**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**Power Modeling Report**

Contract No. 4200022968

by

 TU Braunschweig, Germany

March 8, 2012

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

**Table of Contents**

1 Introduction 2

1.1 Purpose and Scope 2

1.2 Revisions 2

**Table of Figures**

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**  
Wählen Sie im Dokument die Wörter aus, die in das Inhaltsverzeichnis eingeschlossen werden sollen, und klicken Sie dann in der Formatierungspalette unter "Formatvorlagen" auf ein Überschriftenformat. Wiederholen Sie dies für alle einzuschließenden Überschriften, und fügen Sie das Inhaltsverzeichnis dann in das Dokument ein. Sie können ein Inhaltsverzeichnis auch erstellen, indem Sie auf die Option "Manuelle Formatierung" klicken und die Einträge dann manuell eingeben.

**Table of Tables**

Table 1 - Revisions 2

# Introduction

## Purpose and Scope

The purpose of this document is to decribe concept and parameters of the Power Modeling techniques utilized within the HW/SW SystemC Co-Simulation SoC Validation Platform (SoCRocket).

The most prominent use cases of virtual platforms are software development, architecture exploration and system verification. Depending on the field of application the users are interested in high simulation speed, utilization and performance figures, or high accuracy. Nowadays, these use cases get more and more accompanied by the wish to estimate power consumption as early as possible and as high up as possible in the design flow. Especially modern embedded multi-processors have to consider power constraints. Particularly systems in spacecrafts need to operate on given power budgets and are restricted to a maximum peak current. Hence, software for such systems should be written with power demands in mind.

To enable trading off power consumption against performance and latency, the SoCRocket library provides an event based power monitoring concept.

## Revisions

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 0.1 | 08/03/12 | Initial draft for MDR/DFR meeting |
|  |  |  |
|  |  |  |

Table – Revisions

## SoCRocket power monitoring

In the SoCRocket library power monitoring is based on power events (PE), which are emitted by the simulation models and recorded in a database (). This database is called the Power Monitor (PM).

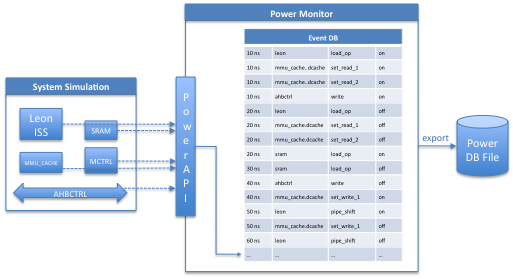


Figure - Power event monitoring

The PM is implemented as a static C++ class, which provides an API for all simulation models in the library. Most SoCRocket simulation models already implement this API. In that case, power monitoring can be enabled by setting the **pow\_mon** constructor parameter.

The example in illustrates the recording of a synthetic sequence of events. The events are emitted by multiple simulation IPs such as the **LEON ISS**, the **MMU\_CACHE**, the **AHBCTRL** and the **GenericMemory** (SRAM). All events of the example relate to dynamic power events. The IPs submit there static power consumption at the begin of the simulation or at changes of the operation mode (e.g. MCTRL low power modes).

It can be seen that after 10 ns simulation time the LEON ISS triggers the **load\_op** event, which indicates a load operation. The cache receives the load operation at the same time-stamp and performs a lookup of both cache sets (**set\_read\_{1,2}**). The operation is instantly forwarded to the **ahbctrl**, because it turned out a read miss (and the **mmu** is disabled). Consequently, also at 10 ns, the **ahbctrl** triggers the **write** event, which is also recorded by the PM. The next time step (20 ns) disables the power events from **leon** and **dcache**. In the meanwhile the load has reached the **SRAM** memory and invokes a **sram.read** event. Another 10 ns later this event is deactivated (30 ns). The transaction response reaches the master at 40 ns (deactivation of **ahbctrl.write**). In the following the **dcache** is updated with the new data (**mmu\_cache.dcache.set\_write**). The operation finishes with a pipeline shift in the LEON ISS (**pipe\_shift**).

After the end of the simulation the data recorded in the event DB may be instantly analyzed by calling the Power Analyzer (PA) or be exported to a file. The PA performs a set of sanity checks on the collected data. The checks make sure that every event in the data base has been properly disabled and appears in the right order. The primary tasks of the PM are the assignment of power values to events and the interpolation of the data (Figure 2).

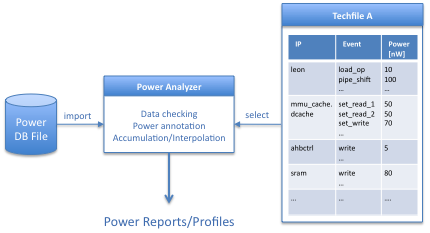


Figure - Power Analyzer

Given the switch-on and switch-off times of the events and the power per event information from the technology file (Techfile), the calculation of power profiles is relatively straight forward. Beginning from simulation time zero the power analyzer works through the database accumulating the overlapping power events at every instance of time. For the given example (, ) the results are calculated as follows:

|  |  |  |
| --- | --- | --- |
| Time Step | Active Events (to accumulate) | Actual Power (Techfile A) |
| 10 ns – 20 ns | leon\_op + set\_read\_1 + set\_read\_2 + ahb\_write | 115 mW |
| 20 ns – 30 ns | ahb\_write + sram | 85 mW |
| 30 ns – 40 ns | ahb\_write | 5 mW |
| 40 ns – 50 ns | set\_write | 70 mW |
| 60 ns - 70 ns | pipe\_shift | 100 mW |

Table - Example: PA accumulation/interpolation of data

IPs that support power monitoring include the **power\_monitor.h** file. They register with the PM using an unique name. This is done in the constructor of the top-level class:

**PM::registerIP(this, “mctrl”, pow\_mon);**

Next to the name of the device, **registerIP** receives a module-pointer and a boolean switch as input arguments. The pointer is used to identify the module in the class hierarchy of the simulation. This also allows to monitor multiple instances of the same device. If the boolean (**pow\_mon**) is false, the body of the function is removed at compile time. If the model has multiple sub-components with relevant contribution to the overall power consumption, each of them can be registered as a separate IP. The MMU\_CACHE IP, for instance, registers their caches, localrams and mmu depending on the configurations.

The overall power consumption of a device consists of a static and a dynamic portion. The static power (leakage) is transmitted using the **send\_idle** function:

**PM::send\_idle(this, “idle”, sc\_time\_stamp(), pow\_mon);**

In the most trivial case the static power of a component is constant during the complete simulation time. This accounts for models such as the GPTimer or the IRQMP. Other IPs (e.g. MCTRL) provide multiple modes of operation, which have direct impact on the static power consumption. Therefore, the **send\_idle** function receives an event name (power mode) and a time stamp as input parameters. For each call to **send\_idle** both parameters are entered in the internal data base. In case **pow\_mon** is disabled (false), the body of the function will be removed by the compiler.

Dynamic power occurs if the internal state of a device changes. This happens for all sorts of events. However, the major share of the dynamic power is usually contributed by embedded register banks and memories. The costs for reading and writing these devices is known from the respective datasheets. Dynamic power events can be switched on and off using the **send** function. The example relates to a read operation from a cache set:

**PM::send(this, “set\_read0”, sc\_time\_stamp(), 0, pow\_mon);**

Similar to static power modes each dynamic power event is identified by a unique name. Other input parameters are the current timestamp, the on/off identifier and the enable switch (**pow\_mon**). It is important that every dynamic event that is switched on is also switched off again. The timestamp of the ‘switch-off’ operation must be higher/later then the timestamp of the ‘switch-on’ operation. In case of a mismatch the **analyze** function of the PM will issue a warning.

At simulation runtime the PM simply records all incoming power events. In order to minimize the impact on the simulation performance all analysis work is done offline. The analysis engine verifies the recorded raw data by running various consistency checks. It also sorts all power data with respect to their timestamp and hierarchically accumulates the information. In the last step all power events are annotated using technology dependent power information. An example for a generic power-tech file is provided with **./models/main\_power.dat**. The structure of this file is self-explanatory. It is grouped in multiple sections describing the IPs of the library. Each IP is identified by a name and provides a list of power events – value pairs:

**!IP!**

**name**

**idle power value  
action 1 power value  
action 2 power value  
action 3 power value**

**!ENDIP!**

The power analyzer can be called from the top-level of the simulation, right after **sc\_stop():**

**// techpath – Path to technology file  
// techfile – The technology specific power file  
// outfolder - Name of the output folder (for results)**

**PM::analyze(string techpath, string techfile, string outfolder);**

Alternatively, the power analyzer can be started in stand alone mode:

**// rawfolder - Path to raw-data  
// rawfile - Name of the raw-data file**

**PM::analyze\_offline(string techpath, string techfile,   
string outfolder, string rawfolder, string rawfile);**

In case the specified **techfile** can not be found the PM checks the POWERMONITORDAT environment variable. If this also fails, the PM generates an error message.

The output of the power analyzer is stored to the **logfiles** sub-directory of the path specified in **outfolder**. The sub-directory plotfiles contains data files and scripts for post-processing with GNUPLOT. To create plots for the power consumption of a certain IP or group of IPs use the following command:

**./plot outfolder/main.IP.gnu**

An example for a power profile plot (data from ) is shown in Figure 3.

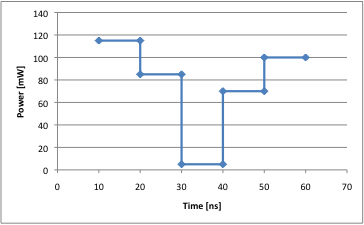


Figure - Example Power Profile

Depending on the extend and length of a simulation, recording power information can generate huge data files. Respectively, the computation effort for the analyze function may become very high. To reduce computation time it is recommended to limit the observation window of the PM to a meaningful timespan. This can be done using the **limit\_region** function:

**PM::limit\_region(sc\_time start, sc\_time end)**

The events recorded during simulation runtime can be annotated with technology dependent power numbers.

# Overview Power Events

# Synthesis Results / Parameter Fitting