configure(idatareg,ip,sp,registerset[idx]);
                registerset[idx].set_frontdoor(fd);
          end

 // setup map for UVM register adapter and sequencer
          begin
                uvm_nodriver_sequencer seqr = new("seqr",null);
                bus2reg_adapter a = new("adapter");
                regblk.default_map.set_sequencer(seqr,a);
          end
    endfunction


    virtual task run_phase(uvm_phase phase);
          uvm_status_e status;
          uvm_reg_data_t data;

          super.run_phase(phase);

          // block until we are done
          phase.raise_objection(this);

          indexreg._dummy.write(status,3);  // write index register
          idatareg.write(status,15); // write reg (registerset[3]=15)

          indexreg._dummy.write(status,2);  // write index register
          idatareg.read(status,data); // read from registerset[2]
          `uvm_info("TEST",$sformatf("got %0x",data),UVM_NONE)

          indexreg._dummy.write(status,3);  // select registerset[3]
          idatareg.read(status,data); // read registerset[3]
```

```
            `uvm_info("TEST",$sformatf("got %0x",data),UVM_NONE)

            registerset[5].write(status,14); // write register non-indexed
            registerset[3].read(status,data); // read register non-indexed


            // now at least we are done
            phase.drop_objection(this);
        endtask
endclass
```

Figure 6 Example configuration sequence (part of example package)

E. *(Optional) IregFrontdoor*

The optional frontdoor object can be used to simplify the access to a storage element. It simply translates the access of a storage element into a proper indirect access including configuration of the index and performing the operation to the data register. It is more or less a preprocessor that translates the high level atomic RW operation into an indirect access. The usage of this class is optional.

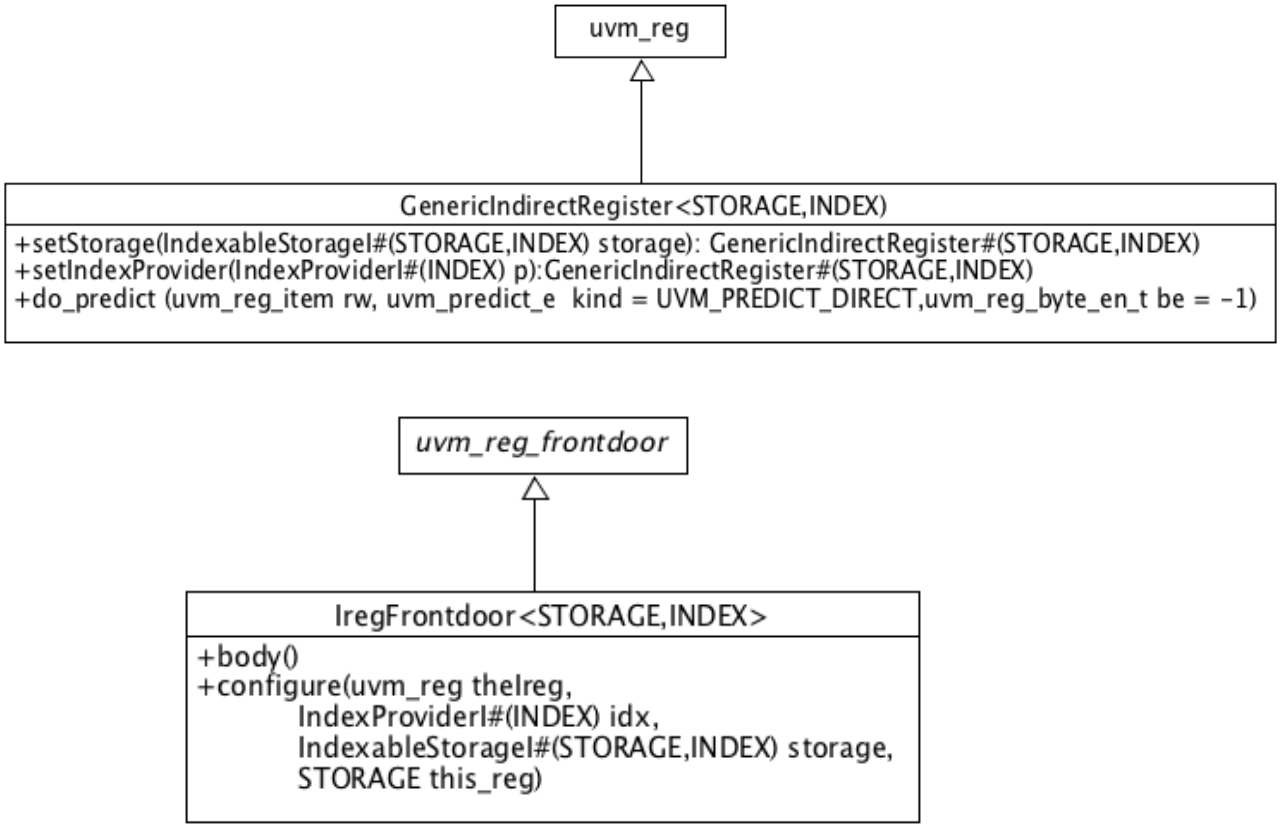

Figure 7 optional frontdoor and GenericIndirectRegister API

F. *Package and Examples*

The code of this package is uploaded to the file area of the Accellera UVM forums (http://forums.accellera.org/files/). The example package contains all the required base classes as well as an

implementation of an IndexProvider (‘URFindexProvider’) providing an ‘int unsigned’ index derived from a ‘uvm_reg_field’ and a StorageProvider (‘MyIndexableStorageI’) providing access to an array of uvm_reg elements. All elements including the provided frontdoor are also shown in a small demo verification environment.

V. SUMMARY

The current release of UVM and the upcoming IEEE version [3] do not provide enough capabilities to model real world indirect register scenarios in a consistent manner. The paper shows that with proper structuring and encapsulation one can build a generic infrastructure for indirect registers. The outlined indirect framework is based on a clear abstraction of index and storage. Through encapsulation and deferral of the unknown or varying functionality into user space users can accomplish a broad spectrum of use models, from common use cases to very special use models, without the need to modify or add to the core library.

VI. REFERENCES:

[1] Accellera, UVM Library, http://accellera.org/downloads/standards/uvm

[2] Freeman&Freemann, Head First Design Pattern, O’Reilly

[3] UVM IEEE 1800.2 Draft D7