

GreenAV Tutorial of use / Example (GreenAV v.1.1.1)

Copyright GreenSocs Ltd 2008

Developed by
Christian Schröder
and Wolfgang Klingauf and Robert Günzel
Technical University of Braunschweig, Dept. E.I.S.

31st July 2008

Contents

1	GreenAV Tutorial of use / Example	2
1.1	Introduction	2
1.2	Tutorial Full Example	3
1.2.1	Analysis features	4
1.3	Analysis Examples	5
1.3.1	Overall Hit Rate Statistics Calculator	5
1.3.2	Extrapolated Transactions per Second	6
1.3.3	Dynamic Transactions per Second	7
1.3.4	Transaction Latency	8
1.3.5	Transaction Recording	9

Chapter 1

GreenAV Tutorial of use / Example

1.1 Introduction

This document introduces SystemC developers to GREENAV: A **GREENCONTROL**¹ service plugin providing analysis and visibility features.

In this tutorial an example system will be equipped with analysis abilities. The example system uses the GREENBUS framework for communication between the modules. The SimpleBus protocol and ports are extended with analysis abilities. At least the GREENBUS version of April 21th, 2008² is needed. See the **GREENBUS project page**³ for more information on GREENBUS.

This example system (*demonstration platform*) is an advancement of the system developed in the **GREENCONFIG Tutorial of use**. This example requires the GREENCONFIG release 0.3 which is part of the GREENAV download.

This tutorial will *not* describe the connection between the modules of the system (using GREENBUS) and their functionality. See the GREENCONFIG Tutorial of use for these information.

The platform is composed of some user modules connected to two buses: Several traffic generators (TG1, TG2, TG3), three caches connected to the traffic generators (Cache1, Cache2, Cache3), and three devices which the traffic generators access: a memory (Mem) and two PCIe devices (PCleDevice1 and PCleDevice2). The caches and the memory module communicate through a SimpleBus (which is provided with GREENBUS), the PCIe devices are connected over a second bus (called PCleBridge) which is another SimpleBus.

The so-called PCIe devices and the PCIe bus are *not* PCI Express implementations since we need a timed bus simulation and the available GREENBUS PCI Express implementation only provides (untimed) PV level.

This tutorial will describe shortly the code snippets that are needed to equip the demonstration platform and the bus implementation with several analysis abilities. For more details and further and alternative analysis functionality use the **GREENAV User's Guide**.


The intended audience for this tutorial have to be familiar with C++ and SystemC.

¹GREENCONTROL project page: <http://www.greensocs.com/projects/GreenControl>

²The needed GREENBUS is the revision 2028 (or later) of the svn <https://svn.greensocs.com/public/packages/greenbus>

³GREENBUS project page: <http://www.greensocs.com/projects/GreenBus>

1.2 Tutorial Full Example

Figure 1.1 shows an overview of the demonstration platform we use in this tutorial. A sample implementation is in the directory  `greencontrol/examples/gav_demonstration_platform`. The modules are:

- *Random traffic generators (TG1, TG2, TG3)*: Acts as bus masters and create test traffic to the memory and the PCIe devices. Their behavior can be set with configurable parameters (and can be changed even during runtime).
- *Caches (Cache1, Cache2, Cache3)*: These caches are write-back LRU caches whose size and line size is configurable during runtime. They also can be disabled during runtime.
- *Router_SimpleBus*: The router is modeled by the SimpleBus router and needs not be developed as user module.
- *Memory (Mem)*: This memory module is a simple double data rate random access memory (DDR RAM). It is a bus slave. Its size and base address should be configured.
- *PCIe router (PCIeBridge_SimpleBus)*: This is another SimpleBus router connected to the Router_SimpleBus.
- *PCIe devices (PCIeDevice1, PCIeDevice2)*: The functionality of these bus slaves is just a stub with registers which can be written and read. The base address should be configured.

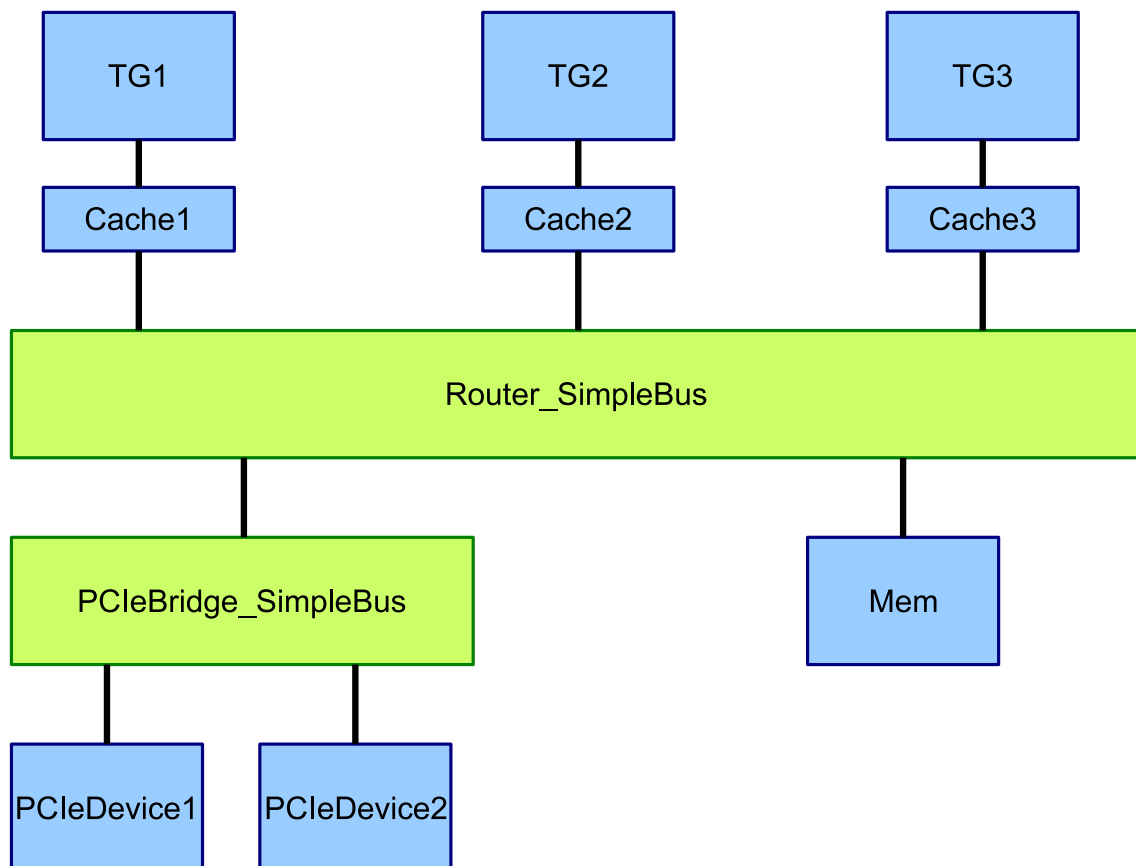


Figure 1.1: Analysis Demonstration Platform.

1.2.1 Analysis features

This is a list of the analysis features this demo platform provides:

■ *Cache hit rates:*

 `greencontrol/examples/analysis_demonstration_platform/lru_cache.*`

Each cache provides two hit rate analysis parameters:

- Overall hit rate
- Hit rate of a specific address range. The address range can be specified using configurable parameters of this module: `hitrate_upper_addr` and `hitrate_lower_addr`.

■ *Transaction rates:*

 `greenbus/protocol/SimpleBus/simpleBusProtocol.h`

The SimpleBus protocol implementation provides several kinds of transaction rates:

- Transactions per second over all ports, *extrapolated* to one second using the time between the first and the last transactions that were ever recorded in this simulation. Updated on each transaction. Does not show the utilization over time but the overall utilization of the ports.
- Transactions per second for each in and out port (*extrapolated* version). Same as transactions per second over all ports but separately for each port.
- *Dynamic* transactions per second over all ports, counting the actual transactions that were transmitted in the last second. Updated at the end of each second. Can be outputted to show the utilization over time (with a granularity of one second).

■ *Transaction latency:*

 `greenbus/api/simplebusApi.h`


The SimpleBus master port (API) provides two kinds of transaction latencies. The resulting value is the double representation of `sc_time`.

- *Transaction overall latency* (in the master port): Time a transaction needs between sending (in the master) and completion (in the master).
- *Transaction transport latency*: Time the request needs to be sent by the master and acknowledged by the slave (phases Request and RequestAccepted), measured in the SimpleBus master port.

■ *Transaction recording:*

 `greenbus/api/simplebusApi.h` (in the slave port)

Transaction recording can be switched on / off with a configurable parameter. Filtering to record transactions of only one master can be enabled by setting a configurable parameter to a master ID. *Note* that this transaction recording is an example and insists not on well performing and beautiful implementation.

To not influence the GreenBus as it is without GreenControl, all changes for analysis are surrounded by `#ifdefs` `GAV_ENABLED` and `#endif`. `GAV_ENABLED` is defined in the demo platform's global include file ( `demo_globals.h`) which is included in each file before including anything else.

1.3 Analysis Examples

This section picks up the most important parts of the analysis features and describes them in more detail. These examples show how analysis framework can be used, but they demand not to be complete. The later examples only handle new elements and do not repeat issues discussed in previous analysis examples.

1.3.1 Overall Hit Rate Statistics Calculator

See file  greencontrol/examples/analysis_demonstration_platform/lru_cache.*

The overall hit rate Statistics Calculator calculates the hit rate within a cache.

Since the cache class registers callbacks it calls the macros `GC_HAS_CALLBACKS()`; (in the body) and `GC_UNREGISTER_CALLBACKS()`; (in the destructor).

Two configurable parameters (`gs_params`) are used to count the overall hits and misses. They are incremented each time a cache hit or miss occurs. The incrementation is independent on the calculation done within the `StatCalc` and is done even if it is disabled.

```
gs::gs_param<unsigned int> overall_hits;  
gs::gs_param<unsigned int> overall_misses;
```

The class member pointer to the `StatCalc`

`gs::av::StatCalc<double> *overall_hit_rate;`
allows to delete the `StatCalc` and its Calculator in the destructor:

```
delete overall_hit_rate->get_calculator();  
delete overall_hit_rate;
```

- The calculator can be deleted before the `StatCalc` object. In this case the (default) trigger will be delete automatically by the `StatCalc` destructor. When instantiating a separate trigger, it must not be deleted before the `StatCalc`!

In the constructor the following Statistics Calculator (`StatCalc`) is created:

```
// Overall hit rate Statistics Calculator  
gs::av::Calculator<double> *hr_calc // Create Calculator  
    = new gs::av::Calculator<double>("overall_hit_rate");  
hr_calc->enable_sloppy(); // allow division by zero  
// prepare parameters for convenience operators  
gs::gs_param_base& hits    = overall_hits;  
gs::gs_param_base& misses = overall_misses;  
// set formula: 100*(hits / (hits+misses))  
hr_calc->calc("100", 100, (*hr_calc)(hits / (*hr_calc)(hits + misses)));  
// create Statistics Calculator  
overall_hit_rate =  
    new gs::av::StatCalc<double>("overall_hit_rate", hr_calc);
```

```
cout << "overall_hit_rate parameter name: "  
      << overall_hit_rate->get_result_param()->getName() << endl;
```

- Both, the Calculator (lines 2-3) and the StatCalc (lines 11-12), are *templated* to double. This means all results and intermediate results and calculations will be done using and returning the data type double. Never use two different templated classes here!
- The Calculator is made *sloppy* (line 4) to ignore divisions by zero.
- The Calculator's *formula* is set in line 9 with a mix of *convenience syntax* and *standard syntax*. For the convenience syntax the input parameters are casted to `gs_param_bases` (lines 6-7). The standard syntax is used for the multiplication because the convenience syntax does not support constants (100).
- The StatCalc is created in lines 11-12. There a *default trigger* will be created implicitly which triggers on each change of the formula input parameters (hits, misses).
- The *result parameter* can be accessed by calling `get_result_param()`.

Needed includes are:

```
1 // GreenControl and GreenConfig  
#include "greencontrol/gcnf/apis/gs_param/gs_param.h"  
// GreenAV  
#include "greencontrol/gav/apis/gav_api/GAV_Api.h"  
5 #include "greencontrol/gav/plugin/StatCalc.h"
```

1.3.2 Extrapolated Transactions per Second

See file  greenbus/protocol/SimpleBus/simpleBusProtocol.h

The GreenBus SimpleBus protocol has been extended to provide transactions per second statistics.

The needed analysis include is:

```
1 #include "greencontrol/gav/plugin/StatCalc.h"
```

A global static function is added for *special calculation needs* to be added to the double Calculator class:

```
1 /// Average calculator function to be added to the Calculator<double>  
static double per_second(double transaction_count ,  
                        double start_time_in_seconds , bool sloppy) {  
    return (transaction_count /  
5         (sc_time_stamp().to_seconds() - start_time_in_seconds));  
}
```

A class member parameter is used to count all transactions:

```
1 gs::gs_param<unsigned int> tr_cnt;
```

The class member pointer to the StatCalc (*overall_tr_per_sec) allows to delete the StatCalc and its Calculator in the destructor.

Within the protocol's constructor this code creates the StatCalc:

```
1 // Add user defined calculation function to Calculator class
gs::av::Calculator<double>::addFunc(&per_second, "per_second");

gs::av::Calculator<double> *trpsec_calc
5 = new gs::av::Calculator<double>("tr_p_sec_overall");
trpsec_calc->enable_sloppy(); // allow division by zero
// set formula: (tr_cnt 'per seconds')
trpsec_calc->calc("per_second", tr_cnt, sc_time_stamp().to_seconds());
// create Statistics Calculator
10 overall_tr_per_sec = new
    gs::av::StatCalc<double>("overall_trans_per_sec_StC", trpsec_calc);
cout << "transactions per second parameter name: " <<
    overall_tr_per_sec->get_result_param()->getName() << endl;
```

- All analysis related code in the SimpleBus protocol is *guarded* by a `#ifdef GAV_ENABLED` which has to be enabled by the simulation including the GreenBus.
- The *new calculation function* `per_second` is added to the `Calculator<double>` class (line 2) with the access string "per_second".
- The current `sc_time_stamp` is used as a constant (using the `to_seconds()` function) (line 8).

Extrapolated transactions per second for each in and out port work the same. For each port one counter parameter and one StatCalc is created.

1.3.3 Dynamic Transactions per Second

See file  `greenbus/protocol/SimpleBus/simpleBusProtocol.h`

A class member static function is added with special functionality: A function-internal static vector stores old values for each protocol instance. The second function parameter is used to identify the protocol instance because the calculation itself only needs one input parameter – the transaction counter. The function calculates the transaction count between the current call and the last call to this function. This function should be called once per simulated second to deliver a transactions per second value that is *dynamically updated*.

```
1 static unsigned int transaction_count_diff(unsigned int
    transaction_count, unsigned int protocol_inst_no, bool sloppy) {
    static vector<unsigned int> old_values;
    if (old_values.size() < protocol_inst_no+1)
        old_values.resize(protocol_inst_no+1);
```



```
    unsigned int diff = (transaction_count - old_values[protocol_inst_no]);  
    old_values[protocol_inst_no] = transaction_count;  
    return diff;  
}
```

The protocol instance number is incremented with a static member function.

```
// Add user defined calculation function to Calculator class  
gs::av::Calculator<unsigned int>::addFunc      ↵  
    (&SimpleBusProtocol::transaction_count_diff, ↵  
     "private_transaction_count_diff");  
  
// Formula Calculator  
gs::av::Calculator<unsigned int> *dy_trpsec_calc  
    = new gs::av::Calculator<unsigned int> ("dynamic_tr_p_sec_overall");  
dy_trpsec_calc->  
    calc("private_transaction_count_diff", tr_cnt, getInstNumber());  
// Trigger  
gs::av::Trigger *dy_tr = new gs::av::Trigger(1, SC_SEC);  
// StatCalc  
dy_overall_tr_per_sec = new gs::av::StatCalc<unsigned int> ↵  
    ("dynamic_overall_trans_per_sec_StC", dy_tr, dy_trpsec_calc);
```

- The *trigger* (line 10) triggers the calculation each 1 SEC.
- The StatCalc gets the Calculator and the Trigger as a parameter (line 12).

The trigger has to be deleted in the destructor (together with the Calculator and StatCalc). The trigger must be deleted *after* the StatCalc object:

```
// do not delete the trigger before deleting the StatCalc!!  
delete dy_overall_tr_per_sec->get_calculator();  
gs::av::trigger_if* t = dy_overall_tr_per_sec->get_trigger();  
delete dy_overall_tr_per_sec;  
delete t;
```

1.3.4 Transaction Latency

See file  greenbus/api/simplebusApi.h (master port).

The SimpleBus master port provides a time measurement for the overall (and transport) transaction latency. This example shows the overall latency.

All analysis features are guarded with a `#ifdef GAV_ENABLED`.

The needed includes are:

```
#include "greencontrol/gcnf/apis/gs_param/gs_param.h"  
#include "greencontrol/gav/plugin/StatCalc.h"
```

Class member parameters are used to store the transaction start and end time points, which are used as input parameters to a StatCalc.

```

1 // Formula Calculator
gs::av::Calculator<double> *tr_overall_latency_calc
  = new gs::av::Calculator<double>("transaction_overall_latency");
tr_overall_latency_calc->calc("-", 2
  transaction_overall_latency_end_timepoint, 2
  transaction_latency_start_timepoint);
5 // Trigger
std::vector<gs::gs_param_base*>
  sensitive_params(1, &transaction_overall_latency_end_timepoint);
gs::av::Trigger *tr = new gs::av::Trigger(sensitive_params);
// StatCalc
10 tr_overall_latency = new gs::av::StatCalc<double>
  ("transaction_overall_latency_StatCalc", tr, tr_overall_latency_calc);

```

- The trigger takes changes of only one of the formula input parameters as a trigger event (“*manually chosen parameters*”) (line 8). The trigger parameters have to be given as a vector of parameter bases (lines 6-7).

- On each start point of a transaction
`transaction_latency_start_timepoint = sc_time_stamp();`
 is changed to the current time stamp. At this point *no* calculation is done because the trigger only triggers on changes of the end point parameter!

- On each end point of a transaction
`transaction_overall_latency_end_timepoint = sc_time_stamp();`
 is changed to the current time stamp. On this change the trigger triggers the calculation and the difference between the end and the start point is outputted as the result.

The destructor deletes the Calculator, the StatCalc and the Trigger:

```

1 delete tr_overall_latency->get_calculator();
gs::av::trigger_if* tr = tr_overall_latency->get_trigger();
delete tr_overall_latency;
delete tr;

```

The *transaction transport latency* works exactly the same with the difference that another end point parameter is used which is changed when the request of the transaction has finished. The same start point parameter is used!

1.3.5 Transaction Recording

See file  `greenbus/api/simplebusApi.h` (slave port).

The GreenBus Generic transactions can be recorded at the SimpleBus slave port. *Note* that this transaction recording is an example and insists not on well performing and beautiful implementation.

This include is needed. This is a specialization for the `gs_param` class which is specialized to the class `GenericTransactionCopy`. This copy class allows a transactions to be assigned to the copy class. The most important attributes will then be copied to this copy object. This is much overhead, not suitable for productive systems!

```
1 #include <
    "greencontrol/gcnf/apis/gs_param/gs_param_GenericTransactionCopy.h"
```

A class member parameter allows to enable or disable the recording:

```
1 gs::gs_param<bool> record_transactions_enabled;
```

This parameter may be set during construction to enable the recording for that port. (The default is disabled.)

A class member parameter

```
1 gs::gs_param<gs::cnf::GenericTransactionCopy> recorded_transaction;
```

is used as the parameter that can be outputted by the user to an *output plugin* like this (part of user code, e.g. testbench):

```
1 #include "greencontrol/gav/plugin/File_OutputPlugin.h"
// ** Analysis file-output for recorded transactions at Mem slave port
gs::av::OutputPlugin_if* record_opp
    = mainGAVApi->create_OutputPlugin(gs::av::FILE_OUTPUT_PLUGIN, &
        "recorded_transactions.log.txt");
5 // add recorded transaction to file output
gs::gs_param_base *recorded_mem_transaction = &
    mainApi->getPar("Mem.simplebus_slaveport.recorded_transaction");
mainGAVApi->add_to_output(record_opp, recorded_mem_transaction);
```

- The needed output plugin file has to be included.
- An *output plugin* is created (lines 1-2) using the specified file name to output a human-readable file.
- The output parameter of the slave port is added to this output plugin (line 6).

Additionally the slave port allows to record transactions only from a specific master. For this the class member parameter

```
1 gs::gs_param<long> record_transaction_master_id_filter;
```

can be set by the user, e.g. (part of user code, e.g. testbench):

```
1 // only record transactions from the master Cachel
gs::gs_param<long> *record_transaction_master_id_filter
```

```
    = dynamic_cast <gs::gs_param<long>*>(  
        mainApi->getPar  
5      ("Mem.simplebus_slaveport.record_transaction_master_id_filter"));  
*record_transaction_master_id_filter = 2  
    cachel.master_port.get_master_port_number();
```

The default is “0”. Then all transactions are recorded.

A transaction is recorded using this code within the slave port:

```
1 // Analysis: record write transaction  
if (record_transactions_enabled) {  
    if (record_transaction_master_id_filter == 0  
        || record_transaction_master_id_filter == ah->getMID())  
5    recorded_transaction = *(ah.get());  
}
```