**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**High-Level DSE Report**

Contract No. 4200022968

by

 TU Braunschweig, Germany

August 31, 2012

**Document No.: IDA-SCSV-DSE-001**

**Table of Contents**

1 Introduction 4

1.1 Purpose and Scope 4

1.2 Referenced Documents 4

1.3 Revisions 4

2 Application benchmark 5

2.1 Lossless multispectral and hyperspectral image compression 5

2.2 Software mapping 5

2.2.1 Porting of code to LEON target system 5

2.2.2 Porting of code to run with RTEMS OS 5

2.2.3 Parallelize code for multi-threaded execution 6

3 Exploration prototype 10

3.1 Leon3mp base platform 10

4 exploration methodology 11

4.1 Defining Parameters 11

4.2 Running Simulation 13

4.2.1 Single/Direct manual execution 13

4.2.2 Single/Makefile controlled execution 14

4.2.3 All(multi-thread)/Makefile controlled execution 14

4.3 Extracting Results 15

4.3.1 Log File Evaluation 15

4.3.2 Analysis API 15

5 Example design space exploration 16

5.1 Initial Configuration 16

5.2 Instruction Cache Variation 16

5.3 Data Cache Variation 16

5.4 Increasing the number of CPUs 17

5.5 Impact of arbitration scheme (AT) 17

5.6 Results summary 17

**Table of Figures**

Figure 1 - Activation of parallel threads 5

Figure 2 - Parallel data flow 6

Figure 3 - Base platform for design space exploration 9

Figure 4 - Spreadsheet for generating json configurations 11

**Table of Tables**

Table 1 - Referenced Documents 3

Table 2 - Revisions 3

# Introduction

## Purpose and Scope

This document is the High-Level DSE Report of the HW/SW SystemC Co-Simulation SoC Validation Platform (SoCRocket). Aim of the document is the demonstration of a design space exploration using models and infrastructure developed in the course of the project. This is supposed to be done by mapping an application benchmark to a Virtual Platforms Prototype. The design space will be explored by running several simulations with varying architectural parameters. Both, the best and the worst configuration shall be reported.

## Referenced Documents

|  |  |  |
| --- | --- | --- |
| **Reference** | **Document Number** | **Document Title, Author** |
| RD01 | IDA-SCSV-UM-001 | SoCRocket User Manual |
| RD02 | IDA-SCSV-DF-001 | SoCRocket Design Flow Report |
| RD03 | IDA-SCSV-ACR-001 | SoCRocket Analysis Capability Report |

Table 1 - Referenced Documents

## Revisions

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 0.1 | 21.08.2012 | High-Level DSE Report |
|  |  |  |
|  |  |  |

Table 2 – Revisions

# Application benchmark

## Lossless multispectral and hyperspectral image compression

As an application for design space exploration we have selected lossless multi-spectral and hyper-spectral image compression as standardized by the Consultative Committee for Space Data Systems (CCSDS) in Standard #123. The algorithm is an example of the applications executed on the payload processors aboard most scientific satellites; as such systems are often resource constraint, the use of SoCRocket enables achieving an optimal configuration of the system with high performance while limiting the overall system’s cost in terms of silicon-area occupation and power consumption.

As said above, the algorithm operates on hyperspectral images; these are three-dimensional data sets, where two of the dimensions are spatial and the third is spectral. A hyperspectral image can be regarded as a stack of individual images of the same spatial scene, with each such image representing the scene viewed in a narrow portion of the electromagnetic spectrum. Overall, the algorithm is based on adaptive linear predictive compression using the sign algorithm for filter adaptation, with local mean estimation and subtraction. The prediction residual is then encoded using a sample-adaptive Golomb-Rice encoder.

## Software mapping

The implementation of the algorithm in C-language has been provided by ESA. The code could not be used out-of-the-box, because it was not prepared for multi-threaded execution. The code itself is in large parts dominated by data movements and bit-level optimizations. Naturally, it would be much better suited for a dedicated hardware implementation. This needs to be kept in mind, especially when looking at the simulation results.

Nevertheless, the application makes an excellent demonstration case for design space exploration. It was decided to execute the code on top of RTEMS OS. The steps required for transforming the code into a flexible multi-processor implementation are explained in the following subsections.

### Porting of code to LEON target system

* Change build system to use Sparc Cross-Compiler
* Adapt Data-types

### Porting of code to run with RTEMS OS

* Create RTEMS task for single-processor execution
* Some adaptations to POSIX interface functions   
  (e.g. changes in header files Linux Posix -> RTEMS Posix)
* Setup RTEMS in-memory file system (IMFS) and align with application
* Implement a data type for global locking (lock\_t)
* Implemented a barrier for thread synchronization (barrier\_t)

### Parallelize code for multi-threaded execution

The code was partitioned into an initialization sequence, Nx producer tasks and 1x consumer task. The initialization sequence consists of following steps:

1. Unpack PROM to SDRAM
2. RTEMS boot and initialization
3. Import hyperspectral input file into RTEMS file system
4. **Calculate Residuals**
5. Image Compression

In the given implementation steps 1-3 are very time consuming. However, in practice they have to be executed only once (step1, 2) or can be largely optimized (step 3). A significant part of the code is dedicated to the computation of the Residuals. **This code has not been parallelized** and is considered a part of the initialization. In the following we only look at the actual image compression, which is done in step 5.

The example in Figure 1 illustrates the activation of the parallel threads. There are one consumer and N=3 producers. The consumer runs on CPU0, the producers run on CPU1, CPU2 and CPU3. At the beginning of the simulation the producers are blocked at a barrier until initialization is completed.



Figure 1 - Activation of parallel threads

Afterwards, the producers independently scan the input image (Figure 2) and copy the selected data to one of the available data blocks. The **data\_blocks** are allocated from a statically defined array of buffers. The order of the data blocks is predefined and has to be obeyed by the consumer. To avoid the producers from blocking, the number of data block buffers should be higher then the number of producers. In the given example 16 data block buffers are used for three producers.

The original software, as received from ESA, can be found in:  
**./software/rtems/ccsds123-mp-v1**

The modified multi-processor code:  
**./software/rtems/ccsds123-mp-v2**



Figure 2 - Parallel data flow

#### Initialization

The initialization sequence is implemented as a single RTEMS tasks. The task is only executed on processor 0 and initializes all global data structures of the system. Most of the initialization functionality is encapsulated in function encode (see file **entropy\_encoder.c**). This also includes the global **struct mp\_data**, which is used for controlling the distributed producer and consumer tasks.

The global control type **m\_data** is structured as follows:

**typedef struct mp\_data {**

**// Buffer for program command line parameters**

**input\_feature\_t input\_params;**

**// Buffer for encoder commands**

**encoder\_config\_t encoder\_params;  
 // The base address of the compressed output data stream.  
 // (Final results are going to be located here)**

**unsigned char \*compressed\_stream;**

**// The current byte offset to the output data stream  
 unsigned int \*written\_bytes;**

**// Additional bit offset to the output data stream and   
// written bytes**

**unsigned int \*written\_bits;**

**// Base address of input array with data ready for entropy // encoding  
// (data is output of prediction phase)**

**unsigned short \*residuals;**

**// Input parameter for entropy encoding**

**int block\_code\_option\_id;**

**// 16 data blocks reserved for producer output; to be**

**// merged/compressed**

**// by consumer thread(s).**

**block\_data\_t blocks[16];**

**// Lock for protection of block\_data (see below).**

**// Only one thread may enter at a time.**

**lock\_t lock;**

**// Barrier for synchronization of all threads (producers +**

**// consumer(s)):**

**// 1. All threads to wait until processor 0 has completed   
// initialization**

**// 2. All threads to synchronize at the end of the**

**// simulation (shut down)**

**barrier\_t barrier;**

**// Barrier for synchronizing producers; all producers have**

**// to be completed // before consumer shut down.**

**barrier\_t producer;**

**// Signal for Consumers that all Producers have completed**

**volatile int run;**

**// Largest block-number currently in computation (by the**

**// producer)**

**volatile unsigned int number;**

**// Number of data blocks already processed by consumer(s)**

**volatile unsigned int consumed;**

**// Lock for number and consumed; may only be incremented**

**// by one processor at a time**

**lock\_t number\_lock;**

**// Number of active threads**

**int procs;**

**} mp\_data\_t;**

The **mp\_data** **struct** is intended for global, distributed control and synchronization. Additionally, we introduced local descriptors for the block data buffers shown in Figure 2.

**typedef struct block\_data {**

**// ID of the thread/cpu which currently owns this block  
 // (or -1 for data block completed (ready for consumer),  
 // (or -2 data block is not allocated (can be claimed by**

**// producer))**

**volatile int proc;**

**// Pointer to base address of data block (the address the // producer is ought // to dump its data).  
unsigned short int \*samples;  
// Data extension array for dedicated compression method   
// (-1)**

**unsigned int second\_extension\_values[32];**

**// The producer selects the optimal compression method for // the block.  
// To be stored here:  
int chosenMethod;**

**// Zero block indicator (this is a zero block)  
unsigned int num\_zero\_blocks;   
// This is the last data block of the file. Eventually has // to be padded with  
// zeros.  
int segment\_end;  
// Global, unique id of the data block / needed for   
// reordering**

**volatile int number;**

**} block\_data\_t;**

#### Worker task (Producer)

After system boot up, all producers wait at the **mp\_data::barrier** for the initialization sequence to complete. Then, computation starts by acquiring a vacant data block. Free data blocks can be found by checking **block\_data::proc** (-2 meaning free). Acquisition is done by overwriting **block\_data::proc** with the own thread ID. The producers then independently execute the code encapsulated in the **encode\_block** function (file **entropy\_block.c**). The encoded data is copied to **block\_data::samples**. Afterwards, the producer selects the optimum compression method for the block and stores it in **block\_data::chosenMethod**. Finally, the block is marked completed (**block\_data::proc = -1**). This procedure is repeated until no input data available anymore.

#### Consumer

The consumer is responsible for collecting and assembling the data blocks generated by the producers. The code is encapsulated in function **block\_compressor** (see file **entropy\_encoder.c**). The way the final/compressed image is build up depends on the selected compression methods. The compression method is selected by the producer and stored in **block\_data::chosenMethod**. A different method may be selected for each block. While collecting data blocks the consumer has to obey a strict order. If one particular block is missing, because e.g. the responsible CPU is overdue, it has to wait. The producers on the other hand are allowed to run ahead until the pool of allocated blocks is exhausted. The consumer recognized completed blocks by checking the **block\_data::proc field** (-1 meaning completed). The global sequence number is stored in **block\_data::number**.

# Exploration prototype

## Leon3mp base platform

The base platform for the design space exploration described in this document is the leon3mp system, which can be found in the repository at following location:

**./platforms/leon3mp**

Figure 1 shows the simplified structure of the system. The **leon3mp** VP consists of **N** processors, which are connected to an AMBA High-Performance Bus (AHB). Each of the processors has it’s own private instruction cache and data cache. The processors can also be configured to enclose memory management units as well as data **localrams** and instruction **localrams**.

The shared **AHB bus** connects the CPUs to a memory controller. The memory controller can be configured for four different types of memory: **IO**, **PROM**, **SRAM** and **SDRAM**. The default configuration provides **PROM** + **SDRAM**.

The **APB-bridge** connects the configuration registers of the **MCTRL**, the multi-processor interrupt controller and the general purpose timer. Other components of the system are the **AHBMEM**, which is a simple **SRAM** device, the **APBUART** IP\*, the **SoCWire**\* bridge and the SpaceWire\* bridge.

\* not in picture



Figure 3 - Base platform for design space exploration

# exploration methodology

Finding the optimal configuration of a design means defining the best set of configuration parameters in terms of performance, power consumption and area occupation. The number of parameter-sets under exploration depends on the system requirements/constraints and the selected exploration starting point. The SoCRocket platform provides all matters for generating parameter-sets and executing simulations in a semi-automatic way. The implemented tools theoretically enable an exhaustive search of the design space covering all configuration parameters. Although for realistic applications with considerable runtime this is hardly practical.

The recommended approach is to start with an educated guess for a sane initial configuration. Usually, the results from the first simulation runs radically narrow the solution space. Compared to exhaustive search, directed and controlled exploration leads to equally good results, but only requires a fraction of the time.

## Defining Parameters

As described in the SoCRocket Design Flow document system configurations are represented by **json** files. Each of these files contains a hierarchical description of all configuration parameters and their initial values. The example below shows a parameter definition for the **ahbctrl IP**, as used in the **leon3mp** platform:

**"conf": {**

**"ahbctrl": {**

**"cfgaddr": 4080,**

**"cfgmask": 4080,**

**"defmast": 0,**

**"fixbrst": false,**

**"fpnpen": true,**

**"ioaddr": 4095,**

**"ioen": true,**

**"iomask": 4095,**

**"mcheck": true,**

**"rrobin": false,**

**"split": false**

**}**

The hierarchical name of e.g. the **rrobin** parameter is **conf.ahbctrl.rrobin**. The **leon3mp** platform comes with a LUA script (**config.lua**), which parses the configuration at the beginning of the simulation and initializes all configuration parameters.

To ease the handling of json configuration files two scripts are provided in the tools directory, which may be used to set/get names of attributes or values. Use as follows:

1. For obtaining the value of a certain parameter:  
   **./tools/get\_json\_attr leon3mp.json conf.ahbctrl.rrobin**
2. For getting a list of all parameter/value pair for a certain IP (here ahbctrl):  
   **./tools/get\_json\_attr leon3mp.json conf.ahbctrl**
3. For setting a parameter to a new value:  
   **./tools/set\_json\_attr leon3mp.json conf.ahbctrl.rrobin=true**

It is also possible to set multiple parameters at once. The new configuration file will be written to **stdout**.

A JSON configuration file can be written manually or generated using the SoCRocket Configuration Wizard. Additionally, a script is provided for generating groups of configurations from Comma-Separated-Values Files (CSV). Such files can be written with e.g. Microsoft Excel. The example in Figure 4 contains four configurations – one per line. The name of the configuration must be given in column A. All other columns represent parameters. The name of the parameter is expected in the first row (header of table). It must be equivalent to the name of the parameter in the platform (e.g. **conf.mctrl.prom.elf** for application binary).

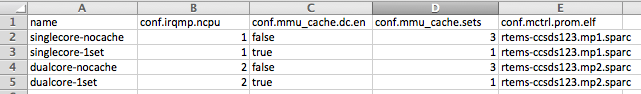


Figure 4 - Spreadsheet for generating json configurations

To generate json configuration files for each of the configurations in the spreadsheet use the following command:

**./tools/csv2json leon3mp.json configurations.csv**

The default configuration file of the **leon3mp** platform (**leon3mp.json**) is used as a template. All parameters that are found in the CSV file will be assigned to new values.

## Running Simulation

All configurations that shall be explored in one step should be located in the benchmark SW directory. For the example of the hyperspectral compressor this path is:

**./software/rtems/ccsds123-mp-v2**

The directory contains a Makefile for controlling the exploration. The Makefile compiles the application code, applies target specific settings and starts the simulation(s).

Software can be compiled with following command:

**make targets**

Boot-code will be generated and linked automatically. The application is not completely platform independent. Especially, the number of CPUs needs to be known at linking time (parameter expected by **mkprom**). Therefore, this step will generate multiple binaries for systems with one, two, four and six CPUs.

If the software is compiled and the JSON configuration file(s) are available, the simulation can be started. This can be done in multiple possible ways.

### Single/Direct manual execution

The platform executable is location in following directory:

**./build/platforms/leon3mp/**

The simulation should be started from the benchmark SW directory (using relative paths). The **-j** parameter allows to hand-over a configuration file:

**leon3mp.platform -j config.json**

The **--help** option displays the following message on the screen:

**SystemC 2.2.0 --- Mar 23 2012 15:15:32**

**Copyright (c) 1996-2006 by all Contributors**

**ALL RIGHTS RESERVED**

**SoCRocket -- LEON3 Multi-Processor Platform**

**Usage: ./build/platforms/leon3mp/leon3mp.platform [options]**

**Options:**

**--help Shows this message.**

**-s [ --luascript ] arg The LUA configuration script. Usual the json.lua to**

**load a JSON configuration.**

**-j [ --jsonconfig ] arg The main configuration file. Usual config.json.**

**-o [ --option ] arg Additional configuration options.**

**-a [ --argument ] arg Arguments to the software running inside the simulation.**

**-l [ --listoptions ] Show a list of all available options**

If the simulation is not started from the benchmark SW directory the paths to the LUA parameter annotation script needs to be defined using the **-s** or **--luascript** option.

If one of the scripts (LUA, JSON) can not be found at the specified location and is not available in the current directory, it is searched in **./platforms/leon3mp**. Afterwards, the tool checks the directory of the platform executable **./build/platforms/leon3mp**. If this also fails environment variables are checked (LUASCRIPT, JSONCONFIG).

The **-o** option allows the overloading of configuration parameters. This is very handy for small explorations. It avoids having to setup a new JSON configuration script in case only one or a few parameters change value.

The application is executed on top of RTEMS OS. RTEMS requires at least one UART interface to be in the system. Therefore, the simulator will stop after starting the simulation and ask for a terminal to be connected.

**UART: waiting for connections on port 2000**

Connect a terminal to the UART using **telnet** or **netcat**.

### Single/Makefile controlled execution

In the benchmark SW directory call:

**make {CFG\_NAME}.sim.log**

(**CFG\_NAME** – name of json configuration: **{CFG\_NAME}.json**)

The simulation starts automatically using the application file name specified in the JSON file. The Makefile takes care of connecting the terminal for the first UART in the system. In case there are multiple UARTs configured, the Makefile must be changed.

### All(multi-thread)/Makefile controlled execution

In the benchmark SW directory call:

**make tests**

This will startup simulations for all configurations (JSON files) in the working directory. The

**-j** switch may be used to specify the number of parallel threads (simulations executed in parallel).

## Extracting Results

### Log File Evaluation

Simulation results can be extracted in many different ways. The most common one is to evaluate simulation log files.

All simulations create log files named **{CFG\_NAME}.sim.log** in the working directory.

(**CFG\_NAME** – name of json configuration: **{CFG\_NAME}.json**)

At verbosity level info or above (see SoCRocket User Manual) all models produce a report with execution statistics at the end of the simulation. The reports are based on internal performance counters. The content of the report is model-dependent. Caches e.g. report hit-rates and miss-rates, busses report the number of bytes transferred. A detailed list of all performance counters and reports can be found in the SoCRocket Analysis Capability Report.

In addition to performance figures, a power report can be generated, by enabling the build-in power monitor. This can be done by setting **conf.report.power=true** (in the JSON config file). The monitor reports average power consumption over the whole runtime of the simulation. Information on how to annotate technology specific power information and to build custom power monitoring tools is given in the SoCRocket Power Modeling Report.

All reports are self-explanatory and can be parsed easily. For this exploration a set of simple bash scripts was used. The scripts are not part of the distribution.

### Analysis API

All performance counters in the system can be accessed at any time during simulation using the build-in analysis API.

The analysis API allows listing parameters, to read/write/display parameters and to log parameters. Furthermore, it is possible to create waveforms for parameters and to create callback functions for custom analysis tools.

A detailed description on how to use the Analysis API can be found in chapter 2 of the SoCRocket Analysis Capability Report.

# Example design space exploration

## Initial Configuration

A dual-processor configuration with small 2-set associative instruction and data caches was selected as a starting point for design space exploration. The input image is restricted to 256 rows, 256 cols and 3 bands.

**Configuration Summary (initialconfig):**

|  |  |
| --- | --- |
| Number of CPUs | 2 |
| Number of Instruction Cache Sets | 2 |
| Size per Instruction Cache Set | 1kB |
| Number of Data Cache Sets | 2 |
| Size per Data Cache Set | 1kB |
| Abstraction Level | LT |

**Results Summary:**

Simulation results without initialization and calculation of residuals (see 2.2.3).

*Benchmark Performance:*

2263804660 ns (2.263 s)

*Simulator Performance:*

200860000 us (3:35 min)

*Power Consumption:*

Static power (leakage): 1302.62 uW

Internal power (dynamic): 2468.36 uW

Switching power (dynamic): 4048.49 uW

-------------------------------------------------------

Total power: 5351.10 uW

**Conclusions:**

The LEON processor cores dominate the power consumption (Core 0 = 1,4 mW, Core 1 = 4,7 mW). Compared to these numbers the power of the caches is very low (539 uW). Hence, it seems reasonable to increase the size of the caches for improving the performance.

## Instruction Cache Variation

## Data Cache Variation

## Increasing the number of CPUs

## Impact of arbitration scheme (AT)

## Results summary