VMM RAL LCA Features

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

Contents

2

RTL Generation from a RALF Description

The RALF specification contains all of the necessary information to generate the RTL code implementing the specified registers. Automatically generating the RTL code and the RAL model ensures that they are kept up to date and requires that only one change be made manually.

This chapter contains the following sections:

RTL Generation Overview

The **`ralgen`** option, **`-R`**, can be used to generate the register RTL code for one or more blocks in the RALF file. When the **`-R`** option is specified, **`ralgen`** generates the register RTL code for the block specified as the top-level block or all the blocks and systems included in the specified top-level system.

The RTL code is generated in separate files for each system, block, register file, and register. The name of the generated file is `ral_`*`typ`*`_`*`name`*`_rtl.sv`, where *`typ`* is one of **`sys`**, **`blk`**, **`rfile`**, or **`reg`** and *`name`* is the unique, fully-scoped name of the system, block, register file or register, as in the RALF file.

The following are some of the key capabilities of RALF to RTL generator:

- RTL code is generated for registers containing the following types of fields:
  - RW fields
  - RO fields
  - WO fields
  - RU fields
  - RC fields
  - W1C fields
  - A0 fields
  - A1 fields

  - OTHER fields
  - USER0 fields
  - USER1 fields
  - USER2 fields
  - USER3 fields

- RTL code is generated for registers that are wider than the block data path, using LITTLE_ENDIAN or BIG_ENDIAN ordering.
- RTL code is generated for register arrays, register files, register files containing register arrays, and register file arrays.
- RTL code is generated for blocks with single and multiple domains and registers shared across multiple domains.
- RTL code is generated for blocks that have a wider datapath than the system they are instantiated in, using LITTLE_ENDIAN or BIG_ENDIAN ordering.
- Generated RTL can be instantiated multiple times in the same RTL design, at different base addresses.

Using the ‘ralgen –R’ Option

The **`ralgen`** command-line option **`-R`** *`dir-name`* must be used to trigger or request RALF to RTL generation.

- *`dir-name`* specifies the folder in which generated RTL files are to be stored.

- **ralgen** command line option **-l sv** is implicit when using the **-R** option but can be specified as well. Only SystemVerilog RTL is supported.
- Using the **-R** option prevents the generation of the RAL model. It cannot be used in conjunction with **ralgen** options **-g**/**gen_c**, **-b**, **-d**/**top_domain**, **-e/ext_ud** or **-c**.

The generated RTL is of a predetermined style and assumes a specific read/write protocol. You must implement a suitable decoding function between the actual bus interface used by the block and the generated register RTL.

The generated register RTL code uses a single clock signal. You are responsible for any clock domain crossing issues between the generated register RTL and the clock domain of the bus interface.

All generated files are written in the directory specified with the **–R** option. Each file contains a single module and is named according to the module name with the `.sv` suffix. This allows you to override a generated module with a user-provided one by locating the user-defined module in a directory that is searched before the **ralgen** generated RTL directory, using the **+incdir** option for simulation, or the *search-path* variable in synthesis.

This section contains the following topics:

- “User RTL” on page ix
- “Host-Side Protocol” on page xi
- “Design-Side Protocol” on page xii
- “Design-Side Protocol for External Fields” on page xv

User RTL

To use the generated RTL modules that implement the registers and address decoders specified in the RALF file, instantiate the appropriate block and host interfaces and block and system modules. As long as the system and block type names do not change, it is not necessary to change your code. As registers or fields are modified, the corresponding design-side signals located in the block interfaces will be modified. The latter changes may require modifications in your code, but only for the portions that make use of the changed registers or fields.

For example, the generated RTL from the following RALF file:

```
system s1 {
   bytes 1;
   system s2 @’h1000 {
      bytes 1
      block b4;
   }
   block b1 @’h2000;
}
```

can be instantiated in your RTL as follows:

File b1.sv:

```
`include "vmm_ral_host_itf.sv"
`include "ral_blk_b1_rtl.sv"

module b1(vmm_ral_host_itf.slave hst, ...);

ral_blk_b1_itf ral_io();
ral_blk_b1_rtl ral(hst, ral_io.regs);
...
endmodule
```

File b4.sv:

```
`include "vmm_ral_host_itf.sv"
`include "ral_blk_b4_rtl.sv"

module b4(vmm_ral_host_itf.slave hst, ...);

ral_blk_b4_itf ral_io();
ral_blk_b4_rtl ral(hst, ral_io.regs);
...
endmodule
```

File s1.sv:

```
`include "vmm_ral_host_itf.sv"
`include "ral_sys_s1_rtl.sv"

`include "b1.sv"
`include "s2.sv"

module s1(input bit clk, input bit rstn, ...);

vmm_ral_host_itf hst_bus(clk, rstn);
vmm_ral_host_itf s2_bus(clk, rstn);
vmm_ral_host_itf b1_bus(clk, rstn);
ral_sys_s1_rtl ral(hst_bus.slave, s2_bus.master,
b1_bus.master);
s2 s2_i(s2_bus.slave);
b1 b1_i(b1_bus.slave);
...
endmodule
```

File s2.sv:

```
`include "vmm_ral_host_itf.sv"
`include “ral_sys_s1_s2_rtl.sv”
`include "b4.sv"

module s2(vmm_ral_host_itf.slave hst, ...);

vmm_ral_host_itf b4_bus(hst.clk, hst.rstn);
ral_sys_s1_s2_rtl ral(hst, b4_bus.master);
b4 b4_i(b4_bus.slave);
...
endmodule
```

Host-Side Protocol

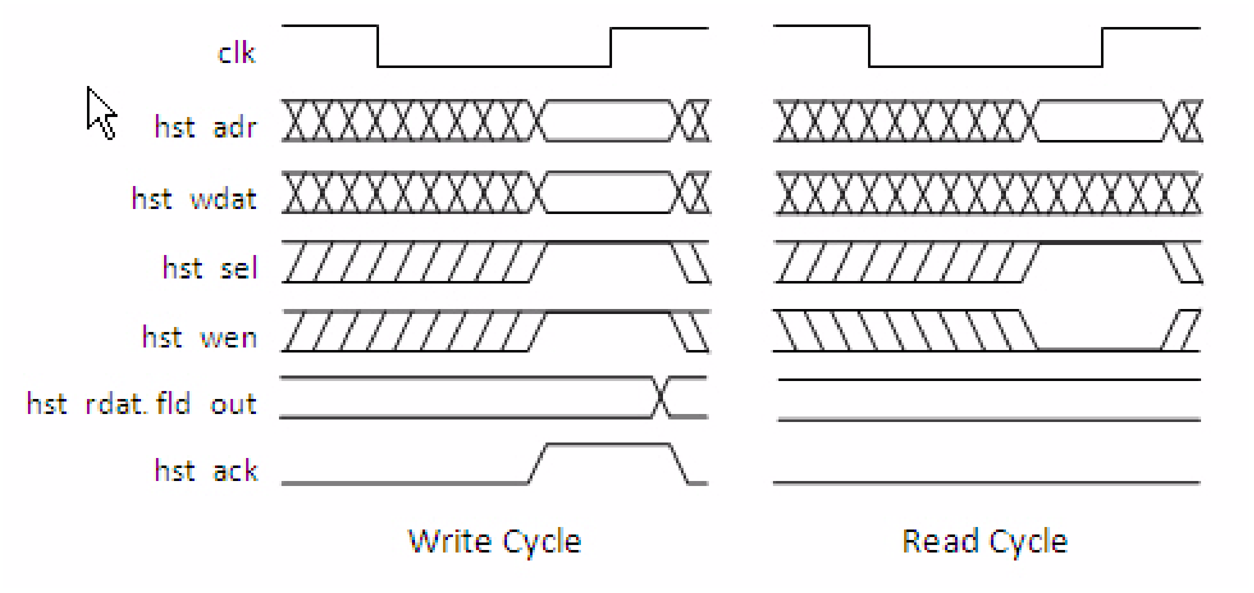
The generated RTL assumes that the host-side protocol is implemented using the signals shown in Table 2-1. Unless otherwise specified, all signals are active high and are sampled at the rising edge of the clock. The direction is specified with respect to the slave size.

Table 2-1 Host-Side Signals

Name	Width	Direction	Description
hst_adr	M	Input	Address offset within the register file, block, or system.
hst_wdat	N	Input	Write data value. How this value is interpreted depends on the type of field being written.
hst_sel	B	Input	When HIGH, the field or corresponding byte lane in the register, block, or system is selected. When LOW, the field or byte lane is not selected and the corresponding “hst_rdat” value is ignored.
hst_wen	I	Input	When HIGH, indicates a write cycle. When LOW, indicates a read cycle.
hst_rdat fld_out	N	Output	Current value in the field or register. Current value of the selected register in the block or system.
hst_ack	I	Output	When HIGH, acknowledges completion of a write cycle. Read cycles are not acknowledged.

The default implementation for the fields assumes that the host-side protocol for read and write accesses is as shown in Figure 2-1.

Figure 2-1   Host-Side Protocol



Design-Side Protocol

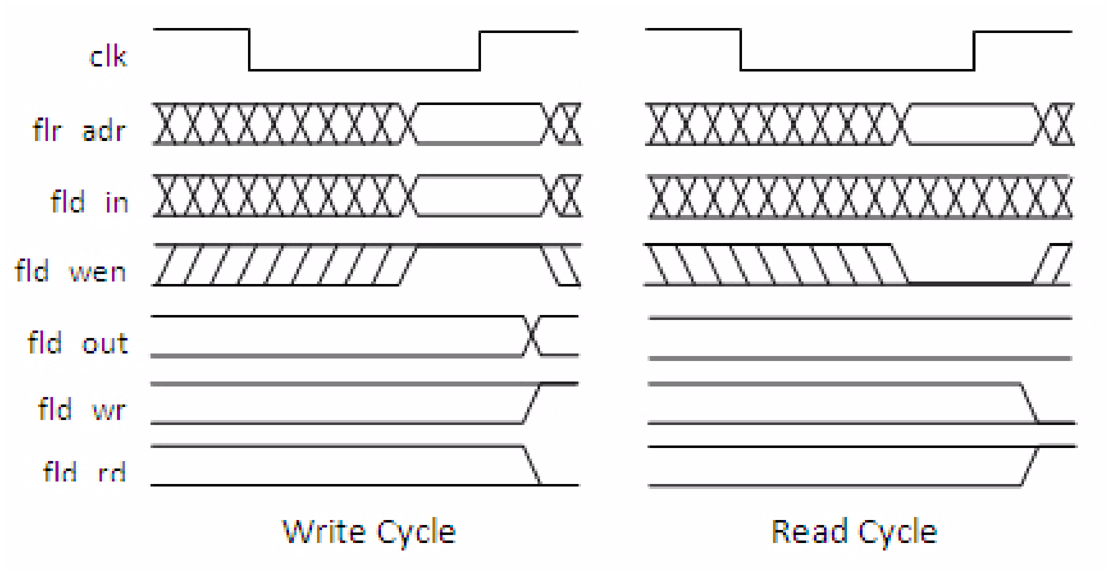
The generated RTL provides the design side with the signals in Table 2-2 (where functionally relevant). Unless otherwise specified, all signals are active high and are sampled at the rising edge of the clock. The direction is specified with respect to the generated RTL code. It is not necessary for a design to make use of all output signals, but all input signals must be driven. If a function provided by an input signal is not used, it must be driven to an inactive state.

Table 2-2 Design-Side Signals

Name	**Width**	**Direction**	**Description**
fld_adr	M	Output	Address offset, within the memory.
fld_in	N	Input	Update data value. How this value is interpreted depends on the type of field being updated.
fld_wen	1	Input	When HIGH, the field, register, block, or system is to be updated, based on the "fld_in" value.
fld_wr	1	Output	A write cycle was performed on a system, block, memory, or register.
fld_rd	1	Output	A read cycle was performed on a system, block, memory, or register.
fld_out	N	Output	Current value in the field or register.

The default implementation for the fields assumes that the design-side protocol for read and write accesses (where relevant) is as shown in Figure 2-2.

Figure 2-2 Design-Side Protocol



Only the design-side signals required by the functionality of a field are generated. Table 2-3 defines which design-side signals are included for the various field types. It also shows how the input value is interpreted by the default implementation of the field.

Table 2-3 Design-Side Signals by Field Type

Name	**RW**	**RO**	**WO**	**RC**	**W1C**	**A0**	**A1**	**RU**
fld_in	As-is	As-is	No	Set Mask	Set Mask	Clr Mask	Set Mask	As-is
fld_wen	Yes	No	No	Yes	Yes	Yes	Yes	Yes
fld_wr	Yes	No	Yes	No	Yes	Yes	Yes	No
fld_rd	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
fld_out	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes

Design-Side Protocol for External Fields

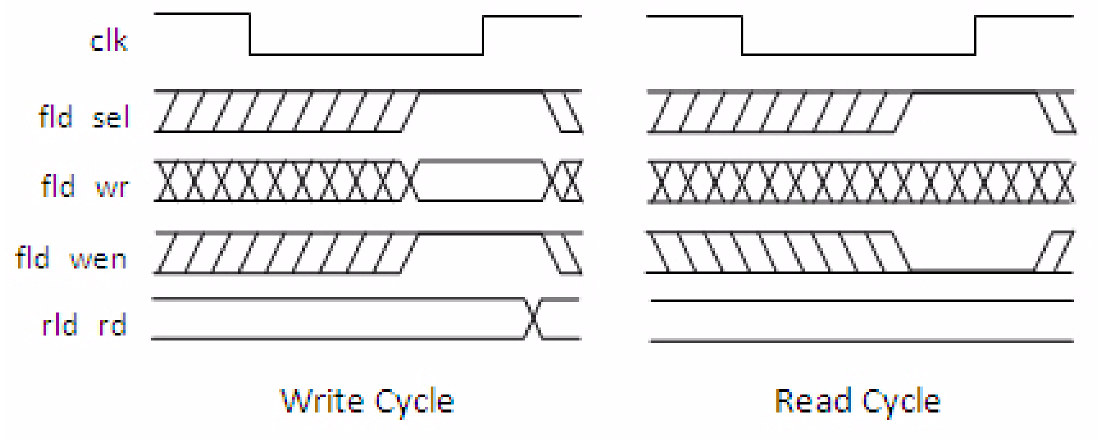
For fields that are assumed to be implemented by the design, and not in the generated RTL, the generated RTL provides the design side with the signals shown in Table 2-4 (where functionally relevant). Unless otherwise specified, all signals are active high and are sampled at the rising edge of the clock. The direction is specified with respect to the generated RTL code. It is not necessary for a design to make use of all output signals, but all input signals must be driven. If a function provided by an input signal is not used, it must be driven to an inactive state.

Table 2-4 Design-Side Signals in the Generated RTL

Name	**Width**	**Direction**	**Description**
fld_sel	1	Output	When HIGH, the field is selected and must be read or written by the design. When LOW, all other signals should be ignored by the design.
fld_wdat	N	Output	Write data value.
fld_wen	1	Input	When HIGH, indicates a write cycle. When LOW, indicates a read cycle.
fld_rdat	N	Input	Current value in the field.

The generated RTL assumes that the design-side protocol for read and write accesses of an externally-implemented field are as shown in Figure 2-3.

Figure 2-3   Design-Side Protocol for External Fields



An external field may not span multiple physical addresses in the generated RTL.

Data Input and Output Requirements

The only user input requirement is a syntactically correct RALF file. No modifications are necessary.

An auxiliary input is a set of files containing the RTL implementation for fields based on the field type. Each file, named `vmm_ral_`*`fieldtype`*`_field_rtl.sv`, must contain the definition of a module named `vmm_ral_`*`fieldtype`*`_field_rtl`, where *`fieldtype`* is the type of the field (for example, **`rw`**, **`wo`**, and so on). A default implementation is provided in a VMM distribution and included in Appendix D. It is possible to override these default implementations by providing a different implementation that must be picked up before the default ones via a suitable module search order (for example the **`-y`** VCS command-line option).

The following sections describe the RTL code generated that corresponds to the various RALF constructs.

This section contains the following topics:

Registers

For every **`register`** definition, a module is generated. The module is named according to the unique scoped register type name and contains instances of the appropriate field modules. A single set of host-side signals is included in the module pins. A set of design-side signals is included for each field in the register.

The RTL generation template for a register is:

```
module ral_reg_<regname>_rtl(<host-side signals>,
                             {<design-side signals>});

{<field module instantiations>}
{<field rd/wr notifiers>}
```

```
<read value concatenation>
endmodule
```

For example, the following RALF declaration:

```
register r1 {
   bytes 4;
   field f10 @0  {bits 8; access rw;}
   field f11 @16 {bits 8; access ro;}
}
```

will generate the following corresponding RTL:

```
module ral_reg_r1_rtl(input          clk,
                      input          rstn,
                      input   [31:0] hst_wdat,
                      output  [31:0] hst_rdat,
                      input   [ 3:0] hst_sel,
                      input          hst_wen,
                      output  [ 7:0] f10_out,
                      output         f10_rd, f10_wr,
                      input   [ 7:0] f10_in,
                      input          f10_wen,
                      input   [ 7:0] f11_in);

vmm_ral_rw_field_rtl #(8, 8’b0)
   f10(clk, rstn, f10_out,
       hst_wdat[7:0], hst_sel[0], hst_wen,
       f10_in, f10_wen);

wire   [ 7:0] f11_out;
vmm_ral_ro_field_rtl #(8)
   f11(clk, rstn, f11_in, f11_out);

vmm_ral_notifier_rtl _n(clk, rstn, hst_sel[0], hst_wen,
                        f10_rd, f10_wr);

assign hst_rdat[31:0] = {f11_out, 8’b0, f10_out};

endmodule
```

Shared registers are generated in an identical fashion: the sharing is implemented in the block RTL code.

Registers with External Fields

Fields of type OTHER and USER are assumed to be implemented by the design. Therefore, their design-side signals are those of an external field instead of those of a type-specific internal field.

For example, the following RALF declaration:

```
register r2 {
   bytes 1;
   field f20 {bits 2; access user0; }
   field f21 {bits 3; access ru; }
   field f22 {bits 3; access other;}
}
```

will generate the following corresponding RTL:

```
module ral_reg_r2_rtl(input        clk,
                      input        rstn,
                      input  [7:0] hst_wdat,
                      output [7:0] hst_rdat,
                      input  [0:0] hst_sel,
                      input        hst_wen,
                      output       f20_sel,
                      output [1:0] f20_wdat,
                      output       f20_wen,
                      input  [1:0] f20_rdat,
                      output [2:0] f21_out,
                      output       f21_rd,
                      input  [2:0] f21_in,
                      input        f21_wen,
                      output       f22_sel,
                      output [2:0] f22_wdat,
                      output       f22_wen,
                      input  [2:0] f22_rdat);

assign f20_sel  = hst_sel[0];
assign f20_wdat = hst_wdat[1:0];
assign f20_wen  = hst_wen;

vmm_ral_rw_field_rtl #(3, 2'b0)
   f21(clk, rstn, f21_out,
       hst_wdat[4:2], hst_sel[0], hst_wen,
       f21_in, f21_wen);

assign f22_sel  = hst_sel[0];
assign f22_wdat = hst_wdat[7:5];
assign f22_wen  = hst_wen;

vmm_ral_notifier_rtl _n(clk, rstn, hst_sel[0], hst_wen,
                        f21_rd, /*open*/);

assign hst_rdat[7:0] = {f22_rdat, f21_in, f20_rdat};

endmodule
```

Fields Spanning Byte Boundaries

To support combining registers, blocks, and systems of different bus widths, individual byte lane selection signals are used in the generated RTL. As long as narrower buses are instantiated in wider buses, there are no problems, but when a wider bus is instantiated in a narrower bus, accesses to the registers must be broken up into multiple accesses. If a field spans a byte boundary, it might straddle different physical addresses in the final RTL. In that case, a warning is issued by the code generator and the field is split across multiple fields wholly contained in a byte lane.

Because there is only one **`fld_wr`** and **`fld_rd`** notification flag per field, a warning will be issued by ralgen for every field spanning multiple physical addresses.

External fields must always be accessible via a single physical address. An error will be issued by **`ralgen`** if RTL code is generated for a RALF description containing an external field that spans multiple physical addresses.

For example, the following RALF declaration:

```
register r3 {
   bytes 2;
   field f30 {bits 16; access rw; }
}
```

will generate the following corresponding RTL:

```
module ral_reg_r3_rtl(input         clk,
                      input         rstn,
                      input  [31:0] hst_wdat,
                      output [31:0] hst_rdat,
                      input  [ 3:0] hst_sel,
                      input         hst_wen,
                      output [15:0] f30_out,
                      output        f30_rd, f30_wr,
                      input  [15:0] f30_in,
                      input         f30_wen);

vmm_ral_rw_field_rtl #(8, 8'b0)
   f30_7_0(clk, rstn, f30_out[7:0],
           hst_wdat[7:0], hst_sel[0], hst_wen,
           f30_in[7:0], f30_wen);
vmm_ral_rw_field_rtl #(8, 8'b0)
   f30_15_8(clk, rstn, f30_out[15:8],
            hst_wdat[15:8], hst_sel[1], hst_wen,
            f30_in[15:8], f30_wen);

vmm_ral_notifier_rtl _n(clk, rstn, |hst_sel[1:0], hst_wen,
                        f30_rd, f30_wr);

assign hst_rdat[15:0] = {f30_out};

endmodule
```

Blocks

For every **`block`** definition, an interface and a module are generated. The interface and module are named according to the unique scoped block type name. The module contains instances of the appropriate register modules. The interface contains a set of design-side signals for each field in the block. The module pins include a host-side slave interface and a block design-side interface.

The RTL generation template for a block is:

```
interface ral_blk_<blkname>_itf();
   {<design-side signals>});
endinterface

module ral_blk_<blkname>_rtl({<host-side slave interface>},
                              <design-side interface>);
<address decoding>
{<register module instantiations>}
{<memory signals decoding>}
{<register muxing>
endmodule
```

For example, the following RALF declaration:

```
block b1 {
   bytes 2;
   register r1;
   register r2=rx;
   register r2=ry @'h0100;
}
```

will generate the following corresponding RTL:

```
interface ral_blk_b1_itf();

logic [7:0] f10_out;
logic       f10_rd, f10_wr;
logic [7:0] f10_in;
logic       f10_wen;
logic [7:0] f11_in;

logic       rx_f20_sel, rx_f20_wen;
logic [1:0] rx_f20_wdat;
logic [1:0] rx_f20_rdat;
logic [2:0] rx_f21_out;
logic       rx_f21_rd;
logic [2:0] rx_f21_in;
logic       rx_f21_wen;
logic       rx_f22_sel, rx_f22_wen;
logic [2:0] rx_f22_wdat;
logic [2:0] rx_f22_rdat;
logic       ry_f20_sel, ry_f20_wen;
logic [1:0] ry_f20_wdat;
logic [1:0] ry_f20_rdat;
logic [2:0] ry_f21_out;
logic       ry_f21_rd;
logic [2:0] ry_f21_in;
logic       ry_f21_wen;
logic       ry_f22_sel, ry_f22_wen;
logic [2:0] ry_f22_wdat;
logic [2:0] ry_f22_rdat;

modport regs(output f10_out, f10_rd, f10_wr,
             input  f10_in, f10_wen,
             input  f11_in,
             output rx_f20_sel, rx_f20_wen, rx_f20_wdat,
             input  rx_f20_rdat,
             output rx_f21_out, rx_f21_rd,
             input  rx_f21_in, rx_f21_wen,
             output rx_f22_sel, rx_f22_wen, rx_f22_wdat,
             input  rx_f22_rdat,
             output ry_f20_sel, ry_f20_wen, ry_f20_wdat,
             input  ry_f20_rdat,
             output ry_f21_out, ry_f21_rd,
             input  ry_f21_in, ry_f21_wen,
             output ry_f22_sel, ry_f22_wen, ry_f22_wdat,
             input  ry_f22_rdat);

modport usr(input  f10_out, f10_rd, f10_wr,
            output f10_in, f10_wen,
            output f11_in,
            input  rx_f20_sel, rx_f20_wen, rx_f20_wdat,
            output rx_f20_rdat,
            input  rx_f21_out, rx_f21_rd,
            output rx_f21_in, rx_f21_wen,
            input  rx_f22_sel, rx_f22_wen, rx_f22_wdat,
            output rx_f22_rdat,
            input  ry_f20_sel, ry_f20_wen, ry_f20_wdat,
            output ry_f20_rdat,
            input  ry_f21_out, ry_f21_rd,
            output ry_f21_in, ry_f21_wen,
            input  ry_f22_sel, ry_f22_wen, ry_f22_wdat,
            output ry_f22_rdat);

endinterface
```

```
module ral_blk_b1_rtl(vmm_ral_host_itf.slave  hst,
                      ral_blk_b1_itf.regs     usr);

reg [3:0] r1_sel;
reg rx_sel, ry_sel;
always @(*)
begin
   r1_sel = 0;
   rx_sel = 0;
   ry_sel = 0;
   hst.ack = 0;
   case (hst.adr)
      0: begin
           r1_sel = {2'b00, hst.sel[1:0]};
           hst.ack = hst.wen;
         end
      1: begin
           r1_sel = {hst.sel[1:0], 2'b00};
           hst.ack = hst.wen;
         end
      2: begin
           rx_sel = 1;
           hst.ack = hst.wen;
         end
      `h0100: begin
                ry_sel = 1;
                hst.ack = hst.wen;
              end
   endcase
end

wire [31:0] r1_out;
ral_reg_r1_rtl r1(hst.clk, hst.rstn,
                  {hst.wdat[15:0], hst.wdat[15:0]},
                  r1_out, r1_sel, hst.wen,
                  usr.f10_out, usr.f10_rd, usr.f10_wr,
                  usr.f10_in, usr.f10_wen, usr.f11_in);

wire [7:0] rx_out;
ral_reg_r2_rtl rx(hst.clk, hst.rstn,
                  hst.wdat[7:0], rx_out, rx_sel, hst.wen,
           usr.rx_f20_sel, usr.rx_f20_wdat, usr.rx_f20_wen,
            usr.rx_f20_rdat, usr.rx_f21_out, usr.rx_f21_rd,
             usr.rx_f21_in, usr.rx_f21_wen, usr.rx_f22_sel,
                  usr.rx_f22_wdat, usr.rx_f22_wen,
usr.rx_f22_rdat);
```

```
wire [7:0] ry_out;
ral_reg_r2_rtl ry(hst.clk, hst.rstn,
                  hst.wdat[7:0], ry_out, ry_sel, hst.wen,
           usr.ry_f20_sel, usr.ry_f20_wdat, usr.ry_f20_wen,
            usr.ry_f20_rdat, usr.ry_f21_out, usr.ry_f21_rd,
             usr.ry_f21_in, usr.ry_f21_wen, usr.ry_f22_sel,
                  usr.ry_f22_wdat, usr.ry_f22_wen,
usr.ry_f22_rdat);

reg [15:0] _rdat;
always @(*)
begin
   _rdat = 16'b0;
   unique casez ({|r1_sel[3:2], |r1_sel[1:0], |rx_sel,
|ry_sel})
      4'b1???: _rdat = r1_out[31:16];
      4'b?1??: _rdat = r1_out[15:0];
      4'b??1?: _rdat[7:0] = rx_out;
      4'b???1: _rdat[7:0] = ry_out;
      4'b0000: _rdat = 15'b0;
   endcase
end

assign hst.rdat[15:0] = _rdat;

endmodule
```

Memories

All memory instances are assumed to be external. For each memory instance, a set of design-side signals is included in the block interface.

For example, the following RALF declaration:

```
block b2 {
   bytes 4;
   register r1;
   memory m1 @'h0100 {
      bits 16;
      size 256;
      access rw;
   }
   memory m2 @'h1000 {
      bits 23;
      size 1k;
      access ro;
   }
}
```

will generate the following corresponding RTL:

```
interface ral_blk_b2_itf();

logic [7:0] f10_out;
logic       f10_rd, f10_wr;
logic [7:0] f10_in;
logic       f10_wen;
logic [7:0] f11_in;

logic [ 1:0] m1_sel;
logic [ 7:0] m1_adr;
logic [15:0] m1_rdat;
logic [15:0] m1_wdat;
logic        m1_wen;

logic [ 2:0] m2_sel;
logic [ 9:0] m2_adr;
logic [22:0] m2_rdat;

modport regs(output f10_out, f10_rd, f10_wr,
             input  f10_in, f10_wen,
             input  f11_in,
             output m1_sel, m1_adr, m1_wdat, m1_wen,
             input  m1_rdat,
             output m2_sel, m2_adr,
             input  m2_rdat);

modport usr(input  f10_out, f10_rd, f10_wr,
            output f10_in, f10_wen,
            output f11_in,
            input  m1_sel, m1_adr, m1_wdat, m1_wen,
            output m1_rdat,
            input  m2_sel, m2_adr,
            output m2_rdat);
```

```
endinterface

module ral_blk_b2_rtl(vmm_ral_host_itf.slave  hst,
                      ral_blk_b2_itf.regs     usr);

reg [3:0] r1_sel;
reg [1:0] m1_sel;
reg [2:0] m2_sel;
always @(*)
begin
   r1_sel = 0;
   m1_sel = 0;
   m2_sel = 0;
   hst.ack = 0;
   case (hst.adr)
      0: begin
            r1_sel = hst.sel[3:0];
            hst.ack = hst.wen;
         end
   endcase
   if (hst.adr >= 'h0100 && hst.adr <= 'h01FF) begin
      m1_sel = hst.sel[1:0];
      hst.ack = hst.wen;
   end
   if (hst.adr >= 'h1000 && hst.adr <= 'h13FF) begin
      m2_sel = hst.sel[2:0];
      hst.ack = hst.wen;
   end
end

wire [31:0] r1_out;
ral_reg_r1_rtl r1(hst.clk, hst.rstn, hst.wdat[31:0],
                  r1_out, r1_sel, hst.wen,
           usr.f10_out, usr.f10_rd, usr.f10_wr, usr.f10_in,
                  usr.f10_wen, usr.f11_in);

assign usr.m1_adr  = hst.adr - 'h0100;
assign usr.m1_wdat = hst.wdat[15:0];
assign usr.m1_wen  = hst.wen;

assign usr.m2_adr  = hst.adr - 'h1000;

reg [31:0] _rdat;
always @(*)
begin
```

```
   unique casez ({|r1_sel, |m1_sel, |m2_sel})
      3'b1??: _rdat = r1_out;
      3'b?1?: _rdat[15:0] = usr.m1_rdat;
      3'b??1: _rdat[21:0] = usr.m2_rdat;
      3'b000: _rdat = 32'b0;
   endcase
end

assign hst.rdat = _rdat;

endmodule
```

Shared memories are generated in an identical fashion: the sharing is implemented in the block RTL code.

Multiple Domains

The arbitration between different physical interfaces for different domains is handled by the generated RTL code for the multi-domain block. All domains are assumed to conform to the host-side protocol and are synchronized to the same clock. For each domain, a host-side slave interface is included in the block module pins.

When multiple domains attempt to write a shared register during the same cycle, the domain declared first has priority.

For example, the following RALF declaration:

```
register r4 {
   bytes 1;
   field f1 {bits 8; access ro};
}

register r5 {
   bytes 1;
   field f2 {bits 8; access rw};
   shared;
}
block b3 {
   domain north {
      bytes 1;
      register r4=r1;
      register r5=r2;
   }
   domain south {
      bytes 1;
      register r4=r3;
      register r5=r2;
   }
}
```

will generate the following corresponding RTL:

```
module ral_blk_b3_rtl(vmm_ral_host_itf.slave north,
                      vmm_ral_host_itf.slave south,
                      ral_blk_b3_itf.regs    usr);

reg r1_sel;
reg north_r2_sel;
reg south_r2_sel;
reg [7:0] r2_in;
reg       r2_wen;
reg r3_sel;
always @(*)
begin
   r1_sel = ‘b0;
   north_r2_sel = ‘b0;
   south_r2_sel = ‘b0;
   r2_in  = ‘bx;
   r2_wen = ‘b0;
   r3_sel = ‘b0;
   north.ack = 0;
   south.ack = 0;

   case (south.adr)
      0: begin
            r3_sel = south.sel[0];
            south.ack = south.wen;
          end
      1: begin
            south_r2_sel = south.sel[0];
            r2_in  = south.wdat[7:0];
            r2_wen = south.wen;
            south.ack = south.wen;
         end
   endcase
   case (north.adr)
      0: begin
            r1_sel = north.sel[0];
            north.ack = north.wen;
         end
      1: begin
            north_r2_sel = north.sel[0];
            if (north.wen) begin
               if (r2_wen) south.ack = 0;
               r2_wen = 1;
               r2_in  = north.wdat[7:0];
               north.ack = 1;
            end
         end
   endcase
end

wire [7:0] r1_out;
ral_reg_r4_rtl r1(north.clk, north.rstn,
                 north.wdat[7:0], r1_out, r1_sel, north.wen,
                  usr.r1_f1_in);

wire [7:0] r2_out;
wire r2_sel = north_r2_sel | south_r2_sel;
ral_reg_r5_rtl r2(north.clk, north.rstn,
                  r2_in, r2_out, r2_sel, r2_wen,
               usr.f2_out, usr.f2_rd, usr.f2_wr, usr.f2_in,
                  usr.f2_wen);

wire [7:0] r3_out;

ral_reg_r4_rtl r3(south.clk, south.rstn,
                 south.wdat[7:0], r3_out, r3_sel, south.wen,
                  usr.r3_f1_in);

reg [7:0] _north_rdat;
always @(*)
begin
   unique casez ({r1_sel, r2_sel})
      2'b1?: _north_rdat = r1_out;
      2'b?1: _north_rdat = r2_out;
      2'b00: _north_rdat = 8'b0;
   endcase
end
```

```
assign north.rdat[7:0] = _north_rdat;

reg [7:0] _south_rdat;
always @(*)
begin
   unique casez ({r1_sel, r2_sel})
      2'b1?: _south_rdat = r3_out;
      2'b?1: _south_rdat = r2_out;
      2'b00: _south_rdat = 8'b0;
   endcase
end
assign south.rdat[7:0] = _south_rdat;

endmodule
```

Register Arrays

Register arrays present a challenge because they can easily create a large number of fields. One possibility would be to flatten the register arrays, but this would create a large number of signals in the block interface. Another approach would be to add an array offset to the design-side interface and consider its value whenever a field in the register array is read or written from the design. However, this would limit access to a single register in the array per clock cycle and make it impossible to concurrently access different fields at different offsets in the array.

Register arrays are implemented using generated instances of the individual register modules and the corresponding design-side signals are provided as arrays.

For example, the following RALF declaration:

```
block b4 {
   bytes 1;
   register r4[256];
}
```

will generate the following corresponding RTL:

```
interface ral_blk_b4_itf();

logic [7:0] f1_in[256];

modport regs(input f1_in);

modport usr(output f1_in);

endinterface

module ral_blk_b4_rtl(vmm_ral_host_itf.slave hst,
                      ral_blk_b4_itf.regs    usr);

reg [7:0] _r4_sel_i;
reg r4_sel[256];
always @(*)
begin
   reg [31:0] i;
   hst.ack = 0;
   _r4_sel_i = 'bx;
   for (i = 0; i < 256; i = i + 1) begin
      r4_sel[i] = 'b0;
      if (hst.adr == 0 + i*1) begin
         r4_sel[i] = hst.sel[0];
         _r4_sel_i = i;
         hst.ack = hst.wen;
      end
   end
end

wire [7:0] r4_out[256];
genvar _r4_i;
generate for (_r4_i=0; _r4_i<256; _r4_i=_r4_i+1)
   begin: r4
      ral_reg_r4_rtl i(hst.clk, hst.rstn, hst.wdat[7:0],
                    r4_out[_r4_i], r4_sel[_r4_i], hst.wen,
                       usr.f1_in[_r4_i]);
   end
endgenerate

reg [7:0] _rdat;
always @(*)
```

```
begin: muxout
   unique casez ({|r4_sel[_r4_sel_i]})
      1'b1: _rdat = r4_out[_r4_sel_i];
      1'b0: _rdat = 8'b0;
   endcase
end
assign hst.rdat[7:0] = _rdat;

endmodule
```

Register Files

For every **`regfile`** definition, a module is generated. The module is named according to the unique scoped regfile type name and contains instances of the appropriate register modules. A set of host-side slave interfaces is included in the module pins. A set of design-side signals is included for each field in the register file.

The RTL generation template for a register file is:

```
module ral_rfile_<filename>_rtl(<host-side signals>,
                                {<design-side signals>});
<address decoding>
{<register module instantiations>}
<register muxing>
endmodule
```

For example, the following RALF declaration:

```
block b5 {
   bytes 1;
   regfile rf {
      register r4;
      register r4=r5;
   }
}
```

will generate the following corresponding RTL:

```
module ral_rfile_b5_rf_rtl(input         clk,
                          input          rstn,
                  input  [`VMM_RAL_ADDR_WIDTH-1:0] hst_adr,
                          input  [31:0] hst_wdat,
                          output logic [31:0] hst_rdat,
                          input  [ 3:0] hst_sel,
                          input         hst_wen,
                          output logic  hst_ack,
                          input  [ 7:0] r4_f1_in,
                          input  [ 7:0] r5_f1_in);

reg r4_sel;
reg r5_sel;
always @(*)
begin
   r4_sel = 0;
   r5_sel = 0;
   hst_ack = 0;
   case (hst_adr)
      0: begin
            r4_sel = hst_sel[0];
            hst_ack = hst_wen;
         end
      1: begin
            r5_sel = hst_sel[0];
            hst_ack = hst_wen;
         end
   endcase
end

wire [7:0] r4_out;
ral_reg_r4_rtl r4(clk, rstn, hst_wdat[7:0],
                  r4_out, r4_sel, hst_wen,
                  r4_f1_in);

wire [7:0] r5_out;
ral_reg_r4_rtl r5(clk, rstn, hst_wdat[7:0],
                  r5_out, r5_sel, hst_wen,
                  r5_f1_in);

reg [7:0] _rdat;
always @(*)
begin
   unique_casez ({|r4_sel, |r5_sel})
      2'b1?: _rdat = r4_out;
      2'b?1: _rdat = r5_out;
      2’b00: _rdat = 8’b0;
   endcase
end
assign hst_rdat[7:0] = _rdat;

endmodule
```

Register files are typically used as register file arrays. A simple register file will be generated as if it were a register file array of size 1.

Register File Arrays

Register file arrays are implemented in a fashion similar to register arrays.

For example, the following RALF declaration:

```
block b5 {
   bytes 1;
   regfile rf[256] {
      register r4;
      register r4=r5;
   }
}
```

will generate the following corresponding RTL:

```
interface ral_blk_b5_itf();

logic [7:0] r4_f1_in[256];
logic [7:0] r5_f1_in[256];

modport regs(input r4_f1_in,
             input r5_f1_in);

modport usr(output r4_f1_in,
            output r5_f1_in);

endinterface
```

```
module ral_blk_b5_rtl(vmm_ral_host_itf.slave hst,
                      ral_blk_b5_itf.regs    usr);

reg [7:0] _rf_sel_i;
reg rf_sel[256];
always @(*)
begin
   reg [31:0] i;
   hst.ack = 0;
   _rf_sel_i = ‘bx;
   for (i = 0; i < 256; i = i + 1) begin
      rf_sel[i] = ‘b0;
      if (hst.adr == 0 + i*1) begin
         rf_sel[i] = hst.sel[0];
         _rf_sel_i = i;
         hst.ack = hst.wen;
      end
   end
end

wire [7:0] rf_out[256];
genvar _rf_i;
generate for (_rf_i=0; _rf_i<256; _rf_i=_rf_i+1)
   begin: r1
      ral_rfile_b5_rf_rtl i(hst.clk, hst.rstn, hst.adr - 0,
                             hst.wdat[7:0], rf_out[_rf_i],
                             rf_sel[_rf_i], hst.wen,
                usr.r4_f1_in[_rf_i], usr.r5_f1_in[_rf_i]);
   end
endgenerate

reg [7:0] _rdat;
always @(*)
begin: muxout
   unique casez ({|rf_sel[_rf_sel_i]})
      1’b1: _rdat = rf_out[_rf_sel_i];
      1’b0: _rdat = 8’b0;
   endcase
end
assign hst.rdat[7:0] = _rdat;

endmodule
```

Systems

For every **`system`** definition, a module is generated. The module is named according to the unique scoped system type name. A host-side slave interface for each domain is included in the module pins. A host-side master interface is included for each module and sub-system instantiated in the system.

The RTL generation template for a system is:

```
module ral_sys_<sysname>_rtl({<slave host-side interface>},
                             {<master host-side interface>});

<address decoding>
<readback muxing>
endmodule
```

For example, the following RALF declaration:

```
system s1 {
   bytes 1;
   system s2 @’h1000 {
      bytes 1
      block b4;
   }
   block b1 @’h2000;
}
```

will generate the following corresponding RTL:

```
module ral_sys_s1_rtl(vmm_ral_host_itf.slave hst,
                      vmm_ral_host_itf.master s2,
                      vmm_ral_host_itf.master b1);

always @(*)
begin
   s2.sel[0] = ‘b0;
   b1.sel[1:0] = ‘b0;
   if (hst.adr >= ‘h1000 && hst.adr <= ‘h1000 + 255) begin
      s2.sel[0] = hst.sel[0];
   end
   if (hst.adr >= ‘h2000 && hst.adr <= ‘h2000 + 7) begin
      b1.sel[1] = hst.sel[0] & hst.adr[0];
      b1.sel[0] = hst.sel[0] & ~hst.adr[0];
   end
end

assign s2.adr  = hst.adr - ‘h1000;
assign s2.wdat = hst.wdat;
assign s2.wen  = hst.wen;

assign b1.adr  = (hst.adr - ‘h2000) >> 1;
assign b1.wdat = {hst.wdat[7:0], hst.wdat[7:0]};
assign b1.wen  = hst.wen;

reg [7:0] _rdat;
always @(*)
begin: muxout
   unique casez ({s2.sel[0], b1.sel[1:0] })
      3’b1??: begin
                 _rdat = s2.rdat;
                 hst.ack = s2.ack;
              end
      3’b?1?: begin
                 _rdat = b1.rdat[15:8];
                 hst.ack = b1.ack;
              end
      3’b??1: begin
                 _rdat = b1.rdat[7:0];
                 hst.ack = b1.ack;
              end
      3’b000: begin
                 _rdat = 8’b0;
                 hst.ack = 0;
              end
   endcase
end
assign hst.rdat[7:0] = _rdat;

endmodule
```

Unsupported RALF Constructs and Functionality

The following RALF constructs or RAL functionality are explicitly not supported and will be either ignored or a warning will be issued.

- Virtual registers
- Register files with a mix of shared and unshared registers
- Soft reset values
- Programmable base offset
- Programmable reset value

Appendix: RTL Implementation of RALF Fields

Default implementation for fields, provided in $VCS_HOME/etc/rvm/shared/lib/RTL:

File vmm_ral_host_itf.v:

```
interface vmm_ral_host_itf(input bit clk,
                           input bit rstn);
logic [`VMM_RAL_ADDR_WIDTH-1:0] adr;
logic [`VMM_RAL_ADDR_BYTES-1:0] sel;
logic [`VMM_RAL_DATA_WIDTH-1:0] rdat;
logic [`VMM_RAL_DATA_WIDTH-1:0] wdat;
logic                           wen;
logic                           ack;

modport master(input  clk, rstn,
               output adr, sel, wdat, wen,
               input  rdat, ack);
modport slave(input  clk, rstn,
              input  adr, sel, wdat, wen,
              output rdat, ack);
```

```
endinterface
```

File vmm_ral_rw_field_rtl.v:

```
module vmm_ral_rw_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen,
        fld_in, fld_wen);

parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;
input  [width-1:0] fld_in;
input              fld_wen;

reg [width-1:0] fld_out;

always @(posedge clk or negedge rstn)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else begin
      if (fld_wen) fld_out <= fld_in;
      else if (hst_sel && hst_wen) begin
         fld_out <= hst_wdat;
      end
   end
end

endmodule
```

File vmm_ral_ro_field_rtl.v:

```
module vmm_ral_ro_field_rtl(
        clk, rstn, fld_in, fld_out);

parameter width = 1;
```

```
input              clk, rstn;
input  [width-1:0] fld_in;
output [width-1:0] fld_out;

assign fld_out = fld_in;

endmodule
```

File vmm_ral_wo_field_rtl.v:

```
module vmm_ral_wo_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen);

parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;

reg [width-1:0] fld_out;

always @(posedge clk)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else if (hst_sel && hst_wen) begin
      fld_out <= hst_wdat;
   end
end

endmodule
```

File vmm_ral_a0_field_rtl.v:

```
module vmm_ral_a0_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen,
        fld_in, fld_wen);
```

```
parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;
input  [width-1:0] fld_in;
input              fld_wen;

reg [width-1:0] fld_out;

always @(posedge clk or negedge rstn)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else begin
      case ({hst_sel, hst_wen, fld_wen})
      3'b111: fld_out <= (fld_out | hst_wdat) & ~fld_in;
      3'b110: fld_out <= fld_out | hst_wdat;
      3'b001: fld_out <= fld_out & ~fld_in;
      endcase
   end
end

endmodule
```

File vmm_ral_a1_field_rtl.sv:

```
module vmm_ral_a1_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen,
        fld_in, fld_wen);

parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;
input  [width-1:0] fld_in;
input              fld_wen;

reg [width-1:0] fld_out;

always @(posedge clk or negedge rstn)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else begin
      case ({hst_sel, hst_wen, fld_wen})
      3’b111: fld_out <= (fld_out & hst_wdat) | fld_in;
      3’b110: fld_out <= fld_out & hst_wdat;
      3’b001: fld_out <= fld_out | fld_in;
      endcase
   end
end

endmodule
```

File vmm_ral_w1c_field_rtl.v:

```
module vmm_ral_w1c_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen,
        fld_in, fld_wen);

parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;
input  [width-1:0] fld_in;
input              fld_wen;

reg [width-1:0] fld_out;

always @(posedge clk or negedge rstn)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else begin
      case ({hst_sel, hst_wen, fld_wen})
      3’b111: fld_out <= (fld_out & ~hst_wdat) | fld_in;
      3’b110: fld_out <= fld_out & ~hst_wdat;

      3’b001: fld_out <= fld_out | fld_in;
      endcase
   end
end

endmodule
```

File vmm_ral_rc_field_rtl.v

```
module vmm_ral_rc_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen,
        fld_in, fld_wen);

parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;
input  [width-1:0] fld_in;
input              fld_wen;

reg [width-1:0] fld_out;

always @(posedge clk or negedge rstn)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else begin
      case ({hst_sel, fld_wen})
      2’b11: fld_out <= fld_in;
      2’b10: fld_out <= 0;
      2’b01: fld_out <= fld_out | fld_in;
      endcase
   end
end
```

```
endmodule
```

File vmm_ral_ru_field_rtl.v:

```
module vmm_ral_ru_field_rtl(
        clk, rstn, fld_out,
        hst_wdat, hst_sel, hst_wen,
        fld_in, fld_wen);

parameter width = 1;
parameter reset = 0;

input              clk, rstn;
output [width-1:0] fld_out;
input  [width-1:0] hst_wdat;
input              hst_sel;
input              hst_wen;
input  [width-1:0] fld_in;
input              fld_wen;

reg [width-1:0] fld_out;

always @(posedge clk or negedge rstn)
begin
   if (!rstn) begin
      fld_out <= reset;
   end
   else if (fld_wen) begin
      fld_out <= fld_in;
   end
end
endmodule
```

File vmm_ral_rw_notifier_rtl.v:

```
module vmm_ral_notifier_rtl(clk, rstn, sel, wen, rd, wr);
   input sel, wen;
   input clk, rstn;
   output rd, wr;

reg rd, wr;

always @(posedge clk or negedge rstn)
```

```
begin
   if (!rstn) begin
      rd <= 0;
      wr <= 0;
   end
   else begin
      rd <= sel & ~wen;
      wr <= sel & wen;
   end
end
```