GreenReg User's Guide (GreenReg v.4.0.x)

Developed by
Christian Schröder
Technical University of Braunschweig, Dept. E.I.S.

Based on Intel DRF

26th January 2010

Contents

Chapter 1

Introduction

The GreenReg project[1] provides a library that can be used for device and register modeling in ESL design. The project is based on the Intel DRF framework.

GreenReg can be understood as two parts: One side is the device and register modeling in the user model which is done by using the Device Register Framework (DRF) API, the other is the configuration mechanism which is provided by GreenConfig[2] and gives configuration tool access to the DRF objects.

This allows ESL modelers to use the powerful configuration abilities of GreenConfig to configure the DRF objects.

The GreenReg framework is not only a register framework. The way the user is able to model hardware registers and use their notifications for modeling behavior is a small model of computation.

1.1 Further Reading

This User's Guide focuses on the configuration aspect of GreenReg and some enhancements to DRF.

See the GreenReg Tutorial Slides[3] for more basic usage information and an introduction to the device and register modeling part.

[1] GreenReg project page: http://www.greensocs.com/projects/GreenReg
[2] GreenConfig project page: http://www.greensocs.com/projects/GreenControl/GreenConfig
[3] GreenReg Tutorial Slides: http://www.greensocs.com/Projects/GreenReg/docs/GreenRegTutorial

Chapter 2

GreenReg

This chapter guides through the enhancements of GREENREG made to DRF.

Section 2.1 clarifies some conventions used in this document and the code. Section 2.2 shows some background information whereas sections 2.3 and 2.4 are important sections telling how to use the GREENREG enhancements. Section 2.6 is for developers enhancing GREENREG itself. The Notes section 2.7 notes general GREENREG facts and small features, section 2.8 points to the code.

2.1 Namespace and naming conventions

The DRF classes of GREENREG are located within the namespace gs::reg and gs::reg_utils. Some configuration specific classes are in the namespace gs.

For the correct namespace of the classes used in this document please refer to the doxygen generated API reference.

The GREENREG classes use some abbreviations (prefixes):

- *I_* stands for *Interface*.

SystemC delimiter The GREENREG framework is developed using the dot (.) as the SystemC delimiter within names, which follows the standard. Contrary to GREENCONTROL, GREENREG does not allow to change this delimiter.

GreenReg specific report macros The end user should never use the internal GREENREG report macros like `GR_ERROR`, `GR_REPORT_ERROR` etc!

2.2 Concepts and Background

2.2.1 Register accesses

There are two basic GREENREG ways accessing registers:

1. Access the register directly from within the register or using the register container
2. Access the register from the bus

If the register makes a difference between its in and out buffers (e.g. a splitio register) it is important to know the behavior the different accesses:

Way of access	*Accessed buffer*	*Example*
set function (and operator =) Parameter access	write *out buffer*	`r[0x01] = 5;`
get function	read *in buffer*	`unsigned int val = r[0x01];`
bus write	write *in buffer*	`m_master_port->write(0x01, dat, 4);` `r.bus_write(data, 0x01, offset);`
bus read	read *out buffer*	`cr = m_master_port->read(0x01, 4);` `r.bus_read(data, 0x01, offset);`

The references i and o of class I_register can be used to access the buffers explicitly.

2.2.2 Types of configuration utilization

GreenReg internally uses two different ways of connecting GreenReg data types to the GreenConfig framework to provide the GreenReg data types as configurable parameters:

1. **The internal GreenReg data storage *is* a GreenConfig parameter.**
   This is like it's done for dr_attribute. This needs no further GreenReg specific explanation. See the GreenConfig User's Guide how to use parameters.
2. **The parameter API is implemented by the GreenReg data type to *make* it a parameter.**
   For a guide how to make use of the gs_param_drf-adapter see section 2.6.1.

2.3 Modeling Style with Functions and Notification Rules

This section is about how to model registers and the dependent methods or functions with GreenReg.

The basic modeling concept is that you have registers which can be equipped with different kinds of methods (like SC_METHODs) or more efficient functions (GR_FUNCTIONs) to react on register changes.

The procedure is the following:

- Create a register
- Announce a method/function (and implement it)
- Create a notification rule the method/function is sensitive to and which defines under which conditions the method /function will be notified/called (possibly delayed).

2.3.1 Methods/functions and their Characteristics

The different methods being available for notification rule notification / callbacks are:

- GR_FUNCTION
- GR_FUNCTION_PARAMS
- GR_METHOD *(not recommended)*
- SC_METHOD *(not recommended)*

Please report to the author(s) of this document which methods/functions you use most and which ones are needless in your opinion.

GR_FUNCTION A GR_FUNCTION is an equivalent to the SystemC SC_METHOD with the difference that it is activated by a gr_event by callback, not by an sc_event notify. This can either happen immediately, with an SC_ZERO_TIME delay (like usual sc_events behave) or with another delay, see the different sensitive macros (2.3.2) and the socket-wide delay switch (2.3.3.4).

⚠ A GR_FUNCTION does not need (and even cannot handle) a dont_initialize() call.

⚠ A GR_FUNCTION is not allowed to use SystemC SC_METHOD or SC_THREAD specific elements because the function is called by the callback in (potentially) any SystemC context.

For the callback register any void-void function with the following signature using the GR_FUNCTION macro:

```
GR_FUNCTION(class_name, function_name);
// will be concatenated to &class_name::function_name
// which needs to be of type callback_type
typedef void(class_name::*callback_type)()

// e.g.

void function_name();
```

Example (stripped-down):

```
void end_of_elaboration() {
  GR_FUNCTION(UserDevice, gr_function_callback);
  GR_SENSITIVE(r[0x01].add_rule(gs::reg::POST_WRITE, "pw1", [...]));
}
void gr_function_callback() {
  cout << "got not delayed post write notification for Reg 0x01" << endl;
}
```

GR_FUNCTION_PARAMS A GR_FUNCTION_PARAMS enhances the GR_FUNCTION with arguments. When being called, the callback function gets the transaction which caused the register access (if there was a bus access, otherwise it will get NULL) and the delay the function call had been delayed (if it had been delayed, otherwise it will get SC_ZERO_TIME).

⚠ Remark: The called function is not able to distinguish between an immediate (not delayed) call and a zero time (delta cycle) delayed call.
TODO: The function could get another (bool) parameter showing if the call had been delayed.

⚠ A GR_FUNCTION_PARAMS does not need (and even cannot handle) a dont_initialize() call.

⚠ A GR_FUNCTION_PARAMS is not allowed to use SystemC SC_METHOD or SC_THREAD specific elements because the function is called by the callback in (potentially) any SystemC context.

For the callback register a function with the following signature using the GR_FUNCTION_PARAMS macro:

```
GR_FUNCTION_PARAMS(class_name, function_name);
// will be concatenated to &class_name::function_name
// which needs to be of type callback_type
typedef void(class_name::*callback_type)(gs::reg::transaction_type*&, ⤸
   const sc_core::sc_time&)

// e.g.

void function_name(gs::reg::transaction_type* &tr,
                         const sc_core::sc_time& delay);
```

Example (stripped-down):

```
void end_of_elaboration() {
  GR_FUNCTION_PARAMS(UserDevice, gr_function_callback_p);
  GR_SENSITIVE(r[0x01].add_rule(gs::reg::POST_WRITE, "pw1", [...]));
}
void  gr_function_callback_p(gs::reg::transaction_type* &tr, const ⤸
   sc_core::sc_time& delay) {
  cout << "got not delayed post write notification for Reg 0x01 with ⤸
     params" << endl  << "transaction ID = " << tr->getTransID());
}
```

GR_METHOD A GR_METHOD is a GR_FUNCTION which is additionally an SC_METHOD internally. It should only be used if special features of the SystemC method are needed.

⚠ A GR_METHOD *does handle* the dont_initialize() call. You need to use it to avoid activation during initialization.

⚠ A GR_METHOD is not allowed to use SystemC SC_METHOD or SC_THREAD specific elements because the function is called by the callback in (potentially) any context.

Example (stripped-down):

```
void end_of_elaboration() {
  GR_METHOD(UserDevice, gr_method_notification);
  GR_SENSITIVE(r[0x01].add_rule(gs::reg::POST_WRITE, "pw1", [...]));
```

```
  dont_initialize();
}
void gr_method_notification() {
  cout << "got post write notification (callback or notify) for Reg 0x01"
     << endl;
}
```

The three GR_ functions/methods all use callbacks to call the user-registered function. This happens either immediately (efficient!) or delayed by an internal payload event queue (inefficient as SystemC methods but with payload if desired).

Implementation Details

Why GR_FUNCTION / METHODs?

The GR_FUNCTIONs have been introduced due to performance reasons: The default SystemC way is to use SC_METHODs which are triggered by sc_events. Events are the worst thing for performance because the SystemC kernel needs to do a context switch. The idea of GR_METHODs is using simple callbacks (that's why the macro needs the class name) without the need of the context switch.

This has the disadvantage that you do not know in which context the method is called (may be SC_METHOD, an SC_THREAD,...). Accordingly you are not allowed to use specific SC_METHOD calls like next_trigger(). We cannot bypass this issue following the standard.

SC_METHOD An SC_METHOD should only be used if special features of the SystemC method are needed.

⚠ Of course handles an SC_METHOD the dont_initialize() call.

When using SC_METHODs you need to enable the notify within the gr_event - which may be disabled by default. See section 2.3.3.2 how to use SC_METHODs with event notification rules and for event switch details see section 2.3.5.

2.3.2 Sensitivity

The three different methods/functions are used in combination with a sensitivity statement:

- SC_METHODs use the standard SystemC `sensitive << notification_rule_event;` whereas
- GR_FUNCTIONs and GR_FUNCTION_PARAMS can use either
  - GR_SENSITIVE(notification_rule_event); or
  - GR_DELAYED_SENSITIVE(notification_rule_event, delay_time);,

  see examples and details below.

SystemC sensitivity for SC_METHODs It is possible (but not recommended) to use the notification rule's gr_event as an input for the usual SystemC SC_METHOD sensitivity. See sections 2.3.3.2 and 2.3.5 how to use the event switch.

GR_SENSITIVE The macro GR_SENSITIVE can be used as a replacement just like the standard sensitive statement. The notification rule's gr_event will make an immediate callback to the GR_FUNCTION (or GR_FUNCTION_PARAMS) when the notification rule condition matches.

This callback is independent from the event switch (sec. 2.3.5)!

Macro syntax:

```
GR_SENSITIVE(notification_rule);
```

GR_DELAYED_SENSITIVE The macro GR_DELAYED_SENSITIVE causes a callback similar to the previous one but can delay this callback. The macro gets an additional time parameter which defines the time the callback should be delayed. If the delay is applied can be switched on and off with the *delayed switch* (sec. 2.3.3.4).

This callback is independent from the event switch (sec. 2.3.5)!

⚠ Remark: By default (only) all bus accesses are delayed with the delay time specified in the macro. Direct local access notifications are not delayed. Hence the delayed sensitivity should only be used for bus access notifications. Local accesses to bus_read and bus_write functions can be delayed by switching the according parameter to true.

Macro syntax:

```
GR_DELAYED_SENSITIVE(notification_rule, sc_time);
```

See the example greenreg/examples/simple for a complete example using different types of sensitivities and functions.

2.3.3 Notification Rules and Callbacks

This is only a short introduction to notification rules, what they are for and how to use them. See the Tutorial Slides for more details. This section handles the configurability of notification rules (subsection 2.3.3.1) and the difference between event driven notification rules and callback driven ones (further subsections).

Implementation Details

Notification rules

Notification rules are added by calling `add_rule` on a register, attribute or bit_range. This creates an `I_notification_rule` (one of its implementations) which owns a `dr_event`. This `dr_event` is notified when a rule is processed.
The `dr_event` can be switched to use an `sc_event` being notified *or* to use a callback.

There are two ways the user can get notifications / callbacks from registers. The concept is to keep the notification rules and configuration parameter callbacks separated.

- Use *parameter callbacks* for configuration and analysis of your system but *not* for modeling.
- Use *notification rules* to model your system (they are highly configurable and provide special register features).
  Notification rules provide two ways of usage:
  - Notification rules with events (default, see section 2.3.3.2):
    When the rule is matched, the event which is returned by the `add_rule()` call is fired.
  - Notification rules with callbacks (see section 2.3.3.3):
    When the rule is matched, the registered callback is called instead of the event being fired.

2.3.3.1 Configuration of Notification Rules

See the Tutorial Slides and further (not yet existing) documentation about how to specify notification rules.

2.3.3.2 Event-based Notification Rules *(not recommended!)*

Event-based notification rules are the default SystemC-like way using `SC_METHOD`s being activated by events. The advantage of this way is that the behavior of notifications and method activations is very SystemC-like, so SystemC users get the behavior they expect. The drawback of this way is the lack of efficiency. The notification of the events consume simulation time as well as the wait statements that are needed allowing pre- and post-notifications on each register change. If no hard requirements conflicts with callbacks, better use callback-based notification rules (see section 2.3.3.3).

Whenever you use this way you should ensure the events are enabled by calling enable_events() for the register. For legacy code the events can be (are *for the current release*) enabled by default (see section 2.3.5).

Example usage:

```
void end_of_elaboration() {
  SC_METHOD( show_notification_reg1_SC_M );
  sensitive << r[0x01].add_rule(GreenReg::USR_OUT_WRITE, 
     "written_to_reg1", GreenReg::NOTIFY);
  dont_initialize();
  r[0x01].enable_events();
}
void show_notification_reg1_SC_M() {
  cout << sc_time_stamp() << "got register notification for Reg1" << endl;
  cout << "value = 0x" << hex << r[0x01] << dec << endl;
}
```

2.3.3.3 Callback-based Notification Rules

The more efficient way of modeling is using callback-based notification rules. They are used in conjunction with GR_FUNCTIONs which do not need kernel context switches and events being fired but uses direct efficient function calls. When having switched off events (see section 2.3.5), even the wait in between the pre- and post-calls is omitted.

See the following example how to use GR_FUNCTIONs. The macro needs the class the callback needs to be called and common the sc_sensitive is replaced by a macro:

```
void end_of_elaboration() {
  GR_FUNCTION(MyMod, show_notification_reg1_GR_M);
  GR_SENSITIVE(r[0x01].add_rule( GreenReg::USR_OUT_WRITE, 
     "written_to_reg1", GreenReg::NOTIFY));
  dont_initialize();
}
void show_notification_reg1_GR_M() {
  cout << "got register notification call for Reg1" << endl;
  cout << "value = 0x" << hex << r[0x01] << dec << endl;
}
```

2.3.3.4 Delayed Switch

The *delayed switch* activates and deactivates the delay of pre- and post- write and read notification rules being caused by bus accesses.

The delay activation is applied to the notification rule by the receiving socket (class GSGPSOCKET::slave_base) on each received register access. Hence there is a switch in the slave socket that can be toggled to enable and disable the delay dynamically during simulation runtime. The switch will apply to all register accesses over this socket.

The interface is the following one:

```
/// Disables the delay
/// for all notification rule callbacks caused by this socket
void disable_delay();
/// Enables the delay
/// for all notification rule callbacks caused by this socket
void enable_delay();
/// Returns if the delay is enabled
bool delay_enabled();
```

Example:

```
class ReceiverSlaveDevice
: public gs::reg::gr_device
{
public:
  // Slave socket with delayed switch
  gs::reg::greenreg_socket< GSGPSOCKET::generic_slave> m_slave_socket;
```

```
  GC_HAS_CALLBACKS();
  SC_HAS_PROCESS( ReceiverSlaveDevice );

  ReceiverSlaveDevice(sc_core::sc_module_name name)
  : gr_device(name, gs::reg::INDEXED_ADDRESS, 2, NULL)
  , m_slave_spcket( "slave_socket", r, 0x0, 0xFFFFFFFF) // Slave socket
  { [...] }
  // SC_THREAD which demonstrates how the delayed switch can be switched
  void delayed_switch_demo() {
    wait(11, sc_core::SC_NS);
    m_slave_port.disable_delay();
    wait(10, sc_core::SC_NS);
    m_slave_port.enable_delay();
  }
}
```

2.3.3.5 Order Notification Rules

In order to create dependencies between several notification rules there is the option to order the existing rules:

Notification rules are stored in an ordered vector (in the notification_rule_container). The order can be manipulated using the functions `move_rule_to_front` and `move_rule_to_back` both moving the notification rule specified by name to the desired position.

Example:

```
DR_METHOD(MyMod, show_notification_reg0_DR_1);
DR_SENSITIVE(r[0x00].add_rule(
  DRF::USR_OUT_WRITE, "written_to_reg0_DR_1", DRF::NOTIFY));
dont_initialize();

[...]

r[0x00].get_user_obuf_write_rules().
   move_rule_to_back("written_to_reg0_DR_1");
```

2.3.4 Use cases

This section gives a short overview over some use cases and suggestions how to model.

Efficient standard case

Case: Standard case: You want to model a more efficient SC_METHOD-like function without using inefficient events.
Suggestion: Use an GR_FUNCTION with the GR_SENSITIVE sensitivity. The function will be called each time the notification rule matches, immediately (without using any delay).

Callback gets transaction

Case: Modification: you want to get the transaction having caused the register access.
Suggestion: Use the GR_FUNCTION_PARAMS instead of the GR_FUNCTION.

Delay callback

Case: Special use case: You want to delay the call of the function for a specific (fixed) delay.
Suggestion: Use the GR_DELAYED_SENSITIVE sensitivity instead of the GR_SENSITIVE. This will cause the notification rule to delay its call for the specified time. This delay may be SC_ZERO_TIME – which will lead to the well-known SC_METHOD-bahavior but with the ability to get parameters (transaction, delay). Combine this sensitivity with GR_FUNCTION or GR_DELAYED_FUNCTION.

Switch delayed / immediate

Case: Special use case: You want to switch between delayed / immediate call dynamically.
Suggestion: Use the GR_DELAYED_SENSITIVE sensitivity and use the socket-wide *delayed switch* (see section 2.3.3.4) to enable or disable delay.

2.3.5 Switch Event Behavior

For a register a switch can be toggled to switch off all events being notified by notification rules. Under the precondition that all notification rules are connected to GR_FUNCTIONs - and no SC_METHODs - a register should be switched to process the rules not using events due to performance reasons.

Note that a notification rule (or a gr_event) having registered a callback will never notify its event, so don't use the event being returned by the notification rule!

For the current release: By default the switch is enabled for legacy code support reason, so that no confusion appears when using SC_METHODs with notification rules. *For now* the recommended way is to disable this switch for each register and only use `GR_FUNCTION`s. In the future the default will be the disabled switch, so prepare your code to enable the switch where needed, too.

```
r.create_register( "Reg1", "Test Register1", 0x01, [...] ); // shortened
r[0x01].disable_events();
```

```
// maybe you want later enable events again:
r[0x01].enable_events();
```

The following itemization illustrates the behavior when a register change causes notification rule actions, depending on the event switch:

- In the case of *events being **en**abled*,
  - all callbacks and events concerning the pre-rules are notified,
  - then a `wait(SC_ZERO_TIME)` is called

- then the value is set (after all event having started the `SC_METHOD`s),
- then the post-rules are performed by performing callbacks and notifying the events.

When having enabled the event switch: Due to the need of calling wait in between, the user may only cause register changes within SC_THREADs, *never within SC_METHODs* which do not allow waits being called in their context.

- In the case of *events being **dis**abled*,
  - all callbacks concerning the pre-rules are performed (notification rules only having events will not notify anything),
  - then the value is set,
  - then the post-rules are performed by calling the functions.

  All this is done without calling any wait. Since the called functions are `GR_FUNCTION`s, they are not allowed to wait as well.

Note that all `GR_FUNCTION`s are called before all `SC_METHOD`s.

Implementation Details
Event switch hack
`GR_FUNCTION`s are internally registered as `SC_METHOD`s without ever being notified but directly called. Hence it will work to register a notification rule callback to a user created `SC_METHOD`.

Implementation Details
Default switch state
The default for the event switch is defined in the file gr_settings.h with the macro GR_DEFAULT_EVENT_BEHAVIOR.

2.3.6 Notes

- The set function of the register data calls all notification rules of the register and of all bit ranges of this register. Hence all notifications are performed if the register is accessed.

2.4 Write Mask, read-only

The write mask for a register defines which bits are allowed to be written to – and which ones are read-only. A write mask can be

- specified during construction or

- modified by calling set_write_mask.

The write mask is applied on every bus write and direct user write (set). Any write to read-only bits is ignored, the other bits are applied (just as one would expect hardware to react).

Basically a warning is shown when

- a bus write to write protected bits is performed or
- a direct user write (set) accesses protected bits.

Two different types of warnings can be chosen by configuring the two report handlers. By default both warning are enabled, the user should disable at least one of them.

- The unequal current warning type warns if any write protected bit is written with a value different from the current one.
- The unequal zero warning type warns if any write protected bit is written with a value different from zero (0).

```
sc_core::sc_report_handler::set_actions( 
    "/GreenSocs/GreenReg/write_protected/unequal_current", 
    sc_core::SC_DISPLAY);
sc_core::sc_report_handler::set_actions( 
    "/GreenSocs/GreenReg/write_protected/unequal_zero", sc_core:: 
    SC_DO_NOTHING );
```

2.5 Configurable Registers

All registers in GreenReg are automatically configurable using the GreenConfig configuration mechanism. The registers are presented as GreenConfig parameters and can be written and read.

Implementation Details

Configuration registers

There are two ways the GreenReg constructs are presented as GreenConfig parameters: All register types derive from a GreenReg specific parameter class (gs_param_greenreg) and implement some additionally required functions. Rarely (e.g. gr_attribute) gs_params are directly used within the GreenReg code.

2.5.1 Register's parameter attributes

All GreenReg registers automatically get some GreenConfig param_attributes.

As a replacement for the GreenConfig wrapper class gs_state which can be used for static code analysis, tools can search for the gs_param_greenreg base class which automatically adds the attribute gs::cnf::param_attributes::state.

GreenReg register type	GreenConfig parameter attributes
gr_register_sharedio	gs::cnf::param_attributes::gr_register, gs::cnf::param_attributes::gr_sharedio_register, gs::cnf::param_attributes::state
gr_register_splitio	gs::cnf::param_attributes::gr_register, gs::cnf::param_attributes::gr_splitio_register, gs::cnf::param_attributes::state
sharedio_bit_range	gs::cnf::param_attributes::gr_bit_range, gs::cnf::param_attributes::state
splitio_bit_range	gs::cnf::param_attributes::gr_bit_range, gs::cnf::param_attributes::state

---

Implementation Details

Parameter attribute state

The parameter attribute gs::cnf::param_attributes::state is appied automatically to the registers because their base class gs_param_greenreg adds this attribute during construction.

2.6 Expanding GreenReg

This section is for developers expanding GreenReg with new register types.

2.6.1 How to add a new register type and utilize it with parameters

Follow these rules to utilize a drf register or other data type with a GreenConfig parameter adapter (see way 2 of section 2.2.2).

- Include gs_param_greenreg.h
- Derive the utilized class from public gs::gs_param_greenreg<datatype>
  The class needs to implement the add_rule(...) function to let the drf parameter class register notification rules that will be mapped to parameter callbacks.
- Call init_param() from the *lowest* constructor in hierarchy (when the object is fully completed).
- In the constructor set the desired parameter attributes. E.g.:

```
add_param_attribute(gs::cnf::param_attributes::drf_register);
add_param_attribute(gs::cnf::param_attributes::drf_splitio_register);
```

- Implement the two virtual functions `set_drf_value()` and `get_drf_value()` to give the drf parameter access to the data.
- Implement the function `std::vector<std::string> add_post_read_param_rules()`.
  This function shall add all notification rules needed for (post read) parameter callback mapping. The functions returns all names of the added notification rules.
- Implement the function `std::vector<gs::reg::dr_notification_rule_container*> get_param_rules()`.
  This function shall return at least all the notification rule container(s) which contain(s) the rule(s) being added by `add_post_read_param_rules`.
- Follow the rules in section 2.6.2.

2.6.2 How to equip classes with the notification rule event switch

Follow these rules to equip a GreenReg register or other data type with a switch enabling/disabling the events of the managed notification rules. This switch is needed for all classes/data types that manage different notification rules within notification rule containers and which need to decide if to call `wait()` or not. Details on the switch can be found in section 2.3.5.

- Include `I_event_switch.h` (e.g. see I_register.h).
- Derive the utilized class from `public I_event_switch` (e.g. see I_register.h).
- Implement the virtual functions `disable_events()` and `enable_events()` handling/performing the notification rule switch.

  Within the implementation of both functions

  - first call the base function, e.g. `I_event_switch::disable_events()` which will update the stored state (bool variable).
  - forward the enable/disable call to all notification rule containers of this data type.

  Example implementation (e.g. see I_register.cpp):

```
void I_register::disable_events() {
  I_event_switch::disable_events(); // updated the state bool
  // switch all owned notification rule containers
  get_pre_write_rules().disable_events();
  get_post_write_rules().disable_events();
  get_pre_read_rules().disable_events();
  get_post_read_rules().disable_events();
  get_user_ibuf_write_rules().disable_events();
  get_user_obuf_write_rules().disable_events();
}
```

- Insert to the function which is adding any new notification rule (function `add_rule(...)`) a reset of the currently added rule to the current status of the switch. (E.g. call one of the enable/disable events functions switching all owned rules, e.g. see register.cpp .)
- Make use of the switch state information where it is needed to check if to call wait (if events are enabled) or not (e.g. see primary_register_data.h).

2.7 Notes

- A read or write bus access to not existing registers causes a warning by default.
  Use the following report handler setting to suppress the warning:

```
sc_report_handler::set_actions(
  "/GreenSocs/GreenReg/wrong_register_access", SC_DO_NOTHING);
```

- A *splitio register* is a special register type having two different independent buffers for input and output. The user needs to synchronize the buffers manually.

2.8 Implementation Code

Visit the GreenSocs web page to get the newest revision of the GreenReg framework:
http://www.greensocs.com/projects/GreenReg