Automated approach to Register Design and Verification of complex SOC

Ballori Banerjee
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979520
bbanerje@lsi.com

Subashini Rajan
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979614
srajan@lsi.com

Silpa Naidu
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979603
snc@lsi.com

ABSTRACT
Today's designs contain several hundreds to thousands of registers and memory elements. Starting from documentation to design implementation to verification of each single register, each bit and its property involves a lot of time and complexity.
Use of a single source, written in a high-level register and memory modeling language like SystemRDL, for documentation, design and verification helps to reduce this complexity.

The paper describes this methodology which provides an almost zero-time, low maintenance, and reusable register design and verification system. A complete solution from SystemRDL to RTL and documentation, to a complete reusable VMM based register verification environment, the Register Abstraction Layer-RAL, is discussed

The paper presents useful RDL constructs for modeling scalable register descriptions, like registers arrays, *regfiles* and register field instantiation. Also presented are constructs for modeling standard register types like interrupt enable and interrupt status register. A few useful field properties and their mapping to hardware implementation are discussed.

Commercially available register automation tools can be used to generate several outputs from SystemRDL input. This includes document, RTL, C headers, verification components and other custom outputs which may be required. Challenges encountered while setting up the flow with third party tools are discussed.

An example comprising a set of read-write and status registers is provided to help in understanding the transition of the input to the outputs formats.

Once setup, the flow is repeatable and can be used across block, cluster (Sub chip-level) and chip level, with lot of reuse of code and environment

Categories and Subject Descriptors
[**Hardware Software Co-design**]: Automatic Register modeling flow, language constructs

General Terms
Language, Design, Verification.

Keywords
CSR: Control and Status Register
RDL: Register Description Language
RTL: Register Transfer Language
HDL: High Level Description Language
SoC: System on Chip
RAL: Register Abstraction Layer, a VMM application package
RALGen: Synopsys tool to convert RDL to RAL
VMM: Verification Methodology Manual, by Synopsys

1. INTRODUCTION
Registers and memory elements constitute a large percentage of today's large and complex designs. On-chip registers define the software interface to the chip, and usually represent the largest portion of the chip specification or programmer's guide. Continuously increasing number of registers makes documentation, implementation and maintenance a growing challenge. Moreover, changing specifications during the design cycle require repeated updates to design, test bench, and register/memory test cases as also to documentation. Manually managing these components affects productivity and increases probability of introducing errors in the process.

Most times, though, registers have a regular structure, defined by their field attributes. Using this characteristic, it is possible to define a flow where the register architecture is defined in a high level register description language like SystemRDL, which in turn is used to generate the design, documentation and verification components. This helps to reduce the often tedious and error-prone task of managing registers, and enables design, verification and firmware teams to work more efficiently from consistent and synchronized views of the chip design.

We have implemented this flow on a multi-million gate SoC (around 140 million gates) where on-chip registers are greater than 25,000 overall! Having initially started with the manual coding of registers and later moving to the unified register management flow, the following section illustrates the advantage in terms of effort saving achieved by adopting the methodology.

2. LEGACY FLOW: NO AUTOMATION
Register definition generally starts with an architect scoping out a specification. Once the specification is completed the hardware engineer, software engineer, and verification engineers can begin coding different views of the registers described in the functional specification. Once we have a design, verification and software engineers can start running tests. Anytime a bug is discovered the specification must be changed and all the subsequent outputs must be changed accordingly. But due to other priorities the designer would have changed the code but not the document or vice versa. This

process repeats itself many times over the course of the project. Bugs are only one source of change though. Marketing requests may also come in at any stage of the design cycle requiring the specification to change and all downstream code to be modified. Figure 1 captures this course in a flow chart.

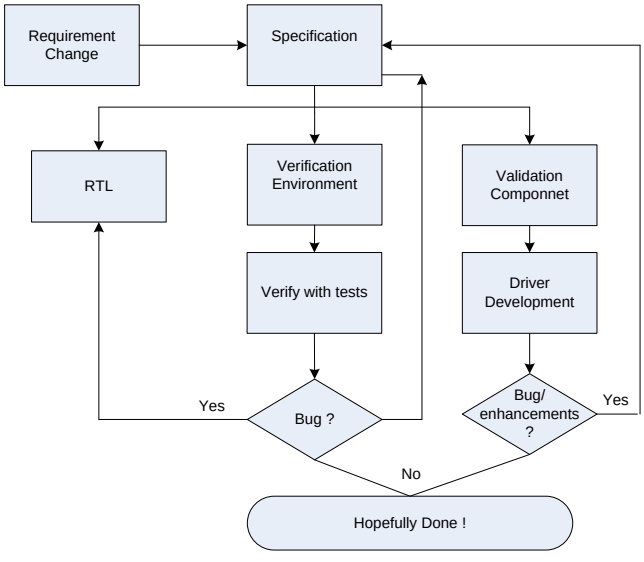

Figure 1: Legacy Flow

As an example, in a module that we are implementing, there are four thousand registers. Translating into number of fields, for 4000 32-bit registers we have 128,000 fields, with different hardware and software properties!

Coding the RTL with address decoding for 4000 registers, with fields having different properties is a week's effort by a designer. Developing a re-usable randomized verification environment with tests like reset value check, read-write is another 2 weeks, at the least. Closure on bugs requires several feedbacks from verification to update design or document. So overall, there is at least a month's effort plus maintenance overhead anytime the address mapping is modified or a register updated/added.

This flow is susceptible to errors where there could be disconnect between document, design, verification and software. The automated register design and verification (DV) flow streamlines this process.

Adopting the automated flow, it took 2 days to write the RDL. The rest of components were generated from this source. A small amount of manual effort may be required for items like back-door path definition, but it is minimal and a one-time effort.

3. AUTOMATED REGISTER DESIGN AND VERIFICATION FLOW

3.1 Methodology

The flow starts with the designer modeling the registers using a high level register description language, like SystemRDL. Third party tools are available to generate the various downstream components from the RDL file:

i. RTL in Verilog/VHDL
ii. C/C++ code for firmware
iii. Documentation (different formats)
iv. high level verification environment code (HVL)

This is shown in Figure 2. The RDL file serves as a one-stop point for any register update required following a requirement change.

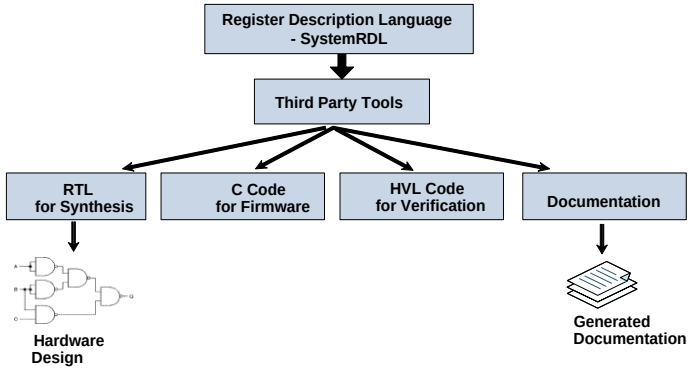

Figure 2: Automated Register DV Flow

3.1.1 Choosing SystemRDL

While evaluating the register modeling options for the flow, following items were considered:

1. Ease of capture: To get various designers to agree to use a language standard other than a HDL for register modeling, it should be possible to capture the specifications in a user readable and writable format.
2. Comprehensive set of constructs: It should be possible to define all types of registers that may be used in a design, for instance read-write, read-only, interrupt enable/mask, multiple instances of a register, a group of similar registers, external registers.
3. Ease of usage: Defining various registers using the constructs should be fairly straight-forward.
4. Ease of maintenance/version control: This allows change control.
5. Support by vendors and stability of flow: We needed to check with third party vendors on how mature their tools are to support a particular standard as an input.
6. Implementation guidelines: If a standard has inbuilt implementation guidelines it is easy to understand the output generated and allows portability.

In this regard, SPIRIT IP_XACT XML and SystemRDL standards were considered. Also, possibility of an Excel spreadsheet for capturing register definitions was explored.

IP_XACT is an XML format for capturing design components. However, SystemRDL provides a user readable and writable format to succinctly capture the description from which rest of the deliverables are produced. It has several constructs with particular implementation guideline for different types of registers, like read-write, read-only. Being a text file, it lends itself to easy editing and maintenance using version control.

An Excel spreadsheet appears easy; however a standard format needs to be used for all IP blocks of a SOC, while having some method for version control. There is no defined specification for RTL implementation when registers are defined in a spreadsheet. Hence generated RTL is open to tool interpretation of spreadsheet register attributes. It lacks a defined method for grouping similar registers or creating an array of registers, where the basic register is defined only once and we can specify the number of instances of it at defined

addresses. SystemRDL constructs are very efficient to capture such requirements.

Thus we arrived at SystemRDL as a standard way of defining registers for all blocks in our project.

3.2 Extending flow for VMM based Verification

Register Abstraction Layer, RAL is a VMM application package which helps create an object oriented abstraction layer to model registers and memories in a design under test.

A complete VMM compliant randomized, coverage driven register verification environment can be created by extending the flow such that:

i. Using 3rd party tool, from SystemRDL the verification component generated is RALF, Synopsys' Register Abstraction Layer File.
ii. RALF is passed through RALGEN, a Synopsys utility which converts the RALF information to a complete VMM based register verification environment. This includes automatic generation of pre-defined tests like reset value check of registers and functional coverage model, which would have taken considerable staff-days of effort to write.

Figure 3 illustrates the Verification flow.

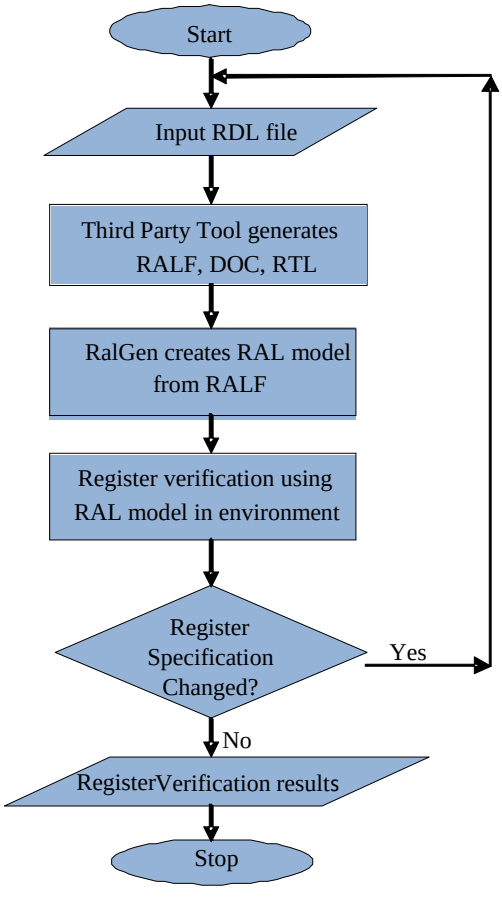

Figure 3: Generating VMM based RAL model for Verification

4. DEMONSTRATING WITH A CSR EXAMPLE

4.1 Source File: SystemRDL Description

SystemRDL is an object-oriented register description language (RDL).Its semantics support the entire life-cycle of registers from specification, model generation and design verification to maintenance and documentation. Components are defined in SystemRDL using four basic types of defining elements:

i. Fields – keyword *field*. This is the basic component that usually maps to a flip-flop or wire/bus. A register's individual bit/bits are mapped to field.
ii. Registers – keyword *reg*. A Register (*reg*) has a set of one or more *field* instances that are atomically accessible by software at a given address
iii. Register files – keyword *regfile*. Describes a logical grouping of one or more register and register file instances.
iv. Address Maps – keyword *addrmap*. *Addressmap* contains registers, register files or other *addressmaps* and assigns an address, defining the boundary of an implementation.

Each component relates to a number of properties which describes its purpose and implementation.

A *field* has four basic properties:

i. *fieldwidth* : describes number of bits/bit
ii. *reset:* has the default/on-reset value.
iii. h*w:* captures design's ability to sample/update a field
iv. sw*:* captures programmer's ability to read/write a field.

In addition to these four, software/hardware properties like *rclr, woclr, hwclr* can be added to model the corresponding behavior.

Also, there are properties to add descriptive content that gets reflected in documentation: *name; desc*

An example of a Control and Status register, CSR, modeled in SystemRDL is given in Table 1.

Table 1: SystemRDL for CSR Example (CSR_EXAMPLE)

```
// **** FIELDS ****
field myControl {
 hw = r;
 sw = rw;
 fieldwidth = 16;
// The above combination would result in a flip-flop in CSR register.
// The following would show up in generated document outputs
//(HTML, etc.)
  desc = "CSR example's 16 bit control field";
  reset = 16'h3020;
};

field myStatus {
  hw = w;
  sw = r;
  desc = "CSR example's status field";
};
// **** REGS ****
reg Control_and_Status_reg {
   myControl control[15:0];   // bit position assigned
  myStatus   status[31:28];   // moves bit position to [31:28]
 // Thus bit[27:16] are now reserved
 status->reset = 4'b1010;   // reset value defined for status
 };
// **** ADDRMAPS ****
addrmap csr_example {
  name = "RDL Example for Control Status Register";
  desc = "An example Addressmap.";
Control_and_Status_reg CSR   @0x0020;
 reg {
    field { reset = 32'hABCD_BEEF;} myField[31:0];
  } myReg @0x0024;
};
```

4.2 Output Component 1: RALF

The Synopsys Register verification file component, RALF, for the RDL CSR description (Table 1) is shown in Table 2.

Table 2: RALF for CSR_EXAMPLE

```
block csr_example {
  bytes 4;
  register CSR @0x20 {
    bytes 4;
    field CSR_control (CSR_control) @0 {
      bits 16;
      access rw;
      reset 0x3020;
    }
    field CSR_status (CSR_status) @28 {
      bits 4;
      access ro;
      reset 0xa;
    }
  }
  register myReg @0x24 {
    bytes 4;
    field myReg_myField (myReg_myField) @0 {
      bits 32;
      access rw;
      reset 0xabcdbeef;
    }
  }}
```

4.3 Output Component 2: RTL

Using a third party tool to generate RTL will ensure standard RTL interface for the registers, including read /write strobes and address bus as inputs and the register fields as outputs. The RTL interface corresponding to the example RDL is shown in Table 3.

Table 3: Verilog RTL Interface for CSR_EXAMPLE

```
module csr_example (
 input wire         CLK,
 input wire         cpu_if_read,
 input wire         cpu_if_val,
 input wire [31:0]    cpu_if_write_data,
 input wire [5:0]     cpu_if_address,
input wire [3:0]    CSR_status_next,
input wire          RESET
input wire [31:0]    myReg_myField_next,

  output wire [31:0]  cpu_if_read_data,
  output wire         cpu_if_access_complete,
  output wire         cpu_if_invalid_address,
  output wire         cpu_if_invalid_access,
  output reg [15:0]   CSR_control,
  output reg [31:0] myReg_myField
  );
……
……
endmodule
```

4.4 Output Component 3: Document

Register documentation of different IPs in a SOC may not have the same look and feel. A generated document will ensure a uniform look and keep the document in sync with the other components. Document view for the CSR is as in Table 4. Most third-party tools will allow some customization in the look and information content of the document .Document formats supported can vary from Microsoft Word, RTF to HTML.

Table 4: Document View for CSR_EXAMPLE

RDL Example for Control Status Register

Type: addressmap
Identifier: csr_example
Access: R/W
Type Name: csr_example
Description: An example Addressmap.

Table 1 Address Table for RDL Example for Control Status Register

Address	Title	Page
0x0 - 0x1F	-	
0x20	CSR	0
0x24	myReg	0

CSR

Type: register
Identifier: CSR
Offset: 0x20
Access: R/W
Reset: 0xa0003020
Type Name: Control_and_Status_reg

Table 2 Fields in CSR

Title	Bit	Access	Reset	Description
control	[15:0]	R/W	0x3020	CSR example's 16 bit control field
-	[27:16]	R	0x0	-
status	[31:28]	R	0xa	CSR example's status field

5. VERIFICATION VIEW

The generated RALF is passed through RALGEN, as shown in Figure 3, to generate an object oriented, reusable, coverage driven VMM based register verification environment. A small part of the System Verilog code generated for CSR_EXAMPLE (above example) is shown in Table 5.

Table 5: VMM based System Verilog environment

```
`include "vmm_ral.sv"
class ral_reg_csr_example_CSR_bkdr extends
vmm_ral_reg_backdoor;
        function new(vmm_ral_reg __ral_reg);
                super.new(__ral_reg);
        endfunction
        virtual task read ();
        super.pre_read(data);
        ……
        endtask
        virtual task write();
        super.pre_write();
        ……
        endtask
endclass

class ral_reg_csr_example_CSR extends vmm_ral_reg;
        rand vmm_ral_field CSR_control;
        vmm_ral_field CSR_status;

        function new();
        super.new();
        endfunction: new
endclass : ral_reg_csr_example_CSR

class ral_block_csr_example extends vmm_ral_block;
……..
endclass : ral_block_csr_example
```

For each field, register, block and system component available in RALF, the RAL contains a System Verilog class. These classes are extended from RAL base classes. The attributes of the components in RALF such as base address, offset, reset value, domain name are passed to the individual classes as arguments.
This RAL model can be integrated in a VMM environment for complete DUT register verification, as in Figure 4. The XL XACTOR translates the RAL commands to interface commands. The BFM uses these to drive DUT signals as per protocol.

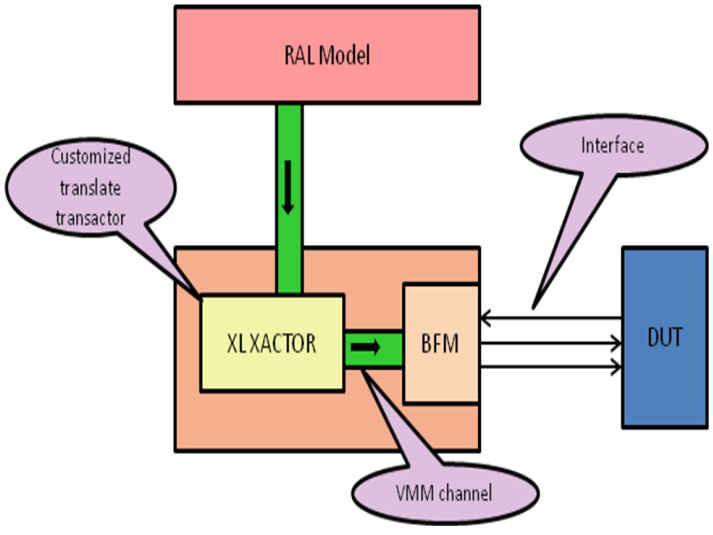

Figure 4: RAL integration in VMM environment

RAL has several useful features that help in building verification environment for large and complex designs:

- Named register access
- Mirror register
- Functional coverage model
- Predefined tests
- Access tasks

These are well documented in the RAL user guide that we need to refer for further usage details.

6. VERIFICATION ASPECTS AND CHALLENGES

A few important verification aspects and challenges encountered while using the verification flow from SystemRDL to RALF to RAL SV classes are discussed here.

6.1 Backdoor Access

There are two ways of accessing design registers in a verification environment: front door and back door. Front door is by using the design register bus. This consumes cycles and follows the register bus protocol. Backdoor is a zero simulation time access by mapping to the design register directly using the HDL path and allows quick configuration of registers. Thus, configuring by backdoor saves simulation time, especially useful for full-chip and sub-chip simulations where several registers need to be setup.

Backdoor access also helps in uncovering address decode design bugs. Since the mirror register gets updated while doing front-door or back-door writes, writing a register in front door and reading it from the backdoor flags any mismatch between design and the mirror register.

Another important use of backdoor access is to know when a register field (RO bit) is updated by design and use that information without polling for this (RO) bit using the register bus. The example is an Interrupt status register bit (RO), Using the back door, we get to know that Interrupt has set and ISR procedure is executed as in the system.
To have the backdoor path in the verification environment, it needs to be present in the RALF. To avoid manually inserting the backdoor path for each register in RALF, it needs to be present in the generated RALF code.

However, generation of backdoor path for a register in RALF is subjected to the implementation method of tool converting RDL to RALF. SystemRDL does not provide any guideline for associating a backdoor path with a register or register field. If not directly provided in the RALF /HVL code, users will need to manually add the backdoor path to RALF to enable backdoor accesses
Further, there is a difference in backdoor path specification format for Verilog and VHDL RTL. Tool support is required for both.
Below example shows the backdoor path definition for Verilog and VHDL RTL, in RALF file. ‘didc’ is the backdoor path in the design.

Backdoor path for Verilog design:

```
register DID (didc) @0x0{
};
```

Backdoor path for VHDL design:

```
register DID vhdl_path = (didc) @0x0{
};
```

The complexity of specifying/generating backdoor path increases when there is legacy/IP RTL that is not generated from the SystemRDL file. If the IP RTL register field name does not match exactly with the RDL register field name, the RTL will need to be traced and manually mapped in the verification environment to the corresponding register’s backdoor path.

Also, field names in generated RTL are usually of the format <register_name>_<filed_name> that is, prefixed with the register name. However the field name in hand-written RTL does not usually follow this format. Thus, if a tool automatically derives backdoor paths from the RDL file, it possibly needs to be different for generated RTL as compared to hand-written RTL.

If a designer writing the System RDL file for an IP RTL does not refer to the RTL while writing the RDL, it is possible to have a scenario where the fields are defined in RDL which do not exist in the RTL. If a tool is automatically deriving backdoor paths from RDL, the field backdoor paths will be present in the RALF component, will come in to the System Verilog environment and give elaboration errors due to absence of the actual path in the RTL.

The backdoor paths for Register arrays or array of regfiles are difficult to map automatically. If a bunch of registers are defined as a register array in RDL, most times the RTL, generated or hand-written, will be scalar. Thus, though the verification environment will have an array, the corresponding RTL implementation will have individual registers. In such a case, the backdoor path to registers will need to be manually added to the verification file.

Adding another dimension to this is the current limitation of RALGEN in supporting hexadecimal array indices. Let us consider a design having 64 registers for CHANNEL_CONFIG, for 64 channels of a DMA. RTL may have these implemented as scalar with name formatting such that the individual register instances have hexadecimal numbering (example : dma_config_a ,dma_config_b where trailing letters stand for hexadecimal numbers). However,

while generating the VMM classes from RALF, RALGEN gives error if a hexadecimal index is present in the backdoor definition of a register array So, for above example following code will give error:
register DMA_CHANNEL[4] (dma_channel_[%x]) @0x4 {};
This needs to be replaced with:
register DMA_CHANNEL[4] (dma_channel_[%d]) @0x4 {};
This will require a level of post processing to convert the decimal indices in backdoor definition of the SV register classes to hexadecimal to map to RTL.

Tools need to be able to handle all these different permutations to generate the backdoor paths correctly. These are backdoor access issues found and tool enhancements requested while working on our project.

If all tools reading SystemRDL followed some standard guidelines while implementing register interface, for hierarchical addressmaps and register arrays implementation, keeping in mind backdoor path requirement, the flow would be considerably smoothened for verification.

While implementing RTL for a SystemRDL defined register array, the possibility of an RTL array (using VHDL/ Verilog generate) could be explored. This would make backdoor path mapping seamless, though of course tools will need to work with HDL limitation on two dimensional arrays at ports.

At present, we have a configuration file being passed to the tool to provide the hierarchical backdoor path of the registers, where it needs to be manually defined. However, to minimize the effort of writing the configuration file, it helps to have a switch to enable generating the backdoor paths such that it is either same as that of the field instance name or is the register name appended with field name or is not generated (where field is not present in say a third-party IP RTL).

A few SystemRDL guidelines if followed can reduce verification effort in defining backdoor path for IP/non-generated RTL.

- Register/field names should match the RTL register/field names to enable generation of backdoor paths automatically.
- If there are no fields in the RTL, it is recommended to have a single field in the RDL with the same name as RTL register name. However, if multiple fields need to be defined in the RDL for use in the verification environment, then field names may be different and that information needs to be captured in the input file used by the third party tool to generate the backdoor paths in RALF. In the input file for the tool, fields with register slices as backdoor paths should be manually added.

6.2 Multiple Views/Interfaces

It is possible to have registers in today's complex SOC designs which need to be connected to two or more different buses and accessed differently. The register address will be different for the different physical interfaces it is shared between. This can be defined in SystemRDL by using a parent *addressmap* with *bridge* property, which contains sub addressmaps representing the different views.
For example:

```
addrmap dma_blk_bridge {
   bridge;// top level address map
    reg commoncontrol_reg {
      shared; //  register will be shared by multiple address maps
          field {
              hw=rw;
              sw=rw;
             reset=32'h0;
             } f1[32];
          };
          addrmap {// Define the Map for the AHB Side of the bridge
            commoncontrol_reg cmn_ctl_ahb @0x0; // at address=0
          } ahb;

          addrmap { // Define the Map for the AXI Side of the bridge
          commoncontrol_reg cmn_ctl_axi @0x40; // at address=0x40
          } axi;
 };
```

The equivalent of multiple view *addressmap*, in RALF is ***domain***. This allows one definition of the shared register while allowing access from each domain to it, where register address associated with each domain may be different .The following code is RALF with domain implementation for above RDL.

```
   register commoncontrol_reg {
       shared;
        field f1 {
          bits 32;
          access rw;
          reset 'h0;
       }
   }
   block dma_blk_bridge {
     domain ahb {
       bytes 4;
      register commoncontrol_reg =cmn_ctl_ahb @'h00 ;
     }
     domain axi {
       bytes 4;
      register commoncontrol_reg=cmn_ctl_axi  @'h40 ;
     }
   }
```

Each physical interface is a domain in RALF. Only blocks and systems have domains, registers are in the block. For access to a register from one interface/domain RAL provides read/write methods which can be called with the domain name as argument.

```
    ral_model.STATUS.write(status, data, "pci");
    ral_model.STATUS.read(status, data, "ahb");
```

This considerably simplifies the verification environment code for the shared register accesses.

However, when tools do not support domain, the RALF is created having effectively two or more top level systems re-defining the registers. This can blow up the RALF file size and also verification environment code.

```
system dma_blk_bridge {
  bytes 4;
  block ahb (ahb) @0x0 {
      bytes 4;
     register cmn_ctl_ahb @0x0 {
          bytes 4;
         field cmn_ctl_ahb_fl(cmn_ctl_ahb_f1)@0{
                bits 32;
                access rw;
                 reset 0x0;
           } }
   }
```

```
block axi (axi) @0x0 {
    bytes 4;
    register cmn_ctl_axi @0x40 {
        bytes 4;
        field cmn_ctl_axi_f1 (cmn_ctl_axi_f1) @0 {
                bits 32;
                access rw;
                reset 0x0;
              } }
        }
}
```

In the above example, the tool is generating two blocks 'ahb' and 'axi' and re-defining the register in each block. For multiple shared registers, the resulting verification code will be much bigger than if *domain* had been used.

Also, without the domain associated read/write methods (as shown above) for accessing the shared registers, it will be at least a few lines of code per register for accessing it from a domain/interface. This makes writing the test scenarios complicated and wordy.

Using domain makes shared register implementation and access in verification environment easy. However, since tool support was not available this feature of RAL could not be exploited. Tools should use the shared and domain properties while generating RALF to support shared register access from multiple interfaces.

7. USEFUL SYSTEMRDL CONSTRUCTS

7.1 Interrupt

RDL provides particular constructs to define registers like interrupt-enable/mask and interrupt-status from which interrupt will be derived. Each bit in the interrupt status register has to be mapped with corresponding enable/mask bit in the interrupt enable/mask register using interrupt field access property enable or mask. If it is enable corresponding interrupt source is used to generate an interrupt. In case of mask, corresponding interrupt source is not used to generate an interrupt. Each fieldwidth defined in interrupt status and interrupt enable register should be 1.SystemRDL example for interrupt status and enable register and their mapping is given in Table 6.

Table 6: Interrupt Register Example

```
reg  irq_status_reg {
name  = " IRQ_STATUS";
desc   = "Interrupt status register";
 field  fifo_status  {
       hw  = w;
      sw  =  rw;
      woclr;
     intr;
     reset = 1'h0;
 };
   fifo_overflow   fifo_status  [0:0] ;  // bit number zero
   fifo_underflow   fifo_status [1:1] ;  // bit number one
   fifo_overflow ->desc = " Set when fifo overflows";
   fifo_underflow ->desc = " Set when fifo underflows";
 };
reg  irq_enable_reg {
name  = " IRQ_ENABLE";
desc   = "Interrupt enable register";
 field  fifo_status_enable  {
           hw  = na;
           sw  =  rw;
           reset = 1'h0;
    };
 fifo_status_enable  fifo_overflow_enable [0:0] ;
fifo_status_enable  fifo_underflow_enable  [1:1] ;

fifo_overflow_enable->desc = "when set high enables interrupt
 generation if corresponding source is set";
fifo_underflow _enable->desc = "when set high enables interrupt
 generation if corresponding source is set
 };
 addrmap{
  irq_status_reg  irq_status@0x0000;  //  offset zero
 irq_enable_reg  irq_enable @0x0004;  //  offset  four
 };
 irq_status.fifo_overflow->enable= irq_enable.fifo_overflow_enable;
 irq_status.fifo_underflow>enable=irq_enable.fifo_underflow_enable;
```

The implemented hardware will have an interrupt signal irq_status which is the logical OR of all interrupt fields in the status register ANDed with interrupt enable according to the mapping.

7.2 Register Array

For defining multiple instances of the same register *regfile* and *register array* are useful RDL constructs. A regfile can contain one or more registers whose array can capture multiple instances of the same register. Using regfile arrays and register arrays helps to configure the registers in a loop in the verification environment,. This helps reduce lines of code required to configure, as in the example below.

Example: DMA design has two channels; a channel can be implemented as a regfile with a set of registers and the numbers of instances are based on the number of channels. Using a for-loop, the registers of all the channels can be accessed. Even variations in the number of channels would not require any modification in the verification environment. In this example, depending on 'dma_descr_pkt.chnum' [2 in this case], all the channels registers can be accessed, using a single line statement. If there are individual registers, the number of lines would increase, since we have to write a line of code for each register.

```
ral_model.dma_blk.CHANNEL[dma_descr_pkt.chnum].CHANNELC
ONTROLSTATUS.read (status, reg_value);
```

7.3 Addressing Mode

The RDL default addressing mode is that address is aligned to the width of the component being instantiated. This requires the regfile array offset to be aligned with the regfile size.

For example, in below example there are two regfiles. DMA_WPL_RegFile has a single register and DMA_VPL_RegFile has two registers, each 32-bit wide. 8 instances of DMA_VPL_RegFile need to be created.

```
 DMA_WPL_RegFile DMA_WPL_RF @0x00000300;
DMA_VPL_RegFile DMA_VPL_RF[8] @0x00000304
```

Since the total component size of one DMA_VPL_RegFile is 8 bytes, the address of this array needs to be 8 byte aligned if the default addressing of SystemRDL is used. Thus the above code will give an error if default addressing is used.

However, System RDL provides a *compact* addressing mode. This can be used for such register components where the offset needs to be continuous and not aligned with the size of the register component.

7.4 PERL Preprocessor

SystemRDL supports embedded Perl as preprocessor. This provides flexibility for file inclusion, text substitution, and conditional compilation. For instance, implementing different register blocks for different chip options is possible by using preprocessors. Each interface registers can be described in different files and if the current version of the chip has that interface then corresponding file can be included in the top level file by defining the option through command line during compile time as shown below.

Perl File Inclusion Example:

```
<%  if($PCI_E) { %>
`include "pci_e.rdl"
<%  }; %>
```

Define $PCI_E =1 to have the pci_e registers in the design.

PERL provides scalability in register and field definitions and allows control over the format of generated register names while instantiating a register multiple times.

For example **:**

```
<% for( $i = 0; $i < 2; $i += 1 ) { %>
dma_enable_status_reg
dma_<%=sprintf("%1x",$i)%>_enable_status
@0x<%=sprintf("%04x",((0x00C0)+($i*4)))%> ;
<% } %>
```

When processed, this is replaced by following code:

```
dma_enable_status_reg dma_0_enable_status @0x00C0
dma_enable_status_reg dma_1_enable_status @0x00C4
```

An example of text substitution on fields for part select is given below:

```
reg test_ob  {
   <%  for($i=0; $i <6;$i+=2) {  %>
   test_ob_sel  sel<%=$i%>  [<%=$i+1%>  : <%=$i%>];
   <%  } %>
```

When processed, this is replaced by following code

```
// Code resulting from embedded Perl script
test_ob_sel  sel0 [1:0];
test_ob_sel  sel2 [3:2];
test_ob_sel sel4[5:4];
```

7.5 MAPPING TO RTL

There are several constructs for defining registers and the RDL specification needs to be read to go through all in detail. However a snapshot of few common register types and their mapping to RTL which were used in our project is provided in Table 7.

Table 7: Useful SystemRDL constructs and their mapping to RTL

Register type	Field access property		Software access property	Interrupt field access property	Implemented hardware	Comments
	hw	sw				
control	r	rw	-	-	flip-flop	Software can read and write, hardware reads it to control design behavior
status	w	r	-	-	wire/bus hardware assigns value	Hardware updates, software reads it to know the design status
Self clearing	r	rw	singlepulse	-	flip-flop	Value written by software will be cleared in the next cycle
read on clear	w	rw	rclr	stickybit	flip-flop	Hardware update value is latched and cleared upon read by software by writing 0.
interrupt enable /mask	na	rw	-	-	flip-flop	Software writes enable/mask value of interrupt source
Interrupt status	w	rw	woclr	intr	flip-flop	Hardware update value is latched and cleared by software writing 1.

8. CONCLUSION

This flow eliminates tedious and error-prone processes of manually managing registers, and enables design, verification and firmware teams to work more efficiently from consistent and synchronized views of the chip design. It provides an almost zero-time, low maintenance, and reusable register design and verification (DV) system.

Once setup, the flow is repeatable and can be used across block, cluster (sub-chip level) and chip level, with lot of reuse of code and environment. Overall, there is lot of saving of effort, enhanced productivity, robust design and better verification. As seen earlier, greater than a month's effort can be cut down to less than a week's by this approach.

However, better tool support for backdoor paths and for multiple physical interface implementations would be very useful in making the flow more seamless and in reducing the effort required in implementing workarounds. We also look forward to some guidelines from SystemRDL for nested addressmap RTL implementation, register array and external register implementation. This would enable uniform tool outputs for these structures.

9. ACKNOWLEDGEMENTS

We would like to first express our gratitude to LSI and DVCon for giving us the opportunity to present our work and share our experiences with others who might benefit from it. We gratefully acknowledge the contribution of Jayendra Dwaraka, our manager for his encouragement, guidance and help. We also acknowledge the help received from Varun.S from Synopsys while setting up the RAL flow. Please note that any trademarks used herein are the property of their respective owners.

10. REFERENCES

[1] SystemRDL v1.0 :A Specification for a Register Description Language
[2] VMM Register Abstraction Layer User Guide, RAL Version 1.15