2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Register Modeling – Exploring Fields, Registers and Address Maps

A Journey through Object Oriented Programming and Back

Rich Edelman, Siemens EDA, Fremont, CA US (rich.edelman@siemens.com)

***Abstract*—SystemVerilog[1] UVM[2] is a powerful way to supply stimulus and check results. It has many facets and functionalities, many of which become roadblocks to beginner users. This paper explores the thought process and implementation details of modeling fields, registers and address maps. The UVM already contains a capable package which models fields and registers and address maps. This paper will develop a lighter weight and less functional package, but simpler to understand, extend and improve.**

Keywords—SystemVerilog, UVM, Testbench, register models, UVM Registers, address maps.

I. Introduction

The UVM is widely used for verification purposes. It suffers from the perception that it is hard. It is only hard when getting started. Re-using old, simple, well-written UVM testbenches can make UVM adoption go more smoothly. There are many free places to get examples and learn about the UVM.

The UVM also benefits by being a standard that other people know. These other people are resources to hire or consult with. Or even a place to get a job. This UVM ecosystem gets stronger every year with trainers, generators, free code, books and consultants. Stimulus generation with UVM using random constraints and checking using functional coverage is the state-of-the-art for productivity and tape-out success.

The UVM has many pieces. One of the most interesting pieces is a register modeling piece. It models registers, fields, address maps and many related objects.

The UVM Register documentation is well developed with examples in UVM 1.1d and UVM 1.2. This paper will refer to APIs and use models from the UVM, but the code development is started from a clean sheet of paper. This development discusses modeling strategies and various techniques. It is not meant as a replacement for the UVM register package, but as an exploration of modeling that can be applied to registers, fields, address maps and other constructs completely unrelated to registers. This is a conversation about encouraging writing models and using those models to improve verification productivity.

II. Modeling Philosophy

Hardware verification is already hard enough. Productivity tools should be small, transparent, and easy to read and debug.

This paper will create a model of registers, fields, and address maps.

A register models ‘bits’ in the device-under-test. A register can be accessed using ‘addresses’. The bits of a register can be collected and called fields. A register write or read treats the whole register as one and does not access single bits or single fields alone. A register in the device-under-test exists in one “hierarchy”. In the testbench, each device-under-test register is “shadowed” by a single register. The testbench register and the device-under-test register are pairs. They are normally in sync. At certain times they may become out of sync. Out of sync is either an error (mismatch) or is a period of time where the two simply haven’t caught up with each other.

A register has a name and a backdoor name. It has a reset value and most importantly a value. Additionally, it has a table of address maps that it belongs to. It is “in” those address maps. See Figure 1.

An address map is much simpler. It has a name, and a table which maps a register handle and an address together.

Things that can be said about the relationship between registers and address maps:

1. A register can be referenced from many address maps.
2. A register can have multiple addresses.

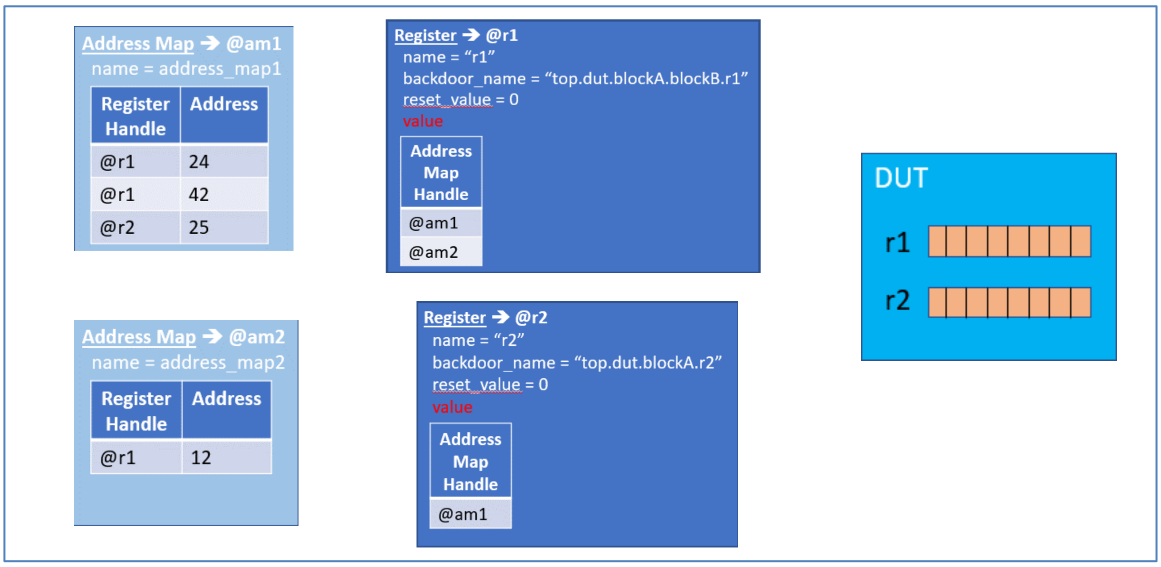

Figure 1- Abstract Model

A register normally is read and written. But there are many special registers which perform special behaviors when they are read or written. For example, a "clear-on-read" register clears its value each time it is read. Or an Id Register which returns the next integer in a predefined sequence each time it is read. Or a broadcast register which causes writes to certain other related registers each time it is written. These special behaviors should be easy to model and easy to debug.

III. Object Oriented Purity

A. *Abstract Classes*

Modeling can take many approaches with respect to the purity of the model. SystemVerilog offers constructs like abstract classes and pure virtual functions. A simple register model definition can be created as

```
pure virtual class register;
  bit [31:0] value;
  pure virtual function T     read();
  pure virtual function       write(T v);
  pure virtual function T     peek();
  pure virtual function       poke(T v);
endclass
```

This is a very nice way to start the modeling. But the value type and the return types for read and peek and the arguments for write and poke cause problems. They are specified as a 'T' – a type to be specified later. Registers are modeling bits and bit vectors, so the T could simply be defined as a "really wide" bit vector – say 4096 bits wide – or just 64 bits – which seems too small. It could even be programmable by recompiling the model. This kind of 'void *' programming doesn't appeal. Instead, this paper will define a simple base class without any type specific items, and an extended class which has the type specific items. See Sections IV and V below

B. *Polymorphic Behavior*

One of the trade-offs with this decision is the loss of some polymorphic behavior[3]. For example, putting all register handles in a list and writing the value 12. Or reading a value. But the argument continues with "Do you really want to blast all the registers with the same value?" A better polymorphic usage would be to call reset on the register list. Registers represent the state on the device-under-test or represent the configuration of the next processing cycle. For example, loading two registers with values and then performing and ALU operation, read the result and check.

```
regA.write(12)
regB.write(12)
```

```
answer = resultReg.read()
```

In this paper, strict polymorphic models will lose favor to more practical models with a "what you see is what you get" approach. In the future when SystemVerilog allows polymorphic return values these base class can be remodeled using this new flexibility. The base class functions can be called polymorphically. The typed functions have less flexibility.

C. The Fields Argument

Fields will not be modeled using a class. Modeling bits with a class is expensive. Field access functions will not be defined. Accessing the field is easy – just access it.

```
register1.value.field2 = 5;
```

The source code above sets the 'field2' struct element of the register1 value to 5.

IV. THE BASE CLASS

The register base class is clean, simple and short.

```
typedef bit [31:0] address_t;

virtual class register_base;
  string name;
  string backdoor_name;

  register_address_map address_maps[address_t];

  virtual function void add(register_address_map am, address_t address);
    address_maps[address] = am;
  endfunction

  virtual function string convert2string();
    return {"No convert2string defined for ", name};
  endfunction

  virtual function void reset();
  endfunction
endclass
```

It defines a few class members, and the address map lookup table that registers will belong to. It also defines any common functions that are not type specific, like reset() and convert2string().

Having a base class will allow for polymorphic programming in appropriate cases.

V. THE REGISTER CLASS

The register class is clean, simple and short. The definition of the register class below defines the access routines for the register. What the API is. Additionally it declares the value and reset_value which every regular register usually has.

```
class register #(type T) extends register_base;
    T reset_value;
    rand T value;

    virtual function void reset();
      poke(reset_value);
    endfunction

    virtual function T read();
      return value;
    endfunction

    virtual function void write(T v);
      value = v;
    endfunction

    virtual function T peek();
      return value;
    endfunction
    virtual function void poke(T v);
      value = v;
    endfunction

    virtual function T backdoor_peek();
      ...
    endfunction

    virtual function void backdoor_poke(T v);
      ...
    endfunction

    virtual function string convert2string();
      return $sformatf("%x", peek());
    endfunction
  endclass
```

The register class can be used as-is for basic registers. Any special behaviors would require extending the class and implementing the special behavior. For example a simple 8 bit register can be defined as

```
register #(bit[7:0]) r1;
```

The register class inherits from the base class and defines a reset value and a value. Those class members are "typed". They are of type 'T' – a parameter that will be specified where the register is declared.

Read and write are defined as functions which return or set the value. If the register has a fancy behavior, then that fancy behavior will be implemented in the read or write routines. Read and write represent using the register in normal operation.

Peek and poke are defined as low-level, basic functions. They do not implement any fancy behavior, but rather simply provide direct access to the value bits. If the peek routine is called, the bits of the register are returned. No special behavior. Peek provides access to the bits. If the poke routine is called, the bits of the register are set. No special behavior. Poke provides a way to set the bits directly.

The backdoor_peek and backdoor_poke are similar in behavior. They provide direct access to the bits that exist in the actual device-under-test and are implemented using the DPI and PLI.

The value class member is a random variable.

The reset function uses the poke function to set the value – for no special reason.

The rest of the implementations are trivial.

Read returns the value. As does peek.

Write writes the value. As does poke.

VI. USING THE BASIC REGISTER

A. *Defining registers*

With this basic register class, many registers can be modeled.

```
typedef register#(bit[7:0]) simple_register;

typedef struct {
  reg [1:0] status;
  reg [7:0] error_count;
  reg [4:0] active_tr;
} csr_t;

simple_register      r1;
register #(bit[7:0]) r2;
register #(csr_t)    b1;
```

The code above defines 3 registers. The first, named 'r1' uses a typedef. The typedef is a nice shorthand – the register can be called a 'simple_register'. The second register, 'r2', just uses the 'register' class directly.

Both of these registers are 8 bits wide.

The third register, ‘b1’, uses a struct to define fields. The fields are named ‘status’, ‘error_count’ and ‘active_tr’.

B. *Constructing Registers*

A register is constructed, and given a name and a backdoor_name, if needed. There is no special handling needed.

```
r1 = new();
r1.name = "r1";
r1.backdoor_name = "top.dut.blockA.blockB.r1";

r2 = new();
r2.name = "r2";
r2.backdoor_name = "top.dut.blockA.r2";

b1 = new();
b1.name = "b1";
b1.backdoor_name = "top.dut.blockA.blockB.b1";
```

The code above would be normally auto-generated inside a block.

C. *Testing the register model*

The model should be tested using the APIs and either automatically checked or logged and checked by hand.

```
typedef register#(bit[7:0]) simple_register;
simple_register             r1;
...
for (int i = 0; i < 256; i++) begin
      // Check write then read
      bv = i;
      r1.write(bv);
      rbv = r1.read();
      if (bv != rbv) // Automatic checking
        $display("Error: Register %s mismatch. Wrote '%8x', Read '%8x'", r1.name, bv, rbv);
      else
        $display(" Info: Register %s match.    Wrote '%8x', Read '%8x'", r1.name, bv, rbv);

      // Check reset
      r1.poke(bv);
      $display("reg=%p, bv=%x", r1, bv); // Logging
      bv = r1.peek();
      $display("reg=%p, bv=%x", r1, bv);

      r1.reset();
      bv = r1.peek();
      $display("reg=%p, bv=%x", r1, bv);
    end
```

The code above is testing register ‘r1’. It is 8 bits wide, and the values from 0 to 255 are written and then read. Then they are poked and peeked. Then the register is reset and peeked again.

There’s some self-checking (comparison code) and some simple logging using %p for fancy formatting and hex printing.

Each register in the register model should be tested without the device-under-test. It is a simple matter since the model itself is so simple.

D. *What about “access modes” or permissions?*

Any access modes can easily be built into the model. For example, a read only field like ‘active_tr’ from the CSR register above could be implemented as

```
class csr_RO_active_tr extends register#(csr_t);
  function void write(T v);
    value.status = v.status;
    value.error_count = v.error_count;
  endfunction
```

```
endclass
```

There are many other special access modes or permissions, but each can be solved with simple coding.

VII. Address Maps

An address map is a way for a register to have one or more addresses. A simple array which contains addresses and handles to class objects (registers) will suffice. It's just a fancy lookup table. Furthermore, the registers can be referenced by name, for example

```
map1.CLRONREAD.read()
```

An address map is a tool which maps register handles with addresses. It can also be used to hold the register handles – for easy access.

A. The Address Map Base Class

From an object-oriented point of view the address map base class is simple. It defines a name and a lookup table and one function to add things to the lookup table. The lookup table is a SystemVerilog associative array. The array is indexed by address – given an address, it is simple to find the register handle.

```
class register_address_map;
  string name;
  register_base registers[address_t];

  virtual function void add(register_base r, address_t address);
    registers[address] = r;
    r.add(this, address);
  endfunction
endclass
```

B. Defining An Address Map

Defining an address map is simply extending the address map base class and adding any desired 'direct handles.

Here are two address maps used in the example code

```
class my_address_map1 extends register_address_map;
  simple_register r1;
  register#(bit[7:0]) r2;
endclass

class my_address_map2 extends register_address_map;
  simple_register r;
endclass
```

C. Adding register, address pairs to the address map

The maps are constructed and named.

```
address_map1 = new();
address_map1.name = "address_map1";
address_map2 = new();
address_map2.name = "address_map2";
```

The registers are added with their addresses

```
address_map1.add(r1, 42);
address_map1.add(r1, 24);
address_map1.add(r2, 25);
```

The special direct handles are set

```
address_map1.r1 = r1;
address_map1.r2 = r2;
```

D. *Using the registers via the address map*

Using the register from the address map is simple

```
value = address_map1.r1.read()
address_map1.r2.write(12)
```

There's nothing so special about address maps. An address map is a simple lookup table. An arbitrary class could contain register handles – say a 'file' or 'block' of registers. That arbitrary class could have a lookup table. It is an address map.

The built-in address map looks up registers by address, but there could also be a lookup by name. Or a lookup by type or any other kind of convenient grouping. It only requires adding a new lookup table and filling as the model is built.

E. *Using the address map*

In a monitor on a bus, an address is identified in the bus transaction. That bus transaction might be recognized to be a READ transaction from address 104. The monitor can lookup address 104 and know that REGISTER13 is being read.

VIII. Fields

There are two ways (at least) to model fields. First, fields are the named 'struct' elements in a register class. Second, the fields are really first-class citizens. They are really just "smaller" registers. In this second case, the fields are simply modeled using the 'class register' and the value is a simple bit vector. This way of modeling fields is quite useful when the exact layout of registers is not yet known – fields will be combined later in the design cycle into groups and given 'register names'.

IX. Back Door Access

Backdoor access is an important part of register modeling. The model under development models an RTL register as a class. One register, one class. In this case, the class can know the full path name to the RTL collection of bits that are the actual, physical representation of the register. This back door path can be used to directly sample or set the register values. Using back door access can speed up verification, since front door bus cycles are not needed for register reading or writing.

X. Conclusion

As a reader of this paper an understanding should have developed for subtle considerations for base class design and trade-offs between simplicity and covering all possible functional specifications with the library package. Certain trade-offs exist and must be considered. The paper advocates for simplicity in all ways. This results in easier to use and easier to debug code. It may also restrict or constrain usage of the model. The certain simplifications made and tradeoffs do not appear to invalidate the model. It is a usable beginning.

Please look for a discussion about register tests, register sequences for using these registers and address maps in a forthcoming paper.

The complete library source code with register models and address maps is below in the Appendix. Please contact the author for a small test program – or write one yourself.

This register model is not meant as a replacement for the UVM register package, but comparisons are likely. This register model is small and transparent. Easy to debug. It lacks the automatic connection to sequences and other conveniences. But the UVM register model has about 5000 lines of code in 3 files that describe registers and fields. This register model has 1 file with less than 100 lines. Comparing lines and speed isn't fair, since the register model here is still new, without error checking and comments. But the new model occupies 1/10 the RAM and builds in a few seconds. The UVM register model builds in a few minutes. As this register model package grows and is refined, a major goal will be to keep it small and fast, while exploring modeling and functionality trade-offs.

Both the UVM register package and this register model naturally require some automated code generation – building the SystemVerilog style model from an IPXACT style model, for example. That automation is key to the success of any modeling package for registers.

Happy Modeling.

XI. References

[1] SystemVerilog LRM, “1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language”, https://ieeexplore.ieee.org/document/8299595

[2] UVM 1.1d - https://www.accellera.org/downloads/standards/uvm

[3] “What Does The Sequence Say? Powering Productivity with Polymorphism”, Rich Edelman, DVCON US 2022

XII. Appendix

A. *The complete register package source code*

```
package registers;

  typedef bit [31:0] address_t;
  typedef class register_address_map;

  // ------------------------------------------------
  virtual class register_base;
    string name;
    string backdoor_name;
    register_address_map address_maps[address_t];

    virtual function void add(register_address_map am, address_t address);
      address_maps[address] = am;
    endfunction

    virtual function string convert2string();
      return {"No convert2string defined for ", name};
    endfunction

    virtual function void reset();
    endfunction

  endclass

  // ------------------------------------------------
  class register #(type T) extends register_base;
    T reset_value;
    rand T value;

    virtual function void reset();
      poke(reset_value);
    endfunction

    virtual function T read();
      return value;
    endfunction

    virtual function void write(T v);
      value = v;
    endfunction

    virtual function T peek();
      return value;
    endfunction

    virtual function void poke(T v);
      value = v;
    endfunction

    virtual function T backdoor_peek();
      //...uvm_dpi implementation...
    endfunction

    virtual function void backdoor_poke(T v);
      //...uvm_dpi implementation...
    endfunction

    virtual function string convert2string();
      return $sformatf("%x", peek());
    endfunction

  endclass
  // -------------------------------------------------
  class register_address_map;
    string name;
    register_base registers[address_t];

    virtual function void add(register_base r, address_t address);
      registers[address] = r;
      r.add(this, address);
    endfunction
  endclass
endpackage
```

B. Defining some fancy registers

```
package my_registers;

  import registers::*;

  typedef struct {
    reg [1:0] status;
    reg [7:0] error_count;
    reg [4:0] active_tr;
  } csr_t;

  // -------------------------------------------------
  // Regular register, but with fields - a struct
  // There are three fields, one of which is read-only
  class csr_RO_active_tr extends register#(csr_t);
    function void write(T v);
      value.status = v.status;
      value.error_count = v.error_count;
    endfunction
  endclass

  // -------------------------------------------------
  // Id Register
  // Each time read() is called, return the next item in the list of Id's
  // This register is read only
  class id_register #(type T) extends register #(T);
    T values[$];
    int index; // -1 means empty. Otherwise points
               //    at the next to read.
               // 0 is the first
               // .size() is that last+1

    function new();
      super.new();
      index = -1;
    endfunction

    function void add_id(T v);
      values.push_back(v);
      index=0;
    endfunction

    function T read();
      T value;
      value = values[index];
      index++;
      if (index >= values.size())
        index = 0;
      return value;
    endfunction

    function void write(T v);
    endfunction
    function T peek();
    endfunction

    function void poke(T v);
    endfunction

    function T backdoor_peek();
      return super.backdoor_peek();
    endfunction

    function void backdoor_poke(T v);
    endfunction

    function string convert2string();
      return $sformatf("values=%p, index=%0d",
        values, index);
    endfunction
  endclass

  // -------------------------------------------------
  // Broadcast Register
  // When a broadcast register is written to, it in turn writes to any associated registers.
  // In this way, writing to the broadcast register causes that value to be broadcast (written)
  // to the associates (targets in the code)
  // This register is never read
  class broadcast_register #(type T) extends register#(T);
    register#(T) targets[$];

    virtual function void add_target(register#(T) r);
      targets.push_back(r);
    endfunction

    function void write(T v);
      super.write(v);
      foreach (targets[i])
        targets[i].write(v);
    endfunction

    function void poke(T v);
      super.poke(v);
      foreach (targets[i])
        targets[i].poke(v);
    endfunction

    function string convert2string();
      return $sformatf("targets=%p", targets);
    endfunction
  endclass

  // -------------------------------------------------
  // Clear On Read
  // When the read function is called, the value in the register is set to zero,
  // and the previous value is returned.
  // A value is read one time, and then the register is cleared.
  class clear_on_read_register #(type T)
        extends register#(T);

    function T read();
      T v;
      v = peek();
      poke('0);
      return v;
    endfunction

  endclass
endpackage
```