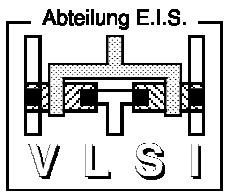
Efficient Configuration and TLM Registers

Using GreenConfig and GreenReg 4.1.0 – Tutorial



Christian Schröder, GreenSocs Ltd,

in cooperation with:
Integrated Circuit Design Lab (E.I.S.)
TU Braunschweig, Germany

Projects today

GreenSocs

- GreenConfig
  - Flexible configuration and inspection
    - Central and stable project already used by some other projects and users.
- GreenReg
  - Easy accessible register modeling

GreenConfig 4.1.0

Configuration and Inspection
Tutorial

Outline

GreenSocs

- Introduction
- Example Scenario
- Basics and Concepts
- Parameters
  - Implicit/explicit Parameters
  - Parameter Callbacks
  - Parameter Arrays
- Configuration API + File Parser
- OSCI CCI WG work
- Summary

This: Tutorial how to use GreenConfig Basics

- GreenConfig:

  GreenSocs Configuration Framework

- Goal is:

  Support all requirements of a complete configuration framework

Configuration Abilities

GreenSocs

- Term
  Configuration = Classical Configuration
  +
  Inspection
- Before (config) and during (control) runtime

More Features

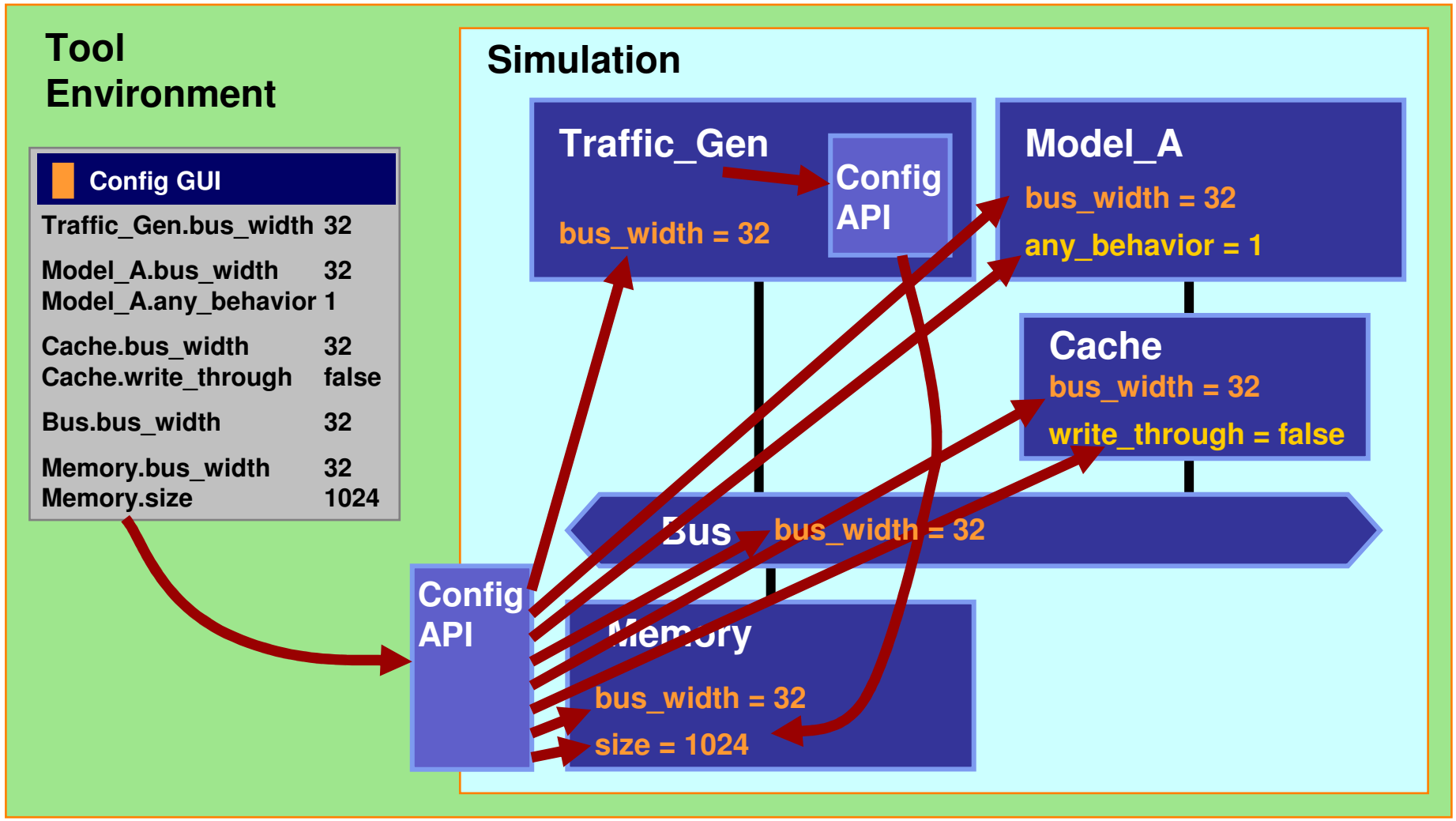
GreenSocs

- Not presented:
  - Internal Structure (GreenControl)
    - Designed for variable APIs
      for connection to other tools
  - Script language connection
- Combined many configuration features of other tools

Example Scenario

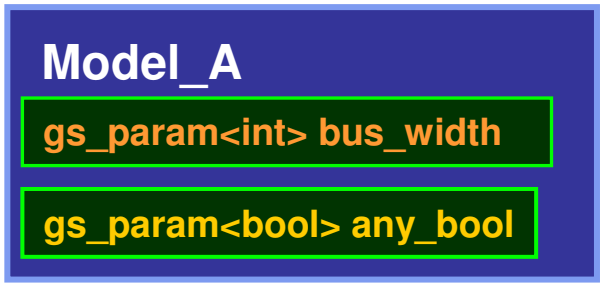
GreenSocs

Pre-Sim Parameters | Runtime Parameters

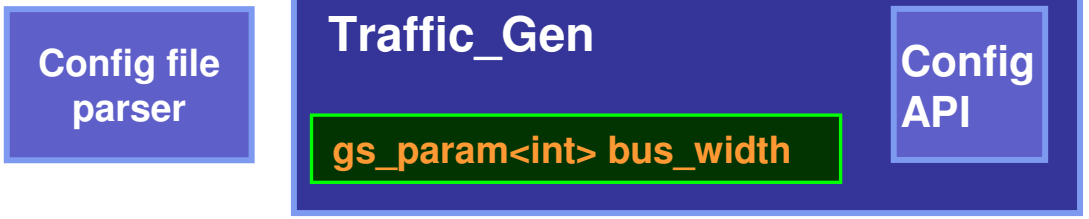


Configuration Concept

- Parameter objects



- User APIs
  - Config User API (for tools and models)
  - Config file parser
  - And others…



- Config Plugin
  - Underlying database containing all parameters
- Control Core
  - Underlying communication mechanism

Basics

GreenSocs

- Namespace

  `gs::cnf`

- Include File(s)

  One for all basics:

  `#include "greencontrol/config.h";`

  Only some special APIs need separate includes

Initialization

- Instantiation
  - Manually (recommended)
    - Rules:
      - Construct before any config usage/instantiation
      - Destruct after last usage

```
int sc_main(int argc, char *argv[]) {
  /// GreenControl Core instance
  gs::ctr::GC_Core        core;
  // GreenConfig Plugin
  gs::cnf::ConfigPlugin   configPlugin;

  // User modules etc.
  sc_start();
  return EXIT_SUCCESS;
}
```

Can be instantiated also in e.g. a top-level module

  - Automatically (experimental)
    - Note: currently not destroyed

Parameter Object

GreenSocs

gs_param

- Construction
  - Use one of the constructors
    (no limitations where to instantiate)

```
gs::gs_param<int>          my_param("my_param", 42          );
gs::gs_param<int>          my_param("my_param", "42"        );
gs::gs_param<int>          my_param("my_param", anyinteger);
gs::gs_param<int>          my_param("my_param"              );
gs::gs_param<bool>         my_bool ("my_bool",  true        );
gs::gs_param<sc_int<8> >   my_scint("my_scint", 42          );
```

data type

Parameter name
should be the same
as the variable name

Default value
provided as data type,
string or variable
or none

Parameter Object

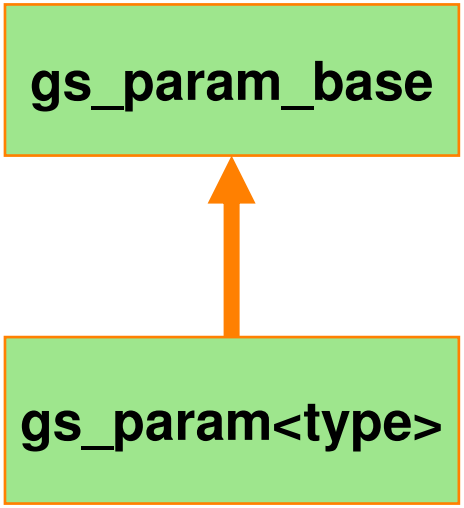
GreenSocs

gs_param

- Parameter names
  - Parameters are (SystemC) hierarchically named
    e.g. *topmodule.submodule.my_param*
  - The local name ("*my_param*") given by constructor
  - Special constructor allows complete naming (advanced)
- Access
  - Local access by owner using the parameter object
  - Global access by tool and other model using the → Config API (and hierarchical name) to get the parameter object

Inheritance Tree

- (Simplified) parameter inheritance



Type independent base class,

- used to store and return parameters,
- has string (de)serialize functions,
- has callback registration

Type templated class,

- used to create and use parameters,
- has value set and get operators,
- has manipulation operators

Parameter Object

GreenSocs

gs_param

- Parameter object access
  - Use as a replacement for its data type
  - Use as if it was the data type
  - Operators
  - Get and set string representation

```
gs_param::getString();
gs_param::setString("value");
```

Operators (as available):	
=	()
+	-
/	*
+=	-=
/=	*=
%=	^=
&=	|=
<<=	>>=
-- (pre-/ postfix)	++ (pre-/ postfix)
==	

Parameter Object - Example

GreenSocs

gs_param

```
class MyIP
: public sc_module
{
public:
  SC_HAS_PROCESS(MyIP);
  MyIP(sc_module_name name) : sc_module(name),
    int_param ("int_param", 1000),  // Constructor with default
    uint_param ("uint_param")       // Constructor without default
  { SC_THREAD(main_action); }

  main_action() {
    gs::gs_param<int> another("another", 1001);  // another parameter
    if (int_param != another) int_param++;       // operators != and ++
    int i = 500;
    if (int_param == another) int_param = i;    // operators == and =
    std::string str_representation = uint_param.getString();  // get string
  }

  gs::gs_param<int>           int_param;
  gs::gs_param<unsigned int>  uint_param;
};
```

Implicit vs. explicit Parameters

GreenSocs

gs_param

- Implicit parameter:
  - A parameter‘s initial value being set (and stored) in the database without an existing object
- Explicit parameter:
  - An existing parameter object, registered in the database
- Implicit → explicit:
  - On construction, a possible implicit parameter gets explicit. The initial value gets assigned to the object.

Supported Data Types

GreenSocs

gs_param

- Supported data types
  (See advanced features how to create new supported data types)
  - C++ types:
    int, unsigned int, bool, double, float, string, unsigned long long, long long, unsigned short, char, unsigned char, signed char,
  - SystemC types:
    sc_int_base, sc_int, sc_uint_base, sc_uint, sc_signed, sc_bigint, sc unsigned, sc_biguint, sc_logic, sc_time
    (with specialized classes including all operators)
  - sc_attribute<T> with T= all other supported types
  - All other data types that provide stream operators

OnChange Callbacks

GreenSocs

gs_param

- Function signature

```
void config_callback_method(gs::gs_param_base &changed_param);
```

- Register callback macro

```
GC_REGISTER_PARAM_CALLBACK(int_param, MyIP_Class,
config_callback_method);
```

- Module macros

```
GC_HAS_CALLBACKS(); // To be used in module body
GC_UNREGISTER_CALLBACKS(); // To be used in module destructor
```

(See advanced features for more callback support)

Callback Example

GreenSocs

gs_param

```
class MyModule
: public sc_module {
public:
  gs::gs_param<int> int_param; // parameter who should call on change
  GC_HAS_CALLBACKS();

  SC_HAS_PROCESS(MyModule);
  MyModule(sc_module_name name) : sc_module(name),
    int_param("int_param")
  {
    GC_REGISTER_PARAM_CALLBACK(&int_param, MyModule, config_callback);
    // Do some param changes, e.g. in an sc_thread
  }
  ~MyModule() {  GC_UNREGISTER_CALLBACKS();  }

  // Callback function with default signature.
  void config_callback(gs::gs_param_base& par) {
    if (par.is_destructing()) { /* dont access the value here*/ }
    else { /* do stuff */ }
  }
};
```

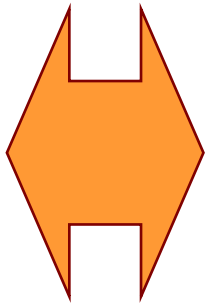
Parameter Arrays

GreenSocs

gs_param

- Arrays of parameters (`gs_param_bases`)

Simple
Parameter Arrays
Unnamed members
of same data type



Extended
Parameter Arrays
Named members
of any data type

Simple Parameter Arrays

GreenSocs

gs_param

- Construction

```
// Array of default size (currently 10, discussed to be 0)
gs::gs_param<int*> my_new_arr("my_new_arr");

// Array of size 3
gs::gs_param<int*> my_new_arr("my_new_arr", 3);

// Array with default values from a vector
std::vector<int> intvec; // and fill with values
gs::gs_param<int*> my_new_arr("my_new_arr", intvec);
```

- Access
  - Similar to vector: operator [], at(pos), resize(size)
  - Size and values can be set by config file or tools

Extended Parameter Arrays

GreenSocs

gs_param

- Constructor
  - All members need to be constructed (and named) explicitly

```
gs::gs_param_arr myArr("myArr"); // Array
// Members
gs::gs_param<int>  i_member("i_member",        myArr);
gs::gs_param<bool> b_member("b_member", true, myArr);
```

- Access
  - Config files (and tools) can only set existing members
  - Access with iterators, find(member_name) etc.

Advanced gs_param Features

GreenSocs

gs_param

Not shown in this tutorial:

- How to get an unknown Parameter type
- Complete gs_param inheritance tree
- How to create new parameter data types
- OnChange events (instead of callbacks)
- Forced top-level names
- Parameter attributes
- Parameter get_value_pointer
- Array details

Configuration API GCnf_Api

- Configuration API **`GCnf_Api`**:
  - Provide access to parameter objects for models and tools
  - Provide parameter lists, callbacks and more

Get a Configuration API

GreenSocs

Config API

- Static get function returns responsible Instance

```
gs::cnf::cnf_api *mApi =
gs::cnf::GCnf_Api::getApiInstance(sc_module*);
```

- Shall be used by a module with its this pointer

```
class MyMod : public sc_module {
public:
  SC_HAS_PROCESS(MyMod);
  MyMod(sc_module_name name) : sc_module(name) {
    mGcnfApi = gs::cnf::GCnf_Api::getApiInstance(this);
  }
protected:
  gs::cnf::cnf_api *mGcnfApi;
};
```

- Top-level and tools may use NULL

Initial Values

GreenSocs

Config API

- Sets a parameter's initial value

```
bool setInitValue(const string &parname, const string &value);
```

  - **`parname`** is the full hierarchical parameter name
  - **`value`** is the value string representation
  - returns if the setting was successful
- The init value has priority to the default value set by the owner!
- Can be used for not (yet) existing parameters
  → implicit parameter gets created

Get Parameters

GreenSocs

Config API

- Get a parameter object

```
gs_param_base* getPar(const string &parname),
```

  - returns a parameter base object pointer
  - NULL if not existing (or e.g. only implicit)
    - Can be used for existence check (for explicit param)

What to do with the parameter?

Param base pointer can be used for string value accesses

Can be converted/casted to actual param type →

```
gs_param_base* p = mApi->getPar("mod.par");
p->setString("1000.42");
cout << p->getString();
```

Get Parameter from Base Class

GreenSocs

Config API

- As long as type is known

```
// get parameter (base) pointer
gs_param_base *p = configApi->getPar("mymodule.int_par");

// convert base pointer to object of correct type
gs_param<int> *ip = p->get_gs_param<int>();
(*ip) = -42;
```

Get Parameter from Base Class

GreenSocs

Config API

- If type is not known
  - Use base parameter functions

```
const std::string getTypeString() const;
// or
const gs::cnf::Param_type getType() const;
```

  - and cast to the correct type

```
// get parameter (base) pointer
gs_param_base *p = configApi->getPar("mymodule.int_par");

// convert base pointer to object of correct type
switch p->getType() {
  case gs::cnf::PARTYPE_INT: {
    gs_param<int> *ip = p->get_gs_param<int>();
  }
  // other cases
}
```

Get Parameter String Value directly

GreenSocs

Config API

- Get value string representation from Config API

```
const string getValue(const string &parname);
```

- Independent of param implicit / explicit status
- Example:

```
gs_param_base* p = mApi->getPar("mod.par");
string val = p->getString();
```

```
string val = mApi->getValue("mod.par");
```

also for implicit parameters

Check if Parameter is existing

GreenSocs

Config API

- Check for explicit parameters
  - **`getPar`**
    (previously introduced on slide Get Parameters)

- Check for parameter existence
  independently of implicit / explicit status

```
bool existsParam(const string &parname);
```

Parameter Lists

GreenSocs

Config API

- Get a list (vector) of parameter names (implicit and explicit parameters)

```
1/  const vector<string> getParamList();
2/  const vector<string> getParamList(const string &module_name);
3/  const vector<string> getParamList(const string &module_name,
                                      bool including_childs);
```

1/ returns all params in the database

2/ returns all params of the given module

- without child's params by default,
- with child's params if `module_name` ends with ".*"

3/ equivalent to 2/ with ".*"

(Detailed wildcard rules, see advanced features)

Parameter Lists

GreenSocs

Config API

- Get a list (vector) of parameter base objects

```
const vector<gs_param_base*> getParams(const string &module_name = "");
```

  - Returns all or module‘s parameter base pointers
  - Same rules for **`module_name`**

New added Parameter Callbacks

GreenSocs

Config API

- Register callbacks for new added parameters (implicit or explicit)
- Function signature

```
void
config_callback_method(const string parname, const string value);
```

- Register callback macro

```
my_GCnf_Api.REGISTER_NEW_PARAM_CALLBACK(
                                MyMod_Class, config_callback_method);
```

Config File Parser API

GreenSocs

APIs

- Configuration file parser API
  - Parses configuration files and applies initial values to parameters
    - Include:

```
#include "greencontrol/config_api_config_file_parser.h"
```

    - Instantiate the API

```
gs::cnf::ConfigFile_Tool configTool("ConfigFileTool");
```

    - Parse hard-coded file

```
configfile  configTool.config("filename.cfg");
```

    - Parse command line argument given file + argument

```
configTool.parseCommandLine(argc, argv);
```

```
% ./my_simulation.x -gs_configfile filename.cfg
```

Config File Format

- Config files for config file parser API

```
# comment
hierarchical.parameter.name value
# e.g.
mymodule.submodule.parameterString  valueString
jpeg.compression 50
```

- Recommended string parameter format

```
mymodule.string_parameter "I am a string parameter with \"quotes\" inside"
```

APIs

- Lua config file API
  - Similar to config file parser API, but using LUA files

- Command line argument parser
  - Set initial values with command line arguments

```
% ./sim_name [-other switches or arguments]
      [--gs_param <param_name>=<param_value>]* [others]
```

  - e.g.

```
% ./my_simulation --gs_param myMod.myIntParam=10
--gs_param myMod.myStringParam="any string"
```

OSCI CCI WG work

- We participate in the OSCI Configuration, Control & Inspection (CCI) Working Group
  - We donated
    - the GreenConfig parameter concept,
    - a JSON (de)serialize concept,
    - the underlying GreenControl API.

GreenConfig Summary

GreenSocs

- Parameter objects
  - Type independent base class, templated param
  - Access
  - Callbacks
- Config API
- Other APIs (parsers)

GreenReg 3.0.0 (and 4.1.0)

Register Framework
Tutorial

Outline

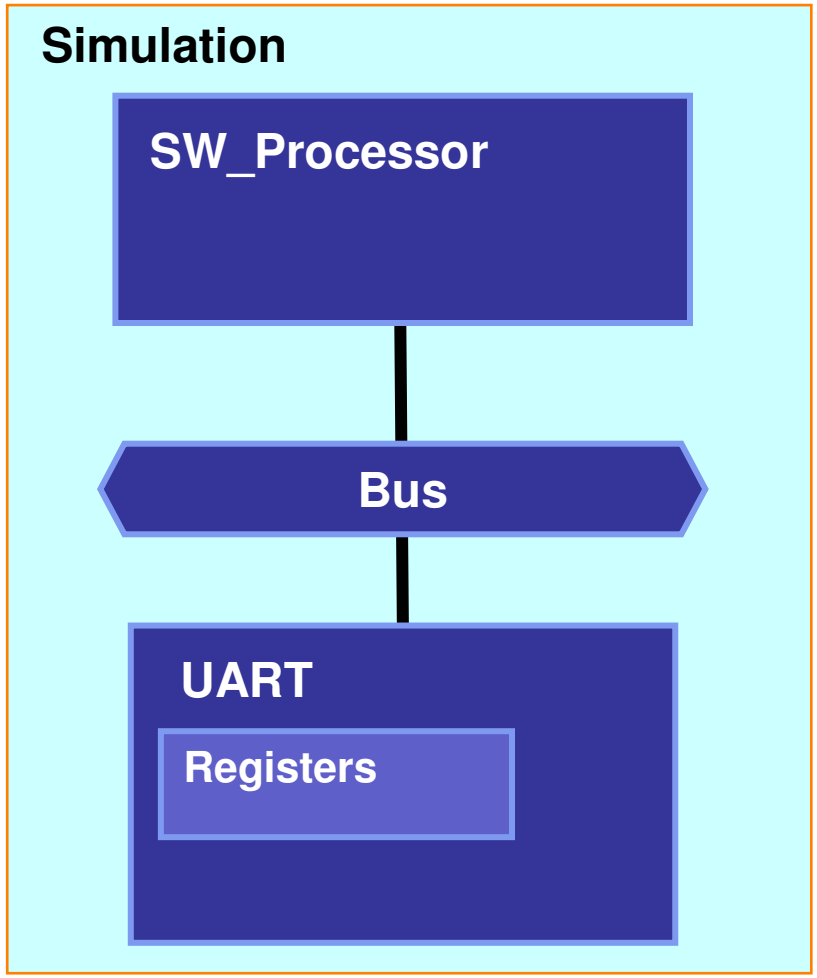
GreenSocs

- Introduction
- Example Scenario
- Basics and Concepts
- Devices
- Registers
- Bit access
- Functions and Notification Rules
- Convenience TLM Sockets
- Summary

This: Tutorial how to use GreenReg Basics

- Register framework – based on Intel DRF
  - Flexible register implementation
  - Devices provided
  - Easy hierarchical access to Registers and Devices
  - Simple bus access to registers
  - Configurable through background connection to configuration framework GreenConfig

Example Scenario



Basics

GreenSocs

- Namespace
  - gs::reg
- Include file(s)
  - #include <greenreg.h>
  - #include <greenreg_socket.h>
- Library
  - link to library libgreenreg

Implementation Concept

- Containers → Content
  - GreenReg components (devices, registers etc.) are organized in containers.
  - E.g.
    - A device container contains devices,
    - A register container contains registers
  - Containers are owned by components and allow access to their members
    - E.g. A device owns a register container with the device's registers

- Accessibility concept →

GreenReg Accessibility Concept

- General accessibility concept
  - Containers are component members, e.g.
    - d: device container
    - sd: subdevice container
    - r: register container
  - Access their members by operator [key], e.g.
    - sd["mysubdevice"].<other accesses>
  - Start locally e.g. within a device
  - Or use global container
    gs::reg::g_gr_device_container[“device_name”].<other accesses>

Devices

GreenSocs

Device

Devices and Subdevices

GreenSocs

- Use Cases (Devices, Subdevices)
  - Logical encapsulation layer, representing a block of capability
  - Derive from sc_module
  - Contain register containers

(See advanced features for the device container)

Device gr_device

Advanced:
when defining overlapping registers (in aligned address mode), the register's overwrite each other (should be avoided)!

- Inheritance
  - class my_device : public gr_device
- Constructor

```
gr_device(      sc_module_name   name,       // device name
                register_container_addressing_mode_e   address_mode, // address mode
                unsigned int   dword_size,   // dword size of entire memory  (if alignedmode)
                                             // number registers in memory (if indexed mode)
                device*   parent)            // parent gr_device or NULL
```

- Addressing mode
  - register_container_addressing_mode_e
    - ALIGNED_ADDRESS – strict alignment forcing 32 bit registers to 32 bit boundaries, 16 to 16, 8 to 8
    - INDEXED_ADDRESS – Allows non-aligned offsets to have any size register (32 bit max for now)

(See advanced features for full specification)

Subdevice gr_subdevice

GreenSocs

- Inheritance
  - class my_subdevice : public gr_subdevice
- Constructor
  - gr_subdevice(sc_module_name name, // device name
    l_device& parent) // parent gr_(sub)device

(See advanced features for full specification)

(Sub)Device Accessibility

GreenSocs

- Global or External Access
- gs::reg::g_gr_device_container[“dev_name”].sd[“subdev_name”].sd[“subdev_name”].<otheraccess>
- gs::reg::g_gr_device_container[“dev_name”].d[“encapsulated_dev_name”].<other accesses>

- Internal Access
  - sd[“subdevice_name”].<other accesses>
  - m_parent.sd[“subdevice_name”].<other accesses>

Registers

GreenSocs

Register

Registers

GreenSocs

- Flexible registers with focus on intuitive accessibility
- Default register container in device: ‘r’
- Create registers within (sub)devices
- Can be mapped to different vendor APIs under the hood

(See advanced features for how to create additional register containers)

Create Registers

GreenSocs

- Register container provides API to create
  - void
    create_register(string name, // register name for external access
    string description, // description of register
    uint_gr_t **offset**, // offset of register relative to device base
    unsigned int **type**, // type of register
    uint_gr_t **data** = 0x0, // default data
    uint_gr_t input_mask = ~0x0, // write mask for data coming in
    // from the bus
    uint_gr_t **width** = 32) // should always be equal to the
    // register container addressing mode

  - Example:
    - Create a single 32 bit register of type SPLIT_IO:
      create_register(“reg”, “desc”, 0x01, gs::reg::SPLIT_IO, 0xab, 0xff, 32);

Create Register Block

GreenSocs

- Register container provides API to create
  - void
    create_register_block(string description, // description of register
    uint_gr_t **offset_start**, // starting offset of registers
    // relative to device base
    uint_gr_t **offset_end**, // ending offset of registers
    unsigned int **type**, // type of register
    uint_gr_t **data** = 0x0, // default data
    uint_gr_t input_mask = 0xFFFFFFFF, // write mask
    uint_gr_t **width** = 32) // width

  - Example:
    - Create a block of 32 bit registers with offset 0x02 to 0x4c:
      create_register_block(“desc”, 0x02, 0x4c, gs::reg::SINGLE_IO, 0x00, 0xff, 32);

Register Access Mechanisms

GreenSocs

- Registers provide two kinds of access mechanisms:
  - Direct access, e.g. from inside the model (using the object or the access mechanism described earlier)
  - (TLM) bus access

Register Access

GreenSocs

- Register interface (I_register)
  - operator uint32 () – **returns** copy of register **data**
    get()
  - operator = (uint32) – **assigns** value to register **data**
    set(uint32)
    put(uint32)
  - get_register_type(), get_width()
  - set_write_mask(uint32), get_write_mask()
  - bool bit_get(uint32 bit), bit_set(bit, data), bit_put(bit, data)
  - bit_is_writable(bit), bit_set_writable(bit, writeable)
  - access to input buffer ‘i’ and output buffer ‘o’

- Container
  - ‘b’ – bit container
  - ‘br’ – bit range container

(See advanced features for complete interface)

Register Container Bus Access

- Register bus interface (I_register_container_bus_access)
  - To be called over a bus (e.g. the →greenreg_socket)

```
bool bus_read(   unsigned int&      data,                 // data buffer to fill
                 unsigned int       address,              // offset address
                 unsigned int       byte_enable,          // byte enable
                 transaction_type*  transaction  = NULL,  // transaction initiating this call
                 bool               delayed      = true)  // if the → notifications shall be
                                                          // delayed or called directly
```

```
bool bus_write(  unsigned int       data,                 // data
                 unsigned int       address,              // offset address
                 unsigned int       byte_enable,          // byte enable
                 transaction_type*  transaction  = NULL,  // transaction initiating this call
                 bool               delayed      = true)  // if the → notifications shall be
                                                          // delayed or called directly
```

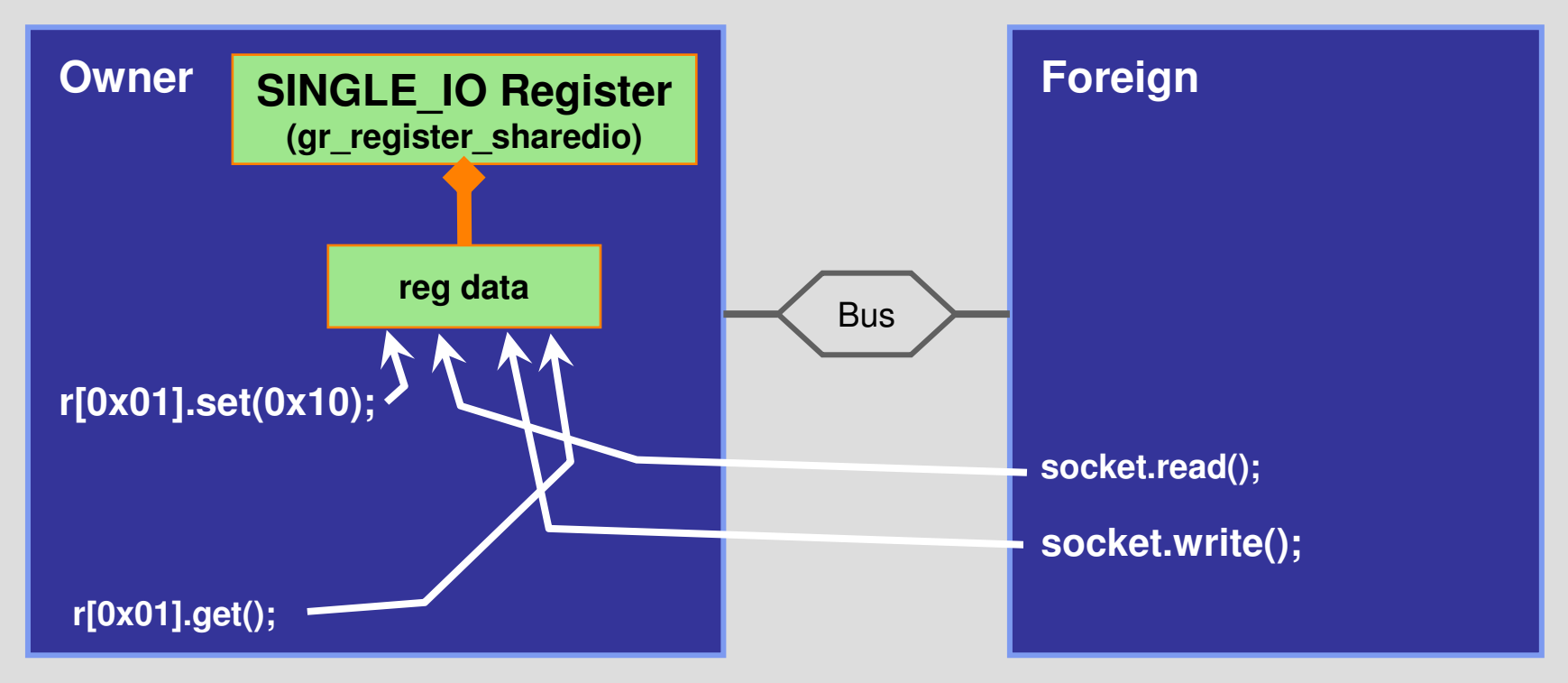
Register Types

GreenSocs

- Two register types available:
  - SINGLE_IO & SPLIT_IO

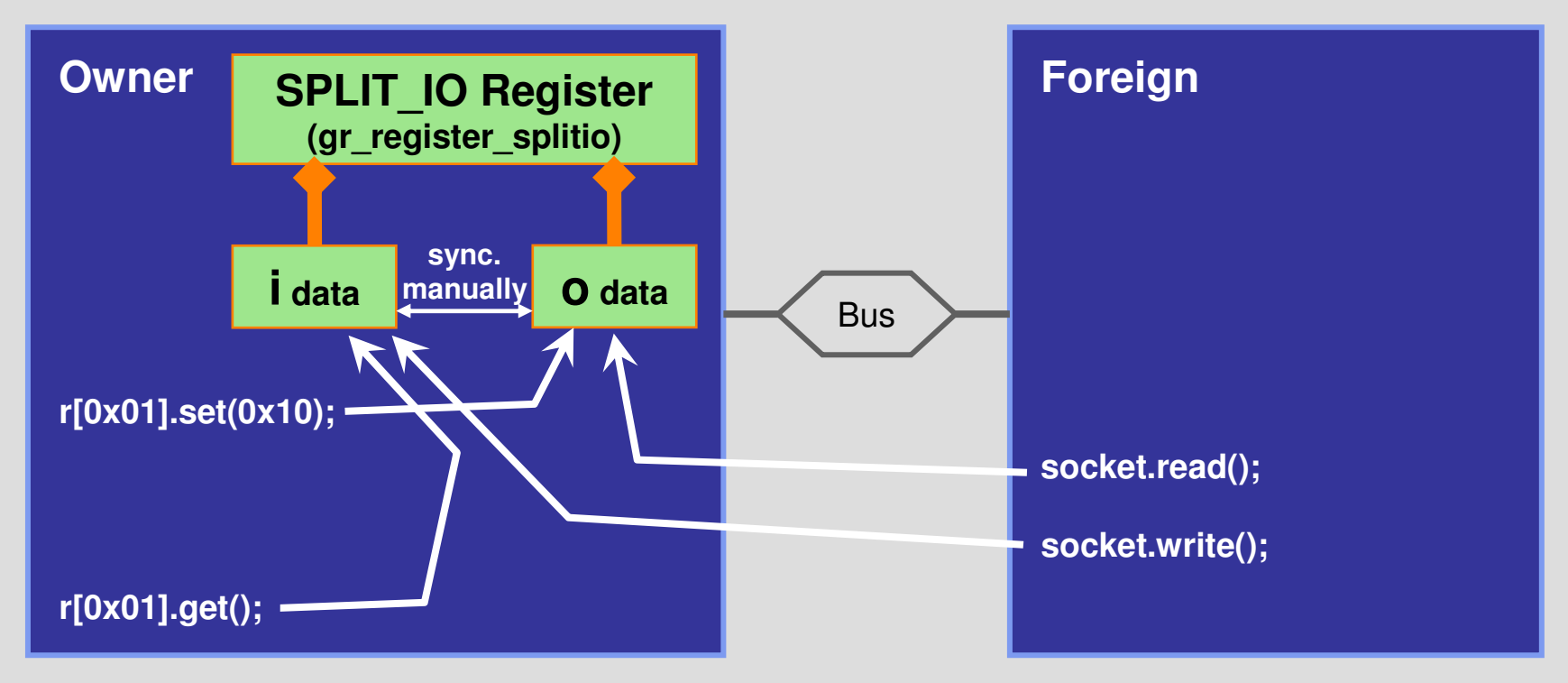
SINGLE_IO Register Type

- SINGLE_IO (default)
  - Reads and writes access the *same* data buffer



SPLIT_IO Register Type

- SPLIT_IO
  - Reads and writes access *separate* data buffers
  - To be synchronized by user e.g. with →notification rules



Register Accessibility

GreenSocs

- Global or External Access
  - gs::reg::g_gr_device_container[“dev_name”].sd[“subdev_name”].r[0x04] = 0xa0;
  - uint my_int = gs::reg::g_gr_device_container[“device_name”].r[0x04];
  - gs::reg::g_gr_device_container[“device_name”].r[0x04].<other accesses>
  - gs::reg::g_gr_device_container[“device_name”].r.bus_write(0xa0, 0x04, 0xf);
- Internal Access
  - r[0x04] = 0xa0;
  - uint my_int = r[0x04];
  - r[0x04].<other accesses>
  - r.bus_write(0xa0, 0x04, 0xf);

GreenSocs

Bit Access

Bit Accessors to Registers

GreenSocs

- bit_accessor (bit container)
  - Each register has a bit container: ‘b’
  - Dynamically creates bit level access objects upon first access

- [] operator (unsigned int)
  - returns bit at the specified index

```
r[0x01].b[2];
```

Bit Access to Registers

GreenSocs

- Bit Class (bit)
  - operator bool () get() – **returns** the value of the bit location
  - operator = (bool) set(bool) – **applies** the value to the bit location
  - is_writable() – returns the write mask setting for the bit location
  - set_writable(bool) – applies the value to the write mask setting for the bit location
- Accessibility
  - 'i', 'o' – access to input/output buffer
  - m_register – pointer to the register which "owns" this bit

(See advanced features for complete interface)

Bit Accessibility

GreenSocs

- Global or External Access
  - gs::reg::g_gr_device_container[“dev_name”].r[0x04].b[3] = true;
  - bool my_bit = gs::reg::g_gr_device_container[“device_name”].r[0x04].b[3];
  - gs::reg::g_gr_device_container[“device_name”].r[0x04].b[3].<other accesses>

- Internal Access
  - r[0x04].b[3] = true;
  - bool my_bit = r[0x04].b[3];
  - r[0x04].b[3].<other accesses>

Bit Range Container

GreenSocs

- bit_range_accessor (bit range container)
  - each register has a bit range container
  - bit access relative to start bit
  - supports multiple overlapping bit ranges

- Construct bit ranges:
  - bit ranges need to be created explicitly
    - create("name", start_bit, stop_bit);
- [] operator (string name)
  - returns the specified bit_range

Bit Range

- Bit Range (bit_range)
  - operator unsigned int () get()
    - **returns** the value of the bit_range
  - bit_range& operator = (unsigned int) set(unsigned int)
    - **applies** the value to the bit_range
- Accessibility
  - m_register – pointer to the register which “owns” this bit
  - ‘i’ – input data buffer
  - ‘o’ – output data buffer
  - ‘b’ – bit_range_bit_accessor relative to bit range starting bit

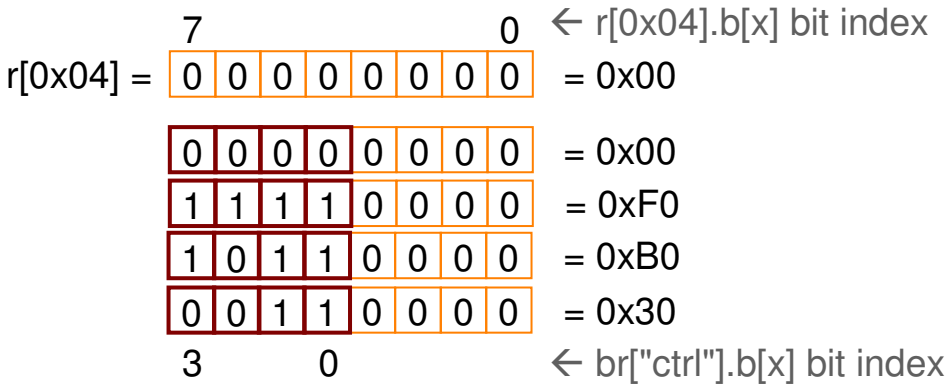
(See advanced features for complete interface)

Bit Range Accessibility

- Global or External Access
  - gs::reg::g_gr_device_container[“dev_name”].r[0x04].br.create("ctrl", 3, 5);
  - gs::reg::g_gr_device_container[“dev_name”].r[0x04].br["ctrl"] = 0xF;
  - gs::reg::g_gr_device_container[“dev_name”].r[0x04].br["ctrl"].b[0] = false;

- Internal Access
  - r[0x04].br.create("ctrl", 4, 7);
  - r[0x04].br["ctrl"] = 0xF;
  - r[0x04].br["ctrl"].b[2] = false;
  - r[0x04].br["ctrl"].b[3] = false;

  - unsigned int v = r[0x04].br["ctrl"];
  - bool b = r[0x04].br["ctrl"].b[2];
  - r[0x04].br["ctrl"].<other accesses>



GreenConfig Access Mechanism

GreenSocs

- Access register data with configuration parameters
  - All registers and bit_ranges are accessible by configuration parameters:
    - Type gs_param_greenreg<unsigned int>, deriving from gs_param_base
    - Hierarchical names are the same as the GreenReg register's name
    - Values are the ones returned by register.get() (IN buffer for split_io registers) – not bus_read()
    - Thus e.g. config file init values are possible
    - Callbacks are given
      - In current release possibly several times per access (on notifications in_write, out_write, post_write)
      - Planned to come only on in_write, which is the value returned by the parameter

GreenSocs

Functions

&

Notification Rules

Modeling Style

GreenSocs

- Functions and notification mechanisms
- GR_FUNCTIONs more efficient than SC_METHODs, and combine well with register notifications
  - Within a gr_device:
    - Create a register
    - Declare and implement a function
    - Create a notification rule which will notify the function

GR_FUNCTION

- Activated by gr_event (→ sensitivity)
  - either immediately (high performance)
  - delayed by SC_ZERO_TIME (similar to sc_event)
  - delayed by other sc_time
- No dont_initialize() call (does not initialize)
- Must not use SC_METHOD or SC_THREAD specific elements

```
void UserDevice::end_of_elaboration() {
  GR_FUNCTION(UserDevice, gr_function_callback);
  GR_SENSITIVE([any gr_event]);
}
void UserDevice::gr_function_callback() {
  /* Do stuff but no sc_method, sc_thread commands like wait */ }
```

GR_FUNCTION_PARAMS

GreenSocs

- Difference to GR_FUNCTION:
  - The callback function gets
    - the transaction causing the notification (if there was one)
    - the delay the function had been delayed (if so, otherwise SC_ZERO_TIME)

```
void UserDevice::end_of_elaboration() {
  GR_FUNCTION_PARAMS(UserDevice, gr_function_callback_p);
  GR_SENSITIVE([any gr_event]);
}
void UserDevice::gr_function_callback(
  gs::reg::transaction_type* &tr, const sc_core::sc_time& delay) {
  /* Do stuff */ }
```

(See advanced features for GR_METHOD and SC_METHOD and the register-wide event switch)

Advanced: Currently the callback function cannot differ between a not delayed and a zero time delayed call. This is a TODO.

Sensitivity

- GR_FUNCTION‘s sensitivity specified with GR_SENSITIVE or GR_DELAYED_SENSITIVE
- GR_SENSITIVE(gr_event)
  - e.g. event from → notification rule
- GR_DELAYED_SENSITIVE(gr_event, sc_time)
  - delays (only!) bus accesses, so use only with bus access notification rules
  - Delays can be switched off per slave socket

Notification Rules

GreenSocs

- Notification rules
  - for registers stimulating model activity
  - fire a gr_event (which possibly not fires but calls)
  - depending on register actions

- add_rule (to registers and bit_ranges)
  - returns a gr_event
  - should be defined in end_of_elaboration

```
gr_event& add_rule(   gr_reg_rule_container_e   container,   // Target container id of rule
                      std::string   name,                    // Unique name of rule
                      gr_reg_rule_type_e   rule_type,        // Type of rule
                      …)                                     // Advanced
e.g.: add_rule( POST_WRITE, “Post Write CWR”, NOTIFY )
```

Notification Rule

GreenSocs

- Enum dr_reg_rule_container
  - PRE_READ – add rule to pre read container
    fires before bus read access
  - POST_READ – add rule to post read container
    fires after bus read access
  - PRE_WRITE – add rule to pre write container
    fires before bus write access
  - POST_WRITE – add rule to post write container
    fires after bus write access
  - USR_IN_WRITE – add rule to model stimulated input buffer write container
    fires after direct write access to *input* buffer
  - USR_OUT_WRITE – add rule to model stimulated output buffer write container
    fires after direct write access to *output* buffer

For detailed notification rule container description see advanced features.

- Enum dr_reg_rule_type
  - NOTIFY – simply notifies for every access
  - further rule types: read/write patterns, bit states (see advanced features)

Notification Rule Order

GreenSocs

- Order the rules are fired
  - get()
    - no rule
  - set(…)
    - USR_OUT_WRITE
  - bus_read(…)
    - PRE_READ → POST_READ
  - bus_write(…)
    - PRE_WRITE → USR_IN_WRITE → POST_WRITE

GreenSocs

Convenience TLM Sockets

GreenReg TLM Socket

GreenSocs

- Registers can be accessed with bus interface
  - Register accesses on socket are automatically mapped to register accesses in the target
- Simple socket: greenreg_socket
  - as master: greenreg_socket<gs::gp::generic_master>, internally a GSGPSocket gs::gp::GenericMasterBlockingPort<32>
  - as slave: greenreg_socket<gs::gp::generic_slave>, internally a GSGPSocket gs::gp::GenericSlaveBlockingPort<32>

GreenReg TLM Master Socket

GreenSocs

- How to create
  - Class member

    gs::reg::greenreg_socket< gs::gp::generic_master > m_master_socket
  - Constructor

    m_master_socket("master_socket")

- Interface read, write

```
unsigned int ->read(  unsigned int   address,
                      unsigned int   width)
     void ->write(    unsigned int   address,
                      unsigned int   data,
                      unsigned int _width)
```

GreenReg TLM Slave Socket

- How to create
  - Within a gr_device!
  - Class member

    gs::reg::greenreg_socket< gs::gp::generic_slave > m_slave_socket
  - Constructor

```
greenreg_socket(sc_module_name   name,// name for the socket
                register_container&   reg_bind,  // register container (simplified)
                gr_uint_t   base_address,        // base address
                gr_uint_t   decode_size)         // decode size
```

    - Example:
      m_slave_socket("slave_socket", r, 0x0, 0xFFFFFFFF)
- Automatically connects to register container
- Bind like usual sockets

GreenReg TLM Slave Socket

GreenSocs

- Notification rule delay handling
  (see GR_DELAYED_SENSITIVE)
  - disable_delay() – Disables delaying of socket-related notification rules
  - enable_delay() – Enables delaying of socket-related notification rules
  - bool delay_enabled() – If the delays for notification rules are enabled

Further Features

- gr_attribute
- gr_switch

GreenReg Summary

GreenSocs

- We learned about
  - Devices, subdevices
  - Registers
  - Bit & bit range access
  - Functions, notification rules
  - Convenience TLM register sockets

Questions?

More information:

www.greensocs.com