HJ-micro Register Design Automation (HRDA)

Neo

2022/04/08

CONTENTS

REVISION HISTORY

Date	Revision	Description
2022-03-22	0.1	Add regmst for deadlock detection.

1
Introduction

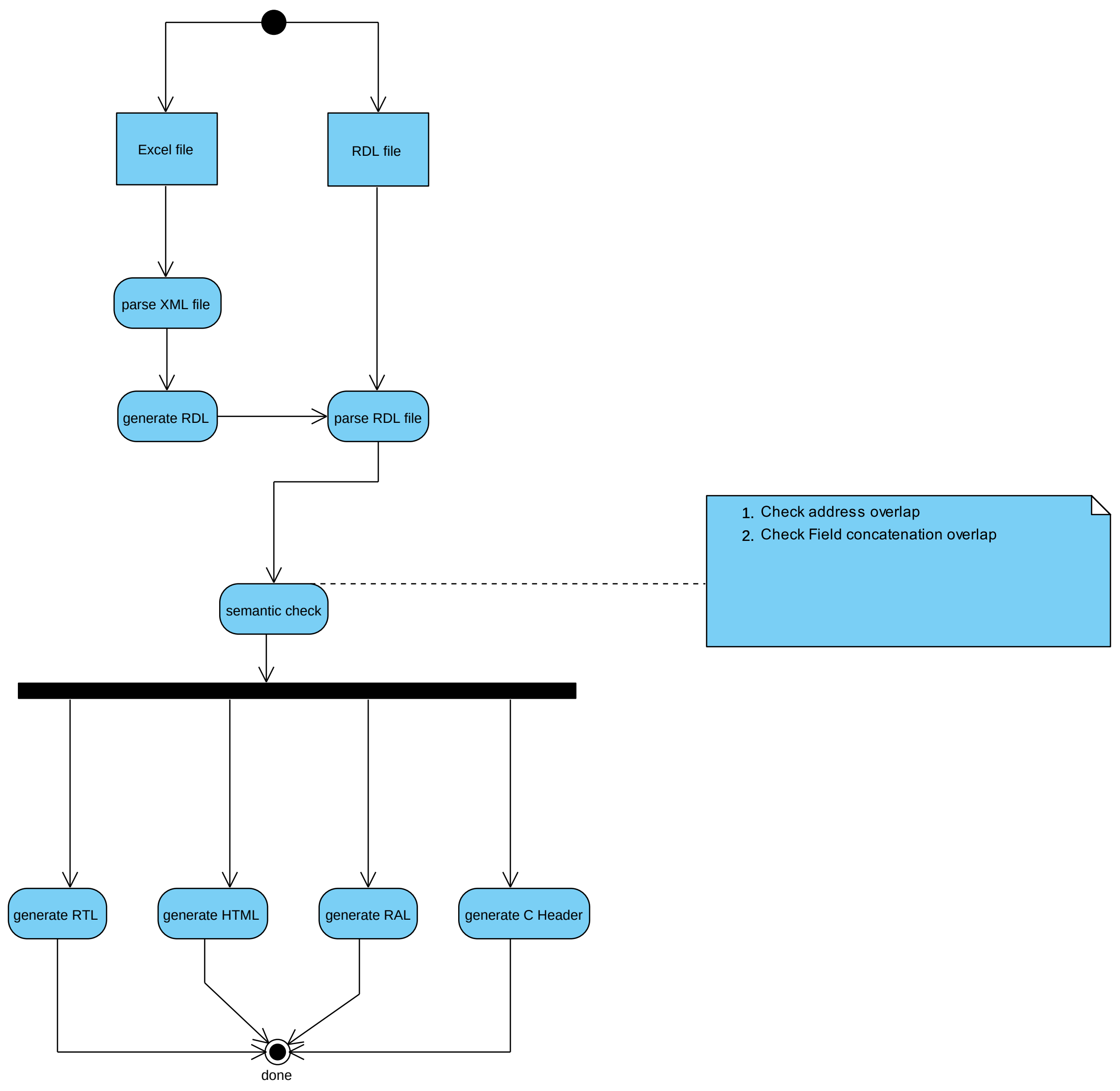


Figure 1.1: HJ-micro Regsiter Design Automation Flow

HJ-micro Register design Automation (HRDA) (as shown in Figure 1.1) is a methdology that automates register module description and code generation. The methdology has two input formats: *Excel spreadsheet* and *SystemRDL*. For modules with a few registers and simple address mapping, Excel spreadsheet is recommended. For complex module with fancy mappings, SystemRDL is more expressive and flexible.

In Excel spreadsheet based flow, HRDA scripts can generate an Excel spreadsheet file which contains predefined layout for register descriptions. Users fill the spreadsheet to finish register module design. HRDA scripts then parse the spreadsheet and generate a equivalent SystemRDL file for rest of the flow.

After parsing is done, semantic checking is performed on internal data structure, such as:

1. check address overlap among all registers
2. check fields concatenation overlap which a register word
3. check start address of each memory block is aligned

After semantic check is done, several output files are generated:

1. RTL modules for each generation boundary.
2. HTML document for all registers
3. RAL for all registers
4. C Header for all registers

2
RTL Architecture

2.1 Register Network

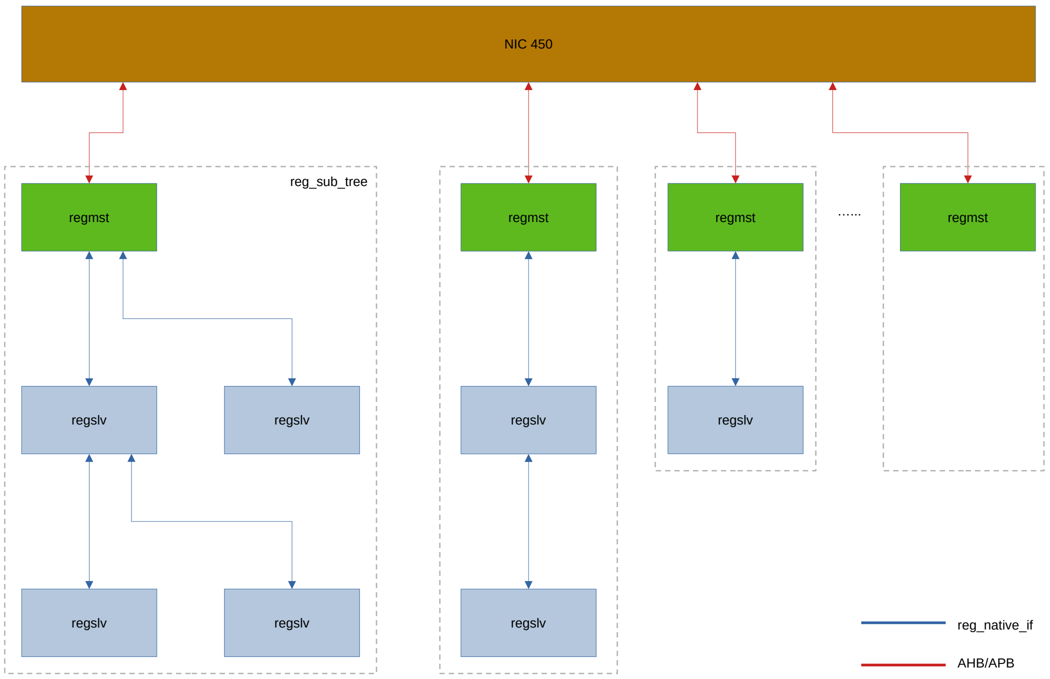


Figure 2.1: Register Network Architecture

Register network implmeneted by HRDA flow is a multi-root hierarchical network. A typical network is shown in Figure 2.1. Control/Status regsiters are distributed all around the chip modules. `regmst` (Register Access Master) is the root of a register hierarchy. It translates AHB or APB access to register native access interface (reg_native_if). A system can have multiple `regmst` to form a large network and support concurrent register access.

Each module that has registers will be connected into register network via reg_native_if. Typically, reg_natvie_if is implmented by follow components:

1. `regslv`: a module generated by HRDA scripts that contains all registers described in SystemRDL. `regslv` can be chained to form a hierarchy.
2. `regslv2mem`: a module genreated by HRDA scripts that bridges reg_natvie_if to memory modules for mapping memory into register space.

3. Other `regslv` bridge that bridge third party IP register access interface to register network. It is expected to be implmented by user, such as:
   - `regslv` to APB bridge
   - `regslv` to CR Parallel brdige for connecting PHY registers
   - etc.

2.2 Register Access Master (regmst)

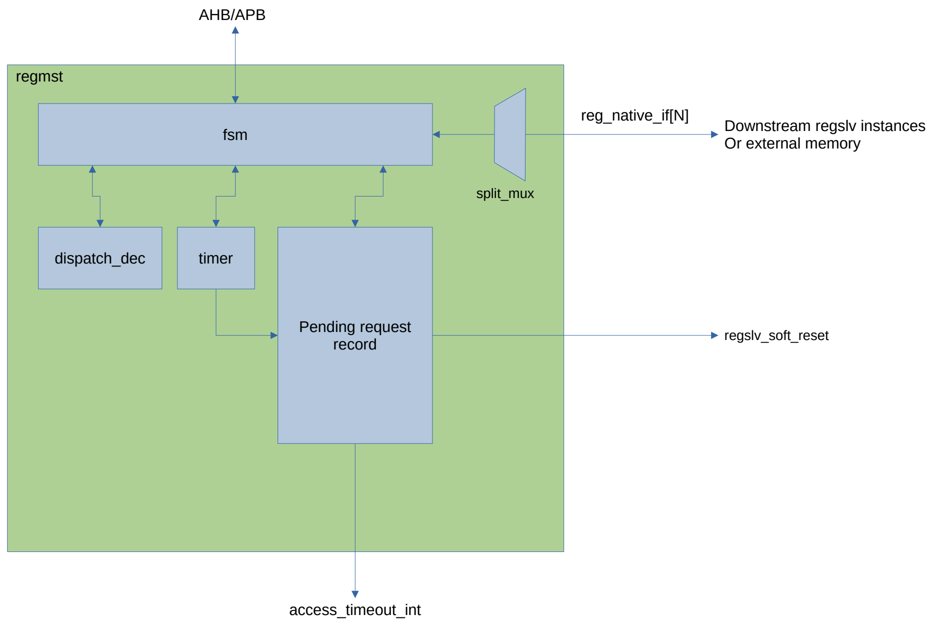


Figure 2.2: Register Network Architecture

Figure 2.2 shows architecture of `regmst`. `regmst` bridges SoC access (usually AHB or APB) to reg_native_if. `dispatch_dec` (dispatch decoder) decodes the target address and forwards the access to next stage `regslv`. After sending the access request, `regmst` starts a timer. If timeout is detected in waiting the response, `regmst` reponds to AHB/APB with a configurable fake data (such as 0xffff_ffff, or 0xdead_beef), and raises an interrupt to report the timeout event. The unresponded request information is logged in local register in `regmst`. Software can determine the problematic module by reading the error log register in `regmst`. And trigger a soft-reset within the hierarchy of `regmst`. `regmst` will assert soft-reset reset signal, which is broadcasted to all `regslv` (including `regslv2mem` and all `regslv` brdiges). The software reset will reset all FSM and bring back the hierarchy below `regmst` to functional.

`regmst` doesn't support outstanding request. So timeout detecting logic is quite straitforward (Figure 2.3) in FSM:

1. `regmst` decodes target address to determine the output interface
2. `regmst` starts forwarding access to next stage `regslv`, waits for response, and starts a 10ms timer.
   (a) If response comes back, `regmst` sends response back to SoC, reset timer, and transaction is completed.
   (b) If timeout occurs during waiting, `regmst` logs the transaction, finishes the transaction with fake data, and raise interrupt.

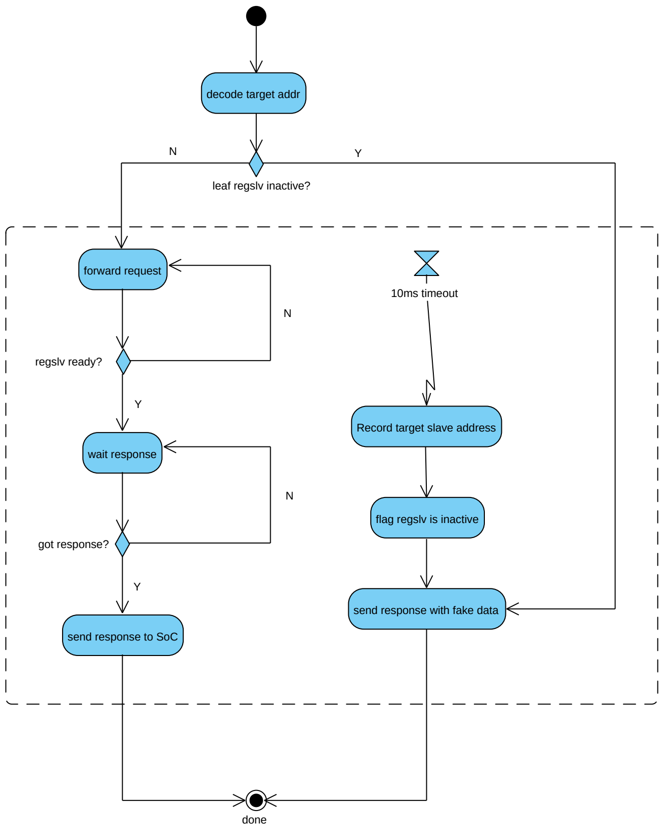


Figure 2.3: Register Mster Operation Flow

2.3 Register Access Slave (regslv)

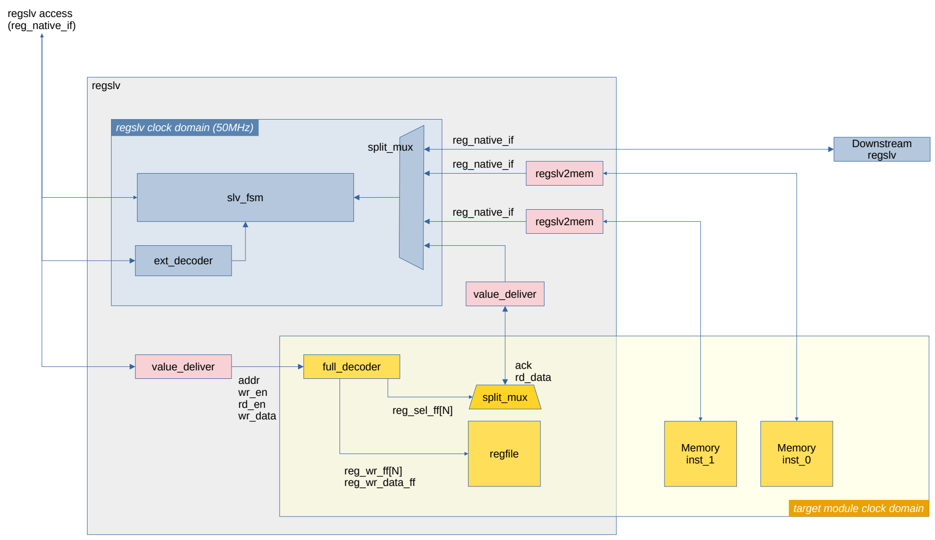


Figure 2.4: Register Slave RTL Module Architecture

The generated module is `regslv`. Its general architecture is shown in Figure 2.4. Each generation boundary (`addrmap` in RDL or a sheet in Excel) have a corresponding `regslv` module. RTL module name (and Verilog filename) is `regslv_XXX`, where `XXX` is RDL or Excel filename.

Each `regslv` can have its own registers. It can also forward access to its downstream `regslv`, memory instances, or third party IPs. For third party IPs, designers will have to implement a bridge to translate between different access protocol.

2.3.1 CDC Considerations

It is obvious that modules in SoC operate at different clocks. `regslv` network operates in a slow clock domain (clk_regs, e.g., 50MHz). Target module operate in its own clock domain. CDC are implemented as below:

1. Use `value_deliver` to transfer reg_native_if request to target module domain, before register address decoding.
2. Use `value_deliver` to transfer register read data from target module domain back to `regslv` domain.
3. Use `value_deliver` to transfer request and response in all `regslv` bridges, including ~regslv2mem and third party IP brdige.

2.3.2 reg_native_if

`regslv` uses reg_natvie_if for register access. It also uses reg_natvie_if to forward register access. All `regslv` bridges should implement reg_natvie_if. Signals of reg_natvie_if are listed as below:

```
parameter ADDR_WIDTH = 64;
parameter DATA_WIDTH = 32;

input                  req_vld;
input                  wr_en, rd_en;
input [ADDR_WIDTH-1:0] addr;
input [DATA_WIDTH-1:0] wr_data;
output                 req_rdy;

output                  ack_vld;
output [DATA_WIDTH-1:0] rd_data;
```

`req_vld` quarlifies other input signals, when both `req_vld` and `req_rdy` are asserted, register access is accepted by `regslv`. `ack_vld` quarlifies output signal. There is no backpressure in response path, so only `ack_vld` is enough.

Input `reg_native_if` is always populated. Number of forward `reg_native_if` is generate according to **external** component described in RDL.

2.3.3 slv_fsm

`slv_fsm` handles transactions at input reg_native_if. It also forwards transaction to output reg_native_if in case the access falls in range of external component.

2.3.4 full_decoder

`full_decoder` decodes address and create a `reg_sel[N-1:0]` vector. It is generated by automation tool as a SystemVerilog `unique case` statement, as demonstrated below:

```
unique case (addr)
  reg_sel = {NUM_OF_REG{1'b0}};
  dummy_reg = 1'b0;

`REG_ADDR_0, `REG_ADDR_1:
  reg_sel[0] = 1'b1;

  // other reg_sel[i] assignment
  // ...
  // ...

  default: dummy_reg = 1'b1;
endcase
```

2.3.5 split_mux

split_mux is a onehot-mux that has parameter to specify *group size*. When number of input candidcates exceed the group size, a two-level MUX network is constructed. DFF is inserted between first level and second level to improve timing.

2.4 Register and Field

field is the lowest-level structure component, it has software and hardware access type. Its structure is shown in Figure 2.5.

`field` is an existing RTL module, it implements various hardware and software access types which can be mapped to RDL descriptions.

```
module field (clk, rst_n,

              sync_rst,

              ext_async_reset_value,
              ext_sync_reset_value,

              write_protect_en,
              sw_wr, sw_rd, sw_wr_data,

              hw_pulse, hw_value,

              field_value);

   // Field width
   parameter F_WIDTH      = 4;

   // Number of synchronous reset.  sync_rst input is always
   // populated, so if SRST_CNT=0, sync_rst port width is the minimal
```

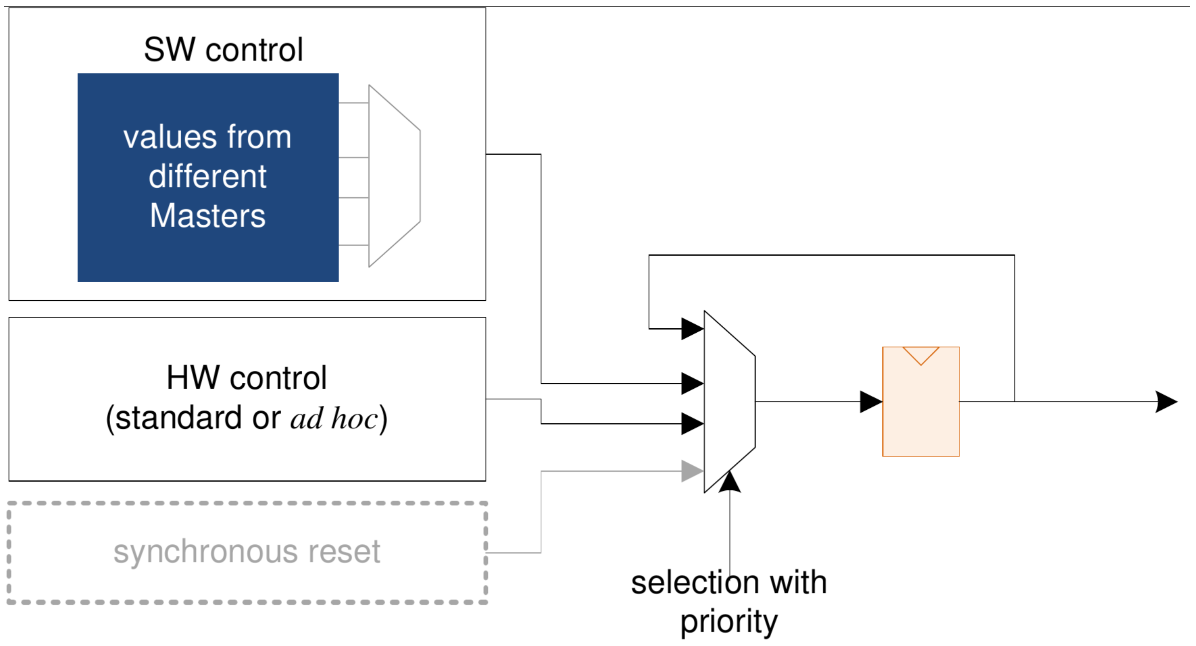

SW control
values from different Masters
HW control (standard or *ad hoc*)
synchronous reset
selection with priority

Figure 2.5: Field RTL structure

```
// value 1 for a legitimate verilog module, and the port is not
// used.  If SRST_CNT>0, sync_rst port width is declared
// accordingly for connection.
parameter SRST_CNT      = 0;
parameter SRST_WIDTH    = SRST_CNT ? SRST_CNT : 1;

// Asynchronous/Synchronous reset values.  By default, they are
// both zero.
parameter [F_WIDTH-1:0] ARST_VALUE = {F_WIDTH{1'b0}};
parameter [F_WIDTH-1:0] SRST_VALUE  = ARST_VALUE;

// Flags to use external reset value.  If asserted,
// asynchronous/synchronous reset values comes from input ports.
parameter USE_EXT_ASYNC_VALUE = 0;
parameter USE_EXT_SYNC_VALUE  = 0;

// Software and hardware access type.
parameter SW_TYPE       = `SW_RW;
parameter HW_TYPE       = `HW_RO;

// Field value update priority, by default software dominates the
// update.
parameter WR_PRIORITY   = `SW;


// Clock and asynchronous active-low reset
input                       clk, rst_n;

// Synchronous active-high reset
input [SRST_WIDTH-1:0]      sync_rst;

// External asynchronous/synchronous reset value.  If used, they
// must be stable before assertion of field local reset.
input [F_WIDTH-1:0]         ext_async_reset_value,
                            ext_sync_reset_value;

// Software access interface
input [F_WIDTH-1:0]         sw_wr_data;
input                       sw_rd, sw_wr;

// Write protection enable, which is checked on each software
// access.  When asserted, software can't modify field value.  In
```

```
    // other words, any SW_TYPE is tempoarily changed SW_RO.
    input                        write_protect_en;

    // Hardware access interface.  Interpretation of this interface is
    // different for different HW_TYPE.
    input [F_WIDTH-1:0]          hw_value;
    input                        hw_pulse;

    // Field value output.  Software read field value from it, any
    // software/hardware modification occurs at next cycle.
    output [F_WIDTH-1:0]         field_value;

endmodule
```

All supported access types are listed in `rdlreg.vh`

```
// Sowftware types
`define SW_RO          5'h0 // Read only
`define SW_RW          5'h1 // Read write
`define SW_RW1C        5'h2 // Read only, write 1 to clear.
`define SW_RW1S        5'h3 // Read only, write 1 to set.
`define SW_ROC         5'h4 // Read clear
`define SW_W1CO        5'h5 // Read return 0, write 1 to clear, same as SW_RW1C
`define SW_W1SO        5'h6 // Read return 0, write 1 to set, same as SW_RW1S


// Hardware types
// simple access type, build-in support
`define HW_RO          5'h0  // Read-Only
`define HW_WIRED       5'h1  // Hard wired.  register value is
                             // refreshed in realtime by hardware.
                             // hw_pulse is ignored, hw_value is always
                             // loaded to field.
`define HW_SET         5'h2  // Bitwise set, hw_pulse input is ignored.
`define HW_CLR         5'h3  // Bitwise clear, hw_pulse input is ignored.
`define HW_ANDED       5'h4  // Bitwise AND, hw_pulse input is ignored.
`define HW_ORED        5'h5  // Bitwise OR, hw_pulse input is ignored.
`define HW_XORED       5'h6  // Bitwise XOR, hw_pulse input is ignored.
`define HW_VALUE       5'h7  // Set hw_value into field when hw_pulse
                             // is asserted, usually a pulse
`define HW_SELF_CLR    5'h8  // The asserted bit in field will be
                             // cleared at next cycle, hw_pulse input
                             // is ignored.

// complex access type, supported via other module
`define HW_CNT             5'h9  // Counter mode
`define HW_INT_EN_STATUS   5'hC  // Interrupt


`define SW 0
`define HW 1

`define XREG_DEBUG 1
```

Field is concatenated to form register and further mapped into address space for software's access, as shown in Figure 2.6.

2.5 External Components

When `regslv` detects access is for external components, it forwards it via output `reg_native_if`. There several types of external components:

1. Memory: memory instances in design can be mapped into register spaces.
2. Third party IP: registers in third party IP should be mapped into SoC.
3. Other `regslv`: downstream modules or `regslv` bridges can be chained to form a large register network.

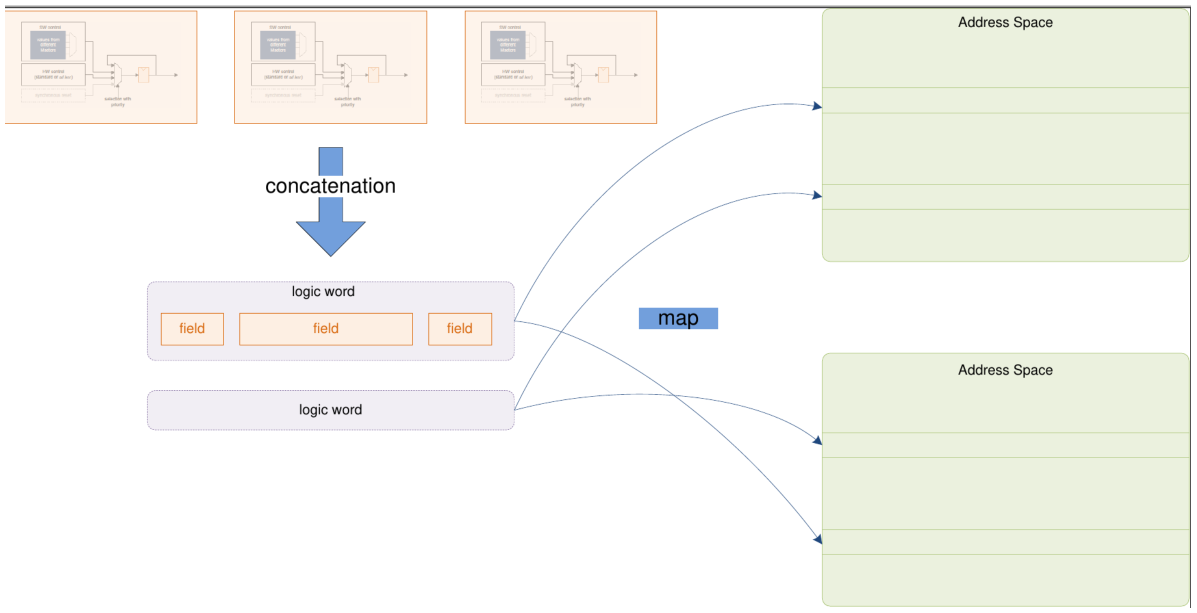


Figure 2.6: field Concatenation and mapping

3
SystemRDL Coding Guideline

SystemRDL ([1]) is a flexible language for describing register modules. This section provides coding guidelines for simplifying register module description.

These coding guidelines specify a subset of SystemRDL syntax and property to use. The generation scripts **only** interpret this subset. Using of other description styles not mentioned in this document is not supported by the generation scripts.

3.1 General Rules

3.1.1 Structural Components

SystemRDL has following structural components that map to RTL design in RTL Architecture:

1. `field` for describing Field
2. `reg` for describing Register
3. `regfile` for packing registers together with address allocation
4. `addrmap` similar to `regfile` on register packing and address allocation. Additionally, it defines a RTL code generation boundary. Each definition of `addrmap` with `gencode` property set will be generated to an `regslv` module.
5. `signal` for declaring signals used in components defintions.

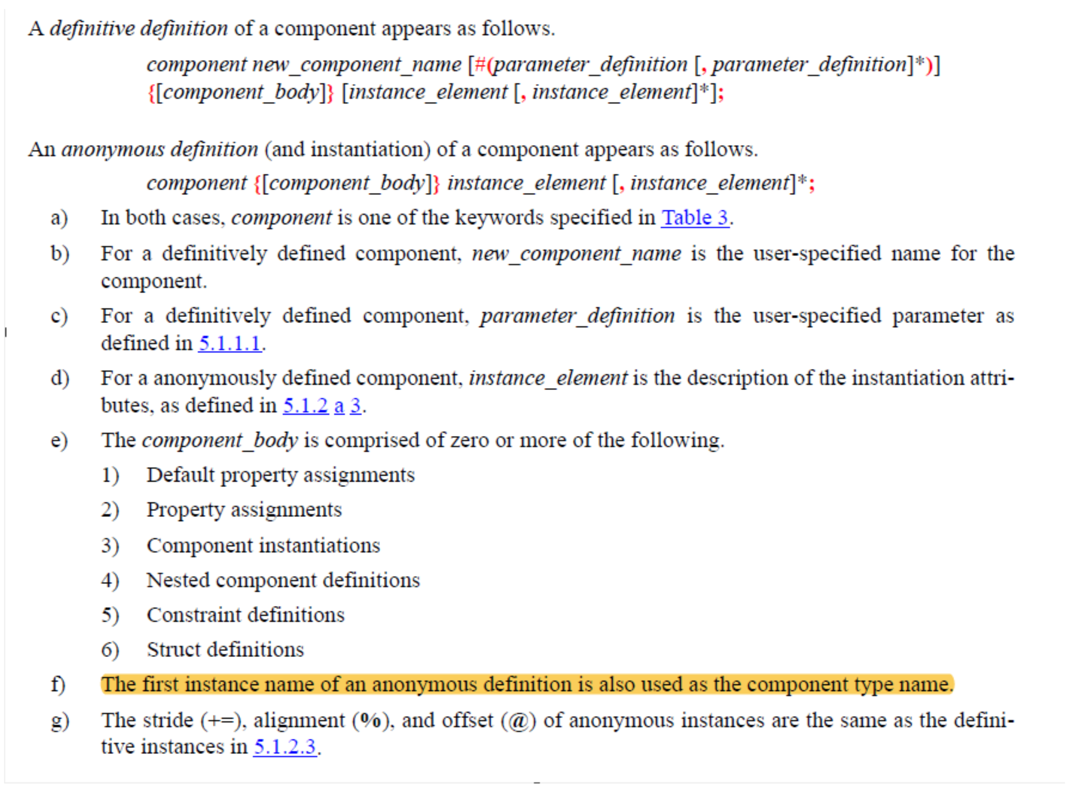
Figure 3.1 shows the syntax for defining components. Components can be defined in any order, as long as each component is defined before it is instantiated. All structural components (and signals) need to be instantiated before being generated.

Here is an example for register definition, where register `myReg` is a *definitive definition*, and field `data` is an *anonymous definition*.

```
reg myReg #(longint unsigned SIZE = 32, boolean SHARED = true) {
  regwidth = SIZE;
  shared = SHARED;
  field {} data[SIZE - 1];
  };
```

Component definitions can have parameters. Parameter can be overridden when component is being instantiated ([1] section 5.1.1.1). Here is an example:

```
addrmap myAmap {
    myReg reg32;
    myReg reg32_arr[8];
    myReg #(.SIZE(16)) reg16;
    myReg #(.SIZE(8), .SHARED(false)) reg8;
};
```



A *definitive definition* of a component appears as follows.

component new_component_name [#(*parameter_definition* [, *parameter_definition*]*)]
{[*component_body*]} [*instance_element* [, *instance_element*]*];

An *anonymous definition* (and instantiation) of a component appears as follows.

component {[*component_body*]} *instance_element* [, *instance_element*]*;

a) In both cases, *component* is one of the keywords specified in Table 3.

b) For a definitively defined component, *new_component_name* is the user-specified name for the component.

c) For a definitively defined component, *parameter_definition* is the user-specified parameter as defined in 5.1.1.1.

d) For a anonymously defined component, *instance_element* is the description of the instantiation attributes, as defined in 5.1.2 a 3.

e) The *component_body* is comprised of zero or more of the following.

  1) Default property assignments
  2) Property assignments
  3) Component instantiations
  4) Nested component definitions
  5) Constraint definitions
  6) Struct definitions

f) The first instance name of an anonymous definition is also used as the component type name.

g) The stride (+=), alignment (%), and offset (@) of anonymous instances are the same as the definitive instances in 5.1.2.3.

Figure 3.1: Syntax for defining Component

3.1.2 Property

Structural components have various **property** to define their behaviors. Each property is associated with at least on data type (such integer, boolean, string, etc). In addition to build-in property defined in SystemRDL, user can add **User-defined** properties.

Property can be assigned at definition time:

```
field {} outer_field ;
reg {
  default name = "default name";
  field {} f1; // assumes the name "default name" from above
  field { name = "new name";} f2; // name assignment overrides "default name"
  outer_field f3 ; // name is undefined, since outer_field is not defined in the
                   // scope of the default name
} some_reg;
```

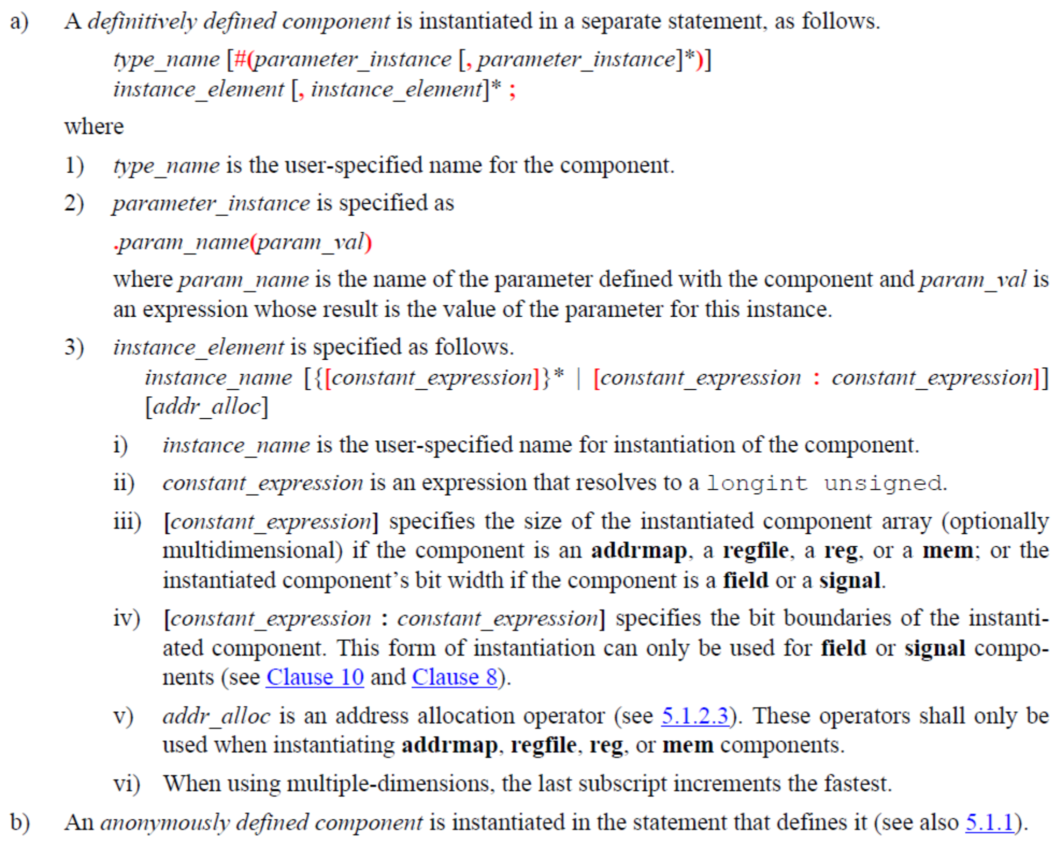
Property can also be dynamically assigned:

```
reg {
  field {} f1;
  f1->name = "New name for Field 1";
  } some_reg[8];

some_reg->name = "This value is applied to all elements in the array";
some_reg[3]->name = "Only applied to the 4th item in the array of 8";
```

3.1.3 Instantiating Components

Figure 3.2 is extracted from [1] section 5.1.2, which describes the syntax for instantiating component. Here are some examples:



a) A *definitively defined component* is instantiated in a separate statement, as follows.

   type_name [#(*parameter_instance* [, *parameter_instance*]*)]
   instance_element [, *instance_element*]* ;

   where

   1) *type_name* is the user-specified name for the component.
   2) *parameter_instance* is specified as

      .*param_name*(*param_val*)

      where *param_name* is the name of the parameter defined with the component and *param_val* is an expression whose result is the value of the parameter for this instance.
   3) *instance_element* is specified as follows.

      instance_name [{[*constant_expression*]}* | [*constant_expression* : *constant_expression*]] [*addr_alloc*]

      i) *instance_name* is the user-specified name for instantiation of the component.
      ii) *constant_expression* is an expression that resolves to a `longint unsigned`.
      iii) [*constant_expression*] specifies the size of the instantiated component array (optionally multidimensional) if the component is an **addrmap**, a **regfile**, a **reg**, or a **mem**; or the instantiated component's bit width if the component is a **field** or a **signal**.
      iv) [*constant_expression* : *constant_expression*] specifies the bit boundaries of the instantiated component. This form of instantiation can only be used for **field** or **signal** components (see Clause 10 and Clause 8).
      v) *addr_alloc* is an address allocation operator (see 5.1.2.3). These operators shall only be used when instantiating **addrmap**, **regfile**, **reg**, or **mem** components.
      vi) When using multiple-dimensions, the last subscript increments the fastest.

b) An *anonymously defined component* is instantiated in the statement that defines it (see also 5.1.1).

Figure 3.2: Syntax for Instantiating Components

```
// The following code fragment shows a simple scalar field component instantiation.
field {} myField; // single bit field instance named  “myField”

// The following code fragment shows a simple array field component instantiation.
field {} myField[8]; // 8 bit field instance named  “myField”
```

3.1.4 Instance address allocation: alignment, addressing mode, addr_alloc

The offset of an instance within an object is always relative to its parent object. If an instance is not explicitly assigned an address allocation operator (see Address Allocation Operator), the compiler assigns the address according to the **alignment** and *addressing mode*. The address of an instance from the top level `addrmap` is calculated by adding the instance offset and the offset of all its parent objects.

3.1.4.1 alignment

The **alignment** property defines the byte value of which the container's instance addresses shall be a multiple. This property can be set for `addrmaps` and `regfiles` , and its value shall be a power of two (2^N). Its value is inherited by all of the container's non-addrmap children. By default, instantiated objects shall be aligned to a multiple of their width (e.g., the address of a 64-bit register is aligned to the next 8-byte boundary).

3.1.4.2 Addressing Mode

addressing property can only be used in `addrmap` component. There are three addressing modes: `compact`, `realign` (the default), and `fullalign`.

`compact` Specifies the components are packed tightly together while still being aligned to the `accesswidth` parameter:

```
addrmap some_map {
    default accesswidth=32;
    addressing=compact;
    reg { field {} a; } a; // Address 0x0 - 0x3: 4 bytes
    reg { regwidth=64; field {} a; } b; // Address 0x4 - 0x7: lower 32-bit,
                                        // Address 0x8 - 0xB: higher 32-bit
                                        // starting address 0x4 tightly follows previous
                                        // reg "a"
    reg { field {} a; } c[20]; // Address 0xC  - 0xF:  Element 0
                               // Address 0x10 - 0x13: Element 1
                               // Address 0x14 - 0x17: Element 2
};


addrmap some_map {
    default accesswidth=64;
    addressing=compact;
    reg { field {} a; } a; // Address 0x0 - 0x3: 4 bytes
    reg { regwidth=64; field {} a; } b; // Address 0x8 - 0xB:
    reg { field {} a; } c[20]; // Address 0x10 - Element 0
                               // Address 0x14 - Element 1
                               // Address 0x18 - Element 2
                               // starting address is 0x10, align to 64-bit, 4 bytes in 0xC-0xF is skipped
};
```

`regalign` Specifies the components are packed so each component's start address is a multiple of its size (in bytes). Array elements are aligned according to the individual element's size (this results in no gaps between the array elements). This generally results in simpler address decode logic.

```
addrmap some_map {
  default accesswidth = 32;
  addressing = regalign;
  reg { field {} a; } a; // Address 0x0
  reg { regwidth=64; field {} a; } b; // Address 0x8-0xF, align to 64-bit
  reg { field {} a; } c[20]; // Address 0x10
```

```
                              // Address 0x14 - Element 1
                              // Address 0x18 - Element 2
};
```

`fullalign` The assigning of addresses is similar `regalign`, except for arrays. The alignment value for the first element in an array is the size in bytes of the **whole array** (i.e., the size of an array element multiplied by the number of elements), **rounded up to nearest power of two**. The second and subsequent elements are aligned according to their individual size (so there are no gaps between the array elements).

```
addrmap some_map {
  default accesswidth = 32;
  addressing = fullalign;
  reg { field {} a; } a; // Address 0
  reg { regwidth=64; field {} a; } b; // Address 8
  reg { field {} a; } c[20]; // Address 0x80 - Element 0
                             // Address 0x84 - Element 1
                             // Address 0x88 - Element 2
                             // starting address align to 4*20=80Byte,
};
```

3.1.4.3 Address Allocation Operator

When instantiating `reg`, `regfile`, `mem`, or `addrmap`, the address may be assigned using one of following address allocation operators

1. `@`: It specifies the address for the instance.

```
addrmap top {
  regfile example{
    reg some_reg {
      field {} a;
    };

    some_reg a @0x0;
    some_reg b @0x4;

    // Implies address of 8
    // Address 0xC is not implemented or specified
    some_reg c;

    some_reg d @0x10;
    };
  };
```

2. `+=`: It specifies the address stride when instantiaing an array of components (controls the spacing of the components). The address stride is relative to the previous instane's address. It is only used for arrayed `addrmap`, `regfile`, `reg`, or `mem`.

```
addrmap top {
  regfile example {
    reg some_reg { field {} a; };

    some_reg a[10]; // So these will consume 40 bytes
                    // Address 0,4,8,C....

    some_reg b[10] @0x100 += 0x10; // These consume 160-12 bytes of space
                                   // Address 0x100 to 0x103, 0x110 to 0x113,....
    };
  };
```

3. `%=`: It specifies the aligment of address when instantiaing a component (controls the aligment of the components). The initial address alignment is relative to the previous instance's address. The `@` and `%=` operators are mutually exclusive per instance.

```
addrmap top {
  regfile example {
    reg some_reg { field {} a; };
```

```
            some_reg a[10]; // So these will consume 40 bytes
                            // Address 0,4,8,C....

            some_reg b[10] @0x100 += 0x10; // These consume 160-12 bytes of space
                                           // Address 0x100 to 0x103, 0x110 to 0x113,....

            some_reg c %=0x80; // This means ((address % 0x80) == 0))
                               // So this would imply an address of 0x200 since
                               // that is the first address satisfying address>=0x194
                               // and ((address % 0x80) == 0)
          };
        };
```

3.2 Signal

3.3 Field Description

3.3.1 Naming Convention

Each RDL **field** instance will be generated to an Verilog `field` module instance. In generated RTL, stem name of field is `<reg_inst_name>__<field_inst_name>`. Other signals belong to the field are named by prefixing/-suffixing elements. e.g., Register instance name is `ring_cfg`, Field instance name is `rd_ptr`:

1. `field` instance name is `x__<stem>` (prefixed with `x__`): `x__ring_cfg__rd_ptr`
2. output port name for current Field value is `<stem>__curr_value`: `ring_cfg__rd_ptr__curr_value`
3. input port for update its value from hardware is `<stem>__next_value`: `ring_cfg__rd_ptr__next_value`
4. input port for quarlifying update is `<stem>__pulse`: `ring_cfg__rd_ptr__pulse`

3.3.2 Description Guideline

SystemRDL defines several properties for describing Field, however, only a subset of them are interpreted by the scripts. Only properties documented in this section are allowed for Field description, others are prohibited to use.

Table 3.1: Field Properties Supported in scripts

Property	Notes	Type	Default	Dynamic
fieldwidth	Width of Field.	*longint unsighed*	1	No
reset	Reset value of Field.	*bit*	0	Yes
resetsignal	Reference to signal used as **Asynchornous reset** of the Field.	*reference*		Yes
hj_syncresetsignal	Reference to signal used as **Synchronous Reset** of the Field.	*reference*		Yes
name	Specifies a more descriptive name (for documentation purposes).	*string*	""	Yes
desc	Describes the component's purpose. MarkDown syntax is allowed	*string*	""	Yes
sw	Software access type, one of `rw`, `r`, `w`, `rw1`, `w1`, or `na`.	*access type*	`rw`	Yes
onread	Software read side effect, one of `rclr`, `rset`, or `na`.	*onreadtype*	`na`	Yes
onwrite	Software write side effect, one of `woset`, `woclr`, `wot`, `wzs`, `wzc`, `wzt`, or `na`.	*onwritetype*	`na`	Yes
swmod	Populate an output signal which is asserted when field is modified by software (written or read with a set or clear side effect).	*boolean*	false	Yes
swacc	Populate an output signal which is asserted when field is read.	*boolean*	false	Yes
singlepulse	Populate an output signal which is asserted for one cycle when field is written 1.	*boolean*	false	Yes
hw	Hardware access type, one of `rw`, or `r`	*access type*	`r`	No
hwclr	Hardware clear. Field is cleared upon assertion on hardware signal in bitwise mode.	*boolean*	false	Yes
hwset	Hardware set. Field is set upon assertion on hardware signal in bitwise mode.	*boolean*	false	Yes
precedence	One of `hw` or `sw`, controls whether precedence is granted to hardware (hw) or software (sw) when contention occurs.	*precedencetype*	`sw`	Yes

resetsignal specifies signal used as **Asynchronous reset** for the Field. By default, `rst_n` is used as asynchronous reset signal. When set to a reference of signal, an input port is populated for the signal and the field's asynchronous reset will be connected to the signal.

hj_syncresetsignal is a *User-defined* property that specifies signal (or multiple signals) used as **Synchronous Reset** for the Field. By default, a Field doesn't have Synchronous reset. User can set **hj_syncresetsignal** property more than once to specify multiple synchronous reset signals. Each synchronous reset signal **must** be active high and one clock cycle wide. Reset value of synchronous reset is the same as that of asynchronous reset.

When **singlepulse** is `true`, **onwrite** property is ignored.

Current value of Field (`<stem>__curr_value`) is always output to user logic. If **hw** is `rw`, two more inputs are populated (`<stem>__next_value` and `<stem>__pulse`) for updating field value from user logic. If value from hardware is expected to be continously updated into Field, user should tie `<stem>__pulse` to `1'b1`. If either **hwclr** or **hwset** is `true` (they are mutually exclusive), `field` module use `<stem>__next_value` in bitwide mode and ignores `<stem>__pulse`. Each pulse in `<stem>__next_value` will clear or set corresponding bit on Field.

3.3.3 Examples

```
field {sw=rw; hw=r;} f1[15:0] = 1234;

field f2_t {sw=rw; hw=r;};

f2_t f2[16:16] = 0;
f2_t f3[17:17] = 0;

field {
    sw=rw; hw=r;
    hdl_path_slice = '{"f4"};
    hdl_path_gate_slice = '{"f4_31_gate", "f4_30_gate"};
} f4[31:30] = 0;
field {
    sw=rw; hw=r;
    hdl_path_slice = '{"f5_29", "f5_28"};
    hdl_path_gate_slice = '{"f5_gate"};
} f5[29:28] = 0;
```

3.4 Regsiter Description

3.4.1 Naming Convention

Each Register is a concatenation of Fields. No RTL module is implemented for Register. Instead, an `always_comb` block is used to concatenate Fields `curr_value` as below:

```
// ring_cfg
always_comb begin
   ring_cfg[31:0] = 32'd0;
   ring_cfg[31] = ring_cfg__ring_en__curr_value;
   ring_cfg[7:4] = ring_cfg__ring_size__curr_value[3:0];
end
```

All Fields in a Register share same register `rd_en`, `wr_en`, and `wr_data`. Scripts will connect the correct signal from address decoder to Field instances.

3.4.2 Description Guideline

Register definitions are all considered to be **internal**. **external** is only applied on `regfile` instances.

alias property ([1] section 10.5) is supported on regsiter instances within regfile.

An *alias register* is a register that appears in multiple locations of the same address map. It is physically implemented as a single register such that a modification of the register at one address location appears at all the locations within the address map. The accessibility of this register may be different in each location of the address block.

Alias registers are allocated addresses like physical registers and are decoded like physical registers, but they perform these operations on a previously instantiated register (called the primary register). Since alias registers are not physical, hardware access and other hardware operation properties are not used. Software access properties for the alias register can be different from the primary register. For example,

```
reg some_intr_r { field { level intr; hw=w; sw=r; woclr; } some_event; };
addrmap foo {
  some_intr event1;

  // Create an alias for the DV team to use and modify its properties
  // so that DV can force interrupt events and allow more rigorous structural
  // testing of the interrupt.
  alias event1 some_intr event1_for_dv;
  event1_for_dv.some_event->woclr = false;
  event1_for_dv.some_event->woset = true;
};
```

shared propery, on the other hand, allows same physical register to be mapped in several different address space.

Table 3.2: Register Properties Supported in scripts

Property	Notes	Type	Default	Dynamic
regwidth	Width of Register.	*longint unsighed*	32	No
shared	Defines a register as being shared in different address maps.	*boolean*	false	No

3.4.3 Examples

3.5 regfile

3.5.1 Description Guideline

A `regfile` is as a logical grouping of one or more registers and `regfile` instances. It packs registers together and provides address allocation support, which is useful for introducing an address gap between registers. The only difference between the `regfile` and the address map (`addrmap`) is an `addrmap` defines an RTL implementation boundary where the `regfile` does not. Since `addrmaps` define a implementation block boundary, there are some specific properties that are only specified for address maps and not specified for `regfiles`.

When `regfile` is instantiated within another `regfile`, scripts consider inner `regfile` instances are flattened and concatenated to form a larger `regfile`. So "`regfile` nesting" is just a technique to organize register descriptions. No **internal** or **external** is considered.

Standard SystemRDL allows **external** to be applied on `regfile` instances, but HRDA scripts ignores **external** modifier on `regfile` instance. `regfile` instance is always considered as packer of registers. **external** only applies on `addrmap` instances.

Table 3.3: regfile Properties Supported in scripts

Property	Notes	Type	Default	Dynamic
alignment	Specifies alignment of all instantiated components in the associated register file.	*longint unsighed*		No

3.5.2 Example

```
regfile myregfile #(.A (32)) {
  alignment = 32;
  reg {} xx;
}
```

3.6 Memory

3.6.1 Descriptions Guideline

Memory instances in `addrmap` are always **external**. When mapping memory into register space, the generated `reg_slv` module forwards access that falls in memory address region to memory access interface. Each mapped memory has a dedicated access data path.

Memory definition accepts properties listed in Table 3.4.

Table 3.4: Memory Properties Supported in scripts

Property	Notes	Type	Default	Dynamic
mementries	The number of memory entries, a.k.a memory depth.	*longint unsighed*		No
memwidth	The memory entry bit width, a.k.a memory width.	*logical unsighed*		No
sw	Programmer's ability to read/write a memory.	*access type*	`rw`	Yes

If **memwidth** is larger than **accesswidth**, each memory entry occupies *N* address slots, where *N* should be power of 2 (2^i) to simplify decode logic. Generated module will implement a snapshot register to atomically read/write memory entry.

3.6.2 Example

3.7 Address Map

3.7.1 Description Guideline

An address map component (`addrmap`) contains registers, register files, memories, and/or other address maps and assigns address to each instance of component. `addrmap` defines the boundaries of an RTL implementation. Each component might have already assigned address offset to its contents, `addrmap` further adds base address to them. After the outter most `addrmap` finishes assigning base address, absolute address allocation is settled.

HRDA scripts processes each `addrmap` definition as below:

1. `memory` instances are always considered **external**. There will be dedicated `reg_native_if` populated for each memory instance.
2. `reg`, `regfile` are generated according to the definition. Their contents address are allocated by the enclosing `addrmap`.
3. `addrmap` instances are handled in different ways depending on value of `hj_genrtl`, `hj_flatten_content` properties in `addrmap` definition, according to Table 3.5

Table 3.5: addrmap instance handling

hj_gentrl	hj_flatten_addrmap	handling behavior	Usage
false	false	Populate a dedicated `reg_native_if` for the `addrmap` instance. No `regslv` RTL module is generated for the `addrmap` definition.	3rd party IP register description
false	true	All contents in the `addrmap` is flattened in current scope, just like `regfile` does. No `regslv` RTL module is generated for `addrmap`.	Use `shared` property to map same register into different address spaces
true	*don't care*	Populate a dedicated `reg_natvie_if` for the `addrmap` instances. And generate `regslv` RTL module for the `addrmap`.	Hierarchical `regslv` chaining

All HRDA suppored properties for `addrmap` is listed in Table 3.6.

Table 3.6: Address Map Properties Supported in HRDA scripts

Property	Notes	Type	Default	Dynamic
alignment	Specifies alignment of all instantiated components in the address map.	*longint unsighed*		No
addressing	Controls how addresses are computed in an address map.	*addressingtype*		No
rsvdset	The read value of all fields not explicitly defined is set to 1 if rsvdset is `true`; otherwise, it is set to 0.	*boolean*	true	No

3.7.2 Example

```
addrmap some_bridge { // Define a Bridge Device
  desc="overlapping address maps with both shared register space and orthogonal register space";
  reg status {// Define at least 1 register for the bridge
    // Shared property tells compiler this register
    // will be shared by multiple addrmaps
    shared;

    field {
      hw=rw;
      sw=r;
      } stat1 = 1'b0;
    };


  reg some_axi_reg {
    field {
      desc="credits on the AXI interface";
      } credits[4] = 4'h7;   // End of field: {}

    };  // End of Reg: some_axi_reg


  reg some_ahb_reg {
    field {
      desc="credits on the AHB Interface";
      } credits[8] = 8'b00000011 ;
    };

  addrmap {
    littleendian;

    some_ahb_reg ahb_credits; // Implies addr = 0
    status ahb_stat @0x20;    // explicitly at address=20
    ahb_stat.stat1->desc = "bar"; // Overload the registers property in this instance
    } ahb;

  addrmap { // Define the Map for the AXI Side of the bridge
    bigendian; // This map is big endian
    some_axi_reg axi_credits;   // Implies addr = 0
    status axi_stat @0x40;      // explicitly at address=40
    axi_stat.stat1->desc = "foo"; // Overload the registers property in this instance
    } axi;
}; // Ends addrmap bridge
```

3.8 User-defined Property

3.8.1 hj_syncresetsignal

Assigning `signal` instance to this property will populate an extra input to the component. That input will be used as synchornous reset signal. If this property is assigned in a `field` definition, the input is added for the `field` instance. If this property is assigned in a `reg`, `regfile`, or `addrmap`, all containing field instances will have the input as synchornous reset.

```
property hj_syncresetsignal {
  component = field|reg|regfile|addrmap;
  type = string;
}
```

3.8.2 hj_genrtl

Refer to Table 3.5 for detail.

```
property hj_genrtl {
  component = addrmap;
  type = boolean;
  default = true;
}
```

3.8.3 hj_flatten_addrmap

Refer to Table 3.5 for detail.

```
property hj_flatten_addrmap {
  component = addrmap;
  type = boolean;
  default = false;
}
```

4

Excel Format

5

Tool Usage

6 Bibliography

[1] Accellera. *SystemRDL 2.0 Register Description Language*. January 2018.