# A Hardware/Software Codesign Methodology for Power-Aware Smart Cards[*]

U. Neffe, K. Rothbart, Ch. Steger, R. Weiss
Graz University of Technology
Institute for Technical Informatics
Inffeldgasse 16/1, 8010 Graz, AUSTRIA
{neffe,rothbart,steger,weiss}@iti.tugraz.at

E. Rieger, A. Muehlberger
Philips Semiconductors
Business Line Identification
Mikronweg 1, 8101 Gratkorn, AUSTRIA
andreas.muehlberger@philips.com

*Abstract – Power and energy consumption is an essential design constraint for passive embedded mobile devices. These devices, e.g. smart cards, do not contain an integrated power supply and often provide only limited resources. Such devices can be designed with traditional methods due to their low complexity, but integrated HW/SW co-design methodologies enable the gain of system-level optimization. This paper presents the abstraction of smart card designs to optimize system architecture and memory system. Functiona-level, transactional-level, and cycle-accurate models are presented and discussed. The proposed design flow and results of the evaluation are depicted.*

**Keywords:** Smart Card, HW/SW Co-design, Low Power, Transaction-level, Java Card

## 1.0 Introduction

Power and energy consumption has been an important design constraint for embedded devices for more than one decade. Embedded systems powered by batteries are designed for minimized energy dissipation to increase stand-by and active times. Passive mobile devices, for instance contact-less smart cards, are often powered by some sort of RF field with constrained field energy. This field does not limit energy dissipation but power consumption. Smart cards embedded into mobile GSM phones (*Subscriber Identification Module* (SIM) cards) have to fulfill both constraints because the mobile phone is powered by battery and smart card supply currents are limited by international standards. To run complex applications on passive mobile devices low-power hardware design is not sufficient. HW/SW co-design methodologies supporting power aware techniques and low-power optimizations on all levels of abstractions are necessary to reach design constraints for power with regard to performance and chip size. This paper presents a methodology for the design of passive mobile devices with limited resources. The remaining paper is organized as follows. Section 2 describes the smart card system. Section 3 discusses the abstraction of smart cards and models. The used design flow is presented in Sect. 4 and the evaluation of this methodology in Sect. 5. Section 6 concludes.

## 2.0 Smart Card Systems

A smart card [15] has no integrated power supply and is therefore a passive device. Passive devices always need an active device to get their power. Hence, a smart card system comprises not only the smart card but also a host terminal and optional a background system. Such a system is presented in Fig. 1. The smart card itself is presented by its hardware, the operating system running on the card and the applications. The link between the card and the smart card reader depends on the interface type. Contact cards are directly connected to a reader via a serial link which contains also two lines for power supply. A contact-less smart card reader generates an RF-field for communication. The card uses this field as the power supply and for communication.

[*] This work has been supported by the Austrian „Bundesministerium für Verkehr, Innovation und Technologie“ under grant number 806014 (Methodologies for Designing Power Aware Smart-Card Systems, FIT-IT Program).

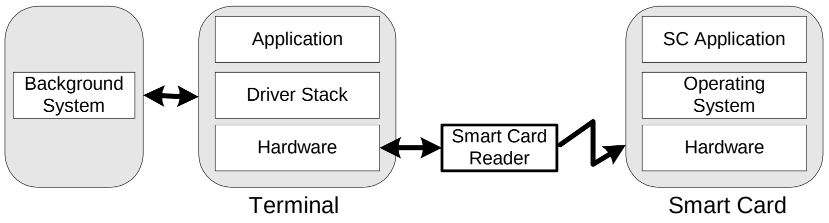


Figure 1: Smart card system with smart card, smart card reader, terminal and background system.

The reader is connected to the terminal via a standard interface. This can be a serial RS-232 connection or may be an USB interface. The terminal contains the interface hardware, the smart card reader driver and terminal applications. Dependent on the driver, a terminal can manage one or more smart card reader independent of the smart card I/O interface. The terminal application can communicate with a background system.

As discussed before, a smart card is a passive component in terms of power but also a passive component due to its behavior. During normal mode of operation a smart card only reacts on commands sent to the card by the host. Hence, a serial universal asynchrony receiver/transmitter (UART) is an essential component of a card. This UART is used to receive and send messages. Additional to the serial interface data have to be stored permanently on the card. Different non-volatile memory technologies such as EEPROM and Flash are used. Smart cards are often used to store confidential information. Thus cryptographic algorithms and secure keys are used to protect confidential data and data transmission. But cryptographic algorithms require a high amount of computation power. This power is mostly provided by dedicated coprocessors. Only high performance 32-bit smart cards can compute them in software.

## 3.0 System Abstraction

This section describes the different levels of abstraction and corresponding system models used in this approach. The reason for system abstraction is the high potential for power and performance optimization capabilities proposed by previous publications. Raghunathan *et.al.* [3] denoted the gain at system-level to a factor of five to ten and at behavioral level a gain of two to five. The following subsections discuss models at different levels of abstractions and their system boundaries.

### 3.1 Functional System-level Models

In general, functional system-level models depend on the type of application. Applications have to be classified as data-dominant, control-dominant, or a combination of them. According to this classification and the model of computation the modeling language and tools can be selected [5]. To determine these aspects the basic behavior has to be analyzed and implementation details have to be abstracted. Because of the focus of this work on power consumption and performance system components with the highest impact on these design parameters have to be identified and represented in a model.

At highest level of abstraction a typical smart card application can be seen as a data flow model, which is presented in Fig. 2. A message is received by the I/O interface and forwarded to the communication protocol analyzer. The protocol analyzer forwards relevant parts of the message to the decryption unit and the control unit processes the message and stores relevant data. But the terminal can also request some data items which have to be forwarded from the control unit to the encryption unit and further to the protocol and I/O unit. Such a model focuses only on the functionality and thus is only useful in a system simulation comprising the background system, terminal and smart card. The data transfer volume on the serial link and the resulting delay times can be determined and required computation power for message de/encryption and necessary data memory utilization can be estimated. UML can be used to model the behavior at this level of abstraction and Java, Matlab/Simulink, C++, or any other appropriate language can be used to implement it. This level of abstraction is appropriate for application designers to get a behavioral, executable model. The drawback of this model is the large gap between the modeling style and the final implementation. Several frameworks were presented for this level of abstraction [5].

In smart card development, optimized code is required due to the strictly limited resources, which is written by specialists. Object oriented design was chosen for modeling.

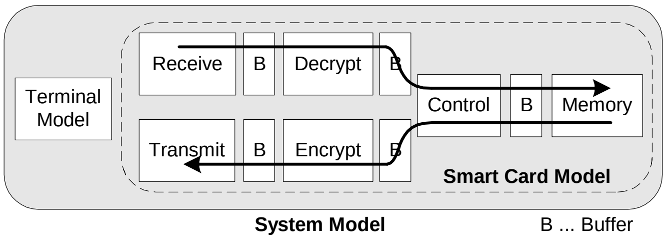


Figure 2: Functional smart card data-flow model embedded into a system model in a homogeneous environment.

C is the main programming language for smart card micro-controllers and therefore C++ was chosen as implementation language to avoid rewriting code for the target processor. This decision allows the smart card programmers to write their optimized code with the final programming language. Parallel processes and system-level modeling constructs are provided by SystemC™ [9] which is used as simulation engine.

Smart card functionality is divided into communicating SystemC modules. Two different types of modules are supported: (i) Application Program Interface (API) modules, and (ii) user-defined modules. API modules provide basic smart card functionality with a programming interface at operating-system level. The main API modules are:

- Different memory technologies (RAM, EEPROM, Flash)
- Serial interface to send and receive messages
- Timer with different modes of operation
- Cryptographic algorithms

API modules are implemented in C++ and their programming interfaces are defined by C++ interfaces. All API modules implement their functional interface and an energy control interface. The implementation of the energy interface depends on the modules' functionality and the energy model.

User defined modules implement the business logic and access API modules using SystemC ports. They also communicate among each other over SystemC interfaces. Due to the missing energy and performance pre-characterization of user-defined modules only estimated execution delay times are supported which have to be provided by the designer. The system boundary is the communication interface (UART) between the smart card and the reader, which can be realized by standard C++ communication channels. Figure 3 shows an example for a functional model. The active process is defined by the module *System Control* and accesses the UART to receive and send messages. Module *M1* only accesses an EEPROM module. In real applications M1 can represent non-volatile memory management. An *Energy Sample Unit* accesses the energy interfaces of the API modules to sample the energy values and to calculate the total energy dissipation.

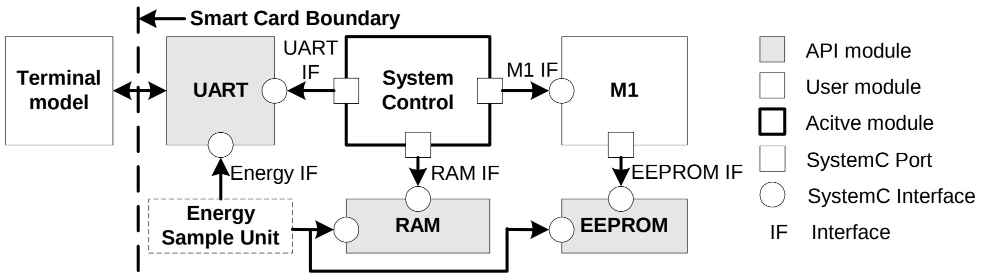


Figure 3: Example for a functional smart card model using API and user-defined modules.

As discussed above, optimization at this level of abstraction should have the highest impact on power and performance. Several techniques have been developed to support system optimization. To reveal more information about power and performance additional analysis units are integrated, which are called *interface adapters* (IA). IAs are positioned between two modules and trace the actual parameters of a method call and delay one simulation cycle. The delay is necessary to get the right order of function calls due to the un-timed, sequential execution. Based on the traces memory access patterns and data transfer statistics between modules can be determined. Based on this information memory locality can be increased and algorithmic transformations can be used to reduce the number of memory accesses.

Also coupling between modules can be calculated based on the traces of function calls. Coupling can be increased or decreased by moving functionality between modules. Strong coupled modules are defined as modules with high intercommunication bandwidth. Weak coupled modules implemented on different processing elements do not stress the communication system as much as strong coupled modules. Tracing of actual method parameters is necessary to determine the data behavior. Important are average values, standard deviation, and minimum and maximum values. The minimum bit number of variables can be calculated to reduce operand size based on quality of service constraints.

## 3.2 Transaction-level Models

The functional models proposed above describe memory organization and module intercommunication in detail. But this model is independent of the underlying hardware architecture. Therefore transaction-level architecture models are used for an efficient design space exploration. SystemC and SpecC [10] introduced programming constructs for transaction-level communication modeling but no design methodologies. This subsection presents the way of modeling and exploration of architectures, the mapping of functional models on architectures, and power/performance optimization techniques.

Architectural models exist of two types of components: (i) processing elements (PE) and (ii) communication units. Processing elements are containers for functionality which abstract processors, co-processors, and all other types of processing element. A basic processing element provides only a configurable bus interface to the functional blocks. Specific models can provide an abstract view of the programming model of a processor. This can be an interrupt system, additional I/O ports, or important parts of the memory system. Due to the encapsulation of the functionality within the processing elements no cross-compilation of code can be performed and therefore no representation of program code memory or instruction paths is available.

Communication units represent all types of communication structures used in embedded systems. They can be single signals or transaction-level representations of bus systems. The presented approach supports the *open core protocol international partnership* [1] protocol (ocp-ip) and uses it for communication structure exploration. The ocp-ip is defined for point-to-point connections, which always requires some sort of bus controller. The provided ocp-ip models do not provide any energy estimation, but bus models for smart card processors with energy estimation features have been developed [12]. These models have an accuracy of around 10%.

Figure 4(a) depicts an example for a transaction-level architecture. It contains three processing elements and five communication units (3 ocp-ip connections, one signal and one bus controller). PE2 can set an interrupt signal which triggers an interrupt process of PE1. This is a simple example for a more specific processing element with interrupt system. These architectures do not contain any functionality. Functionality is inserted by mapping of a functional model to the architecture. The mapping can be done with different granularities. Coarse grained mapping is done by mapping entire functional modules. Fine grained mapping can be done at method level, where each method is mapped on an own processing element.

Interface synthesis is necessary to implement the mapping. If the designer decides to implement both modules on the same PE, the interface has to be replaced by software function calls or otherwise by a hardware interface. If they are mapped on different processing elements interface calls have to be propagated over the communication system to the target processing element. Therefore, master module adapters (MMA) are necessary to connect a functional module port to the bus port of a PE. At the slave side a slave module adapter (SMA) is necessary to handle bus requests and forward function calls. Figure 4(b) presents a functional module structure where M1 calls a function of M2. Figure 4(c) shows the mapped solution whereby M1 is mapped to PE1 and M2 is mapped to PE2.

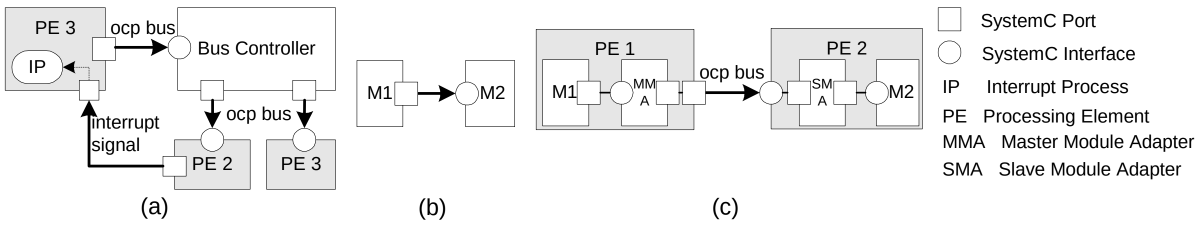


Figure 4: Mapping process: (a) Example of a transaction-level architecture model; (b) functional model; (c) functional model mapped on the processing elements PE1 and PE3.

MMA transmits all function parameters word by word to the SMA which collects all data and performs the function call to M2 after a control byte is sent to SMA. The MMA has to reveal the result from SMA after the function call was performed successfully which is indicated by a status bit in the SMA.

Energy characterization of API modules is performed in the same way as at functional level. Due to the availability of hardware components and software solutions for different processor platforms, performance, energy and code size for a target processor can be estimated with high accuracy.

Architecture selection and optimization have a high potential to increase performance and decrease energy dissipation with regard to chip size. Several optimization techniques were developed by other authors to optimize bus systems and memory hierarchies [2] at register-transfer-level. The presented methodology has been developed to explore these techniques fast and automatically. As presented above each functional model can instantiate and control its own memory blocks which leads to several small blocks of the same memory technology. These blocks have to be merged to larger blocks. An optimizer calculates the best TL memory architecture based on memory access patterns generated by IAs. Fine tuning of the memory map and memory access patterns is supported by transaction-level trace adapters similar to IAs.

The optimal bus encoding can be calculated based on the memory traces. Tools have been developed to determine the energy dissipation for binary bus encoding, T0 code [7] and bus-invert coding [8]. This allows a fast exploration of different bus encodings for a given application without implementing them at RTL. Based on these traces also memory mapped control registers of peripherals can be optimized. The number of registers to control peripherals, the position of control and data-bits within a register and their relative addresses can be optimized to reduce switching activity on the system bus.

## 3.3 Cycle-Accurate Platform

The most accurate supported level of abstraction is the cycle-accurate platform. Commercial hardware estimators are used to estimate application specific hardware energy consumption. This section describes the developed estimation techniques for transaction-level system bus and software energy estimation.

### *Transaction-level layer 1 and layer 2 bus models*

Bus models have a high impact on the total energy dissipation. Therefore accurate estimation techniques are needed. As discussed before, switching activity is an indicator for energy dissipation. In top-down design no information about the final implementation is available and therefore only indicators can be used and optimization is performed based on the indicators. But if an existing platform is used to implement a design the parameters (e.g. parasitics) of this design can be used for bus characterization.

With every new level of abstraction details are lost and the effects of abstraction are unknown. Thus this section describes the abstraction of a dedicated bus system and discusses the effects of abstraction [70]. The design characterization has been done with a gate-level energy estimation tool which also considers parasitics back annotated from the layout. The considered bus model is based on a 32-bit unidirectional bus with separated address, data read, and data write buses. The bus protocol supports splitted and outstanding transactions.

The first model was implemented at transaction-level layer 1, which means pin abstraction but timing accuracy. The bus protocol state machine can be implemented as a pure functional model which provides functional object interfaces for bus accesses. The bus state machine has to be triggered once per simulated clock cycle to be compliant with the timing accuracy. The structural model and data-flow model of a transaction-level layer 1 bus representation shows Figure . Because of possible outstanding transactions bus request queues are necessary and separate queues for incoming, outgoing (finished), read, and write request. The internal bus process has to check all the slave states and further has to update the protocol state machines. The finish queue is necessary to return an error state to the caller.

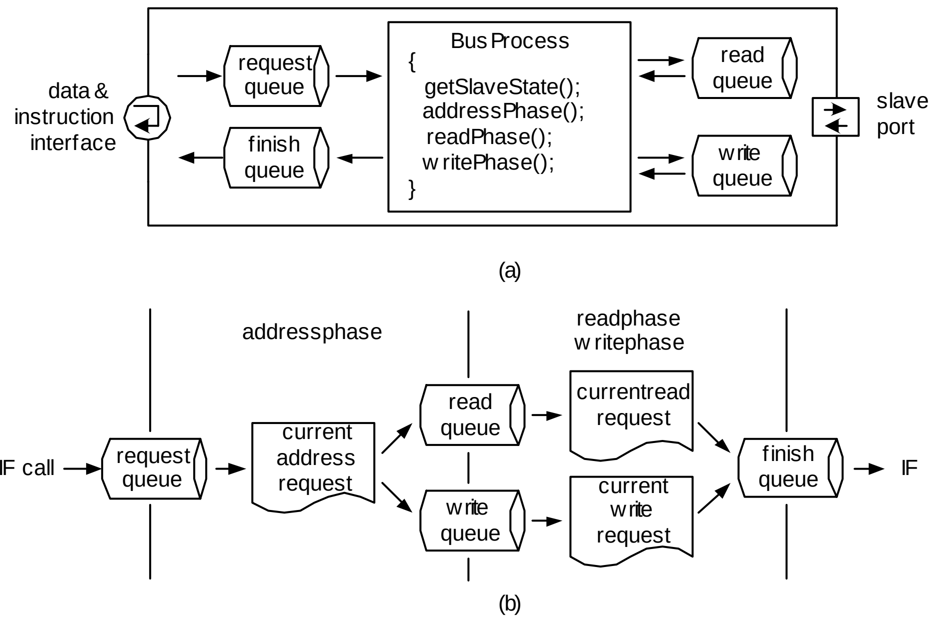


Figure 5: Transaction-level layer 1 bus model: (a) structural model: (b) data-flow model.

The energy estimation unit can be designed as a separate component and can be deactivated to increase simulation performance. The actual state of the bus model has to be passed to the energy estimation unit. The energy estimation unit consists of two parts. The first part translates bus requests into signal activity and the second contains the energy model which is driven by the signal activity. Thus a register transfer-level energy model for the system bus can be used. The model itself depends on the bus and process technology. The structure of the energy estimation unit is presented in Figure .

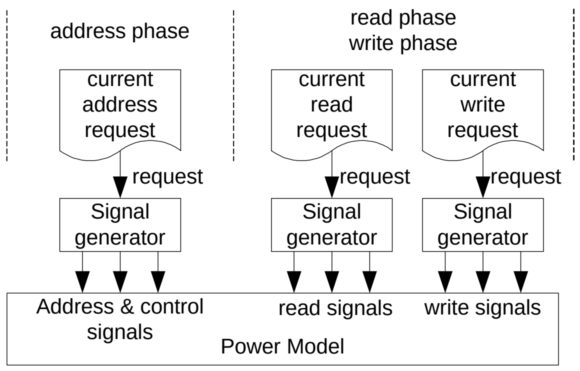


Figure 6: Coupling between bus model and bus power model.

It is also possible to abstract the bus model to transaction-level layer two. This model is not pin and timing accurate. Data items are passed by pointers and burst bus accesses are handled as single transactions. Thus a timing estimation is needed. The general bus representation is similar to the transaction-level layer one model but a simplified interface is provided to master and slave processing elements.

The energy estimation is divided into two phases, the address phase and data phase energy estimation. The dissipated energy is estimated for each phase in a single step because the necessary data and delays are in the bus request data structure. There are some more sources of inaccuracies additional to the layer one model. This model does not allow an accurate count of transitions of control signals due to a missing interaction with the slave. Also the energy model considers each transaction on its own and does not consider interactions between subsequent transactions.

*Energy Model for Secure Software Development*

The intention of this software energy estimation model is a flexible and accurate combination of the instruction-level energy model and data dependent models. The proposed model decouples instruction dependencies and data dependencies as presented in the next equation. The total energy $E_{total}$ consumed by a program is the sum of the energy consumption per cycle $E_{cycle}$ of all clock cycles $n_{total}$. The energy per clock cycle can be decomposed into four parts: instruction dependent energy dissipation $E_i$, data dependent energy dissipation $E_d$, energy dissipation of the cache system $E_c$, and finally the dissipation of all external components $E_e$ containing the bus system, memories and peripherals. This also leads to an independent instruction and data characterization process and enables a model refinement based on the requirements for accuracy.

$$E_{total} = \sum_{n=0}^{n_{total}} E_{cycle}(n) = \sum_{n=0}^{n_{total}} [E_i(n) + E_d(n) + E_c(n) + E_e(n)]$$

The instruction dependent part of this model is based on the instruction-level energy model defined by Tiwari [16]. It defines *base costs* (BC) for each instruction and considers switching activity between different consecutive instructions by the use of a *circuit state overhead* (CSO). The instruction dependent energy dissipation per cycle $E_i$(n) is the sum of the base costs $E_{BC}$ and the circuit state overhead $E_{CSO}$. Base costs and circuit state overhead are calculated based on the next equations. Where $n_s$ is the number of pipeline stages and *i[j]* is the instruction executed in stage *j*.

$$E_{BC} = \sum_{j=0}^{n_s} BC_j, withBC_j = \begin{cases} BC(i[j], j), if\ jisactive \\ BC_{stall,j}, if\ jisstalling \end{cases}$$

$$E_{CSO} = \sum_{j=1}^{n_s-1} CS(i[j], i[j+1], j)$$

Base costs are characterized in the same way as proposed in [16]. To characterize a particular instruction a loop is used which is small enough to fit in the instruction cache and large enough to minimize the effect of the needed branch instruction. But the difference to the original approach is the selection of operands. The same operands are used for the entire loop and source values are mostly set to zero. This choice minimizes switching activity due to operand and data dependencies like source register switches or switching activity in the ALU. The pipeline aware model [17] is used for cycle by cycle estimation. The main idea of this approach is to distribute the measured base costs among all pipeline stages. A uniform distribution is used for this model, the general context between the measured base costs and the base costs for one stage describes the following equation:

$$BC(i, j) = \frac{T_i \times BC_i - N_{i,nop} \times BC_{nop,j} - N_{i,stall} \times BC_{stall} - \alpha_i}{N_i}$$

*BC(i,j)* is the power consumed by stage *j* when executing instruction *i*, $BC_i$ is the base cost value measured for instruction *i*, $BC_{stall}$ is the base cost value for stalling, $T_i$ is the number of clock cycles, $N_{i,nop}$ is the number of stages executing *nop*, $N_{i,stall}$ is the number of stages in *stall* state, $N_i$ is the number of active stages for instruction *i* and  is a constant factor caused by stalling of instruction *i*.

The use of this equation can be illustrated with the target smart card processor. An abstract view of the pipeline architecture gives Figure 7. The architecture comprises an integer unit (IU) and a separate multiply-/divide unit (MDU) pipeline. The first and second pipeline stages are shared between both pipelines. If an instruction stays in a pipeline stage longer than one cycle, the control unit stalls only the previous pipeline stages and does not affect subsequent stages.

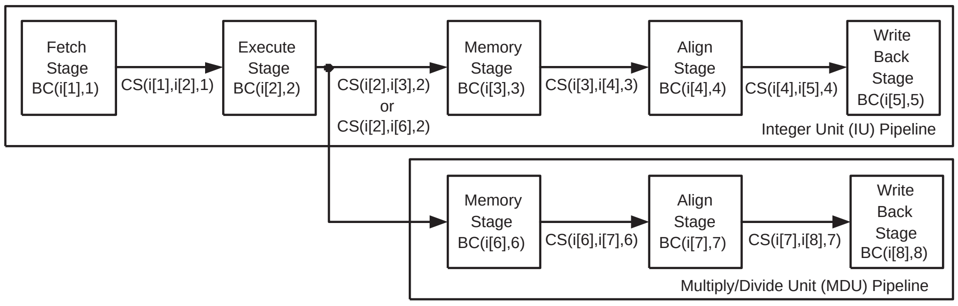


Figure 7: Instruction path energy model for a pipeline architecture.

A loop with two alternating instructions is used to measure the circuit state overhead *CS* between subsequent instructions. The CS between two consecutive pipeline stages *j* and *j+1* executing instructions $i_1$ and $i_2$ is given in the following equation. $n_{\mathrm{MDU}}$ and $n_{\mathrm{IU}}$ are the numbers of pipeline stages for MDU and IU pipeline, respectively. $\gamma_j$ is the specific weight for each pipeline transition. In this example $\gamma_j$ is specified equal for all transitions, but this can be changed to get a more realistic behaviour.

$$CS(i_1, i_2, j) = \gamma_j \{CS - \sum_{k=1}^{n_{MDU}} BC_{nop,j} - \frac{1}{2} \cdot \sum_{k=1}^{n_{IU}} [BC(i_1, k) + BC(i_2, k)]\}$$

$$\gamma_j = \frac{1}{n_{IU} - 1}$$

The data dependent energy consumption per cycle $E_d$(n) is the sum of the energy consumption of all pipeline stages. The energy consumption of a particular pipeline stage *E(i[j],j)* is determined by the instruction *i[j]* executed in this stage *j*:

$$E_d(n) = \sum_{j=1}^{n_s} E(i[j], j)$$

Each instruction is responsible for the calculation of its energy consumption for the pipeline stages. A micro-architectural energy model shared between all instructions is used to consider data dependent switching activity of buses and functional units. An instruction can select shared buses and functional units out of this model, but bit patterns and specific properties are calculated with its own instruction energy model.

The characterization of the basic behaviour of each instruction is the first step to get a data dependent model. A loop containing pairs of the same instruction with different source and target registers is used for this. The basic power consumption of this loop is determined by using the same source operand values as for base cost characterization. An additional power consumption compared to the base cost loop can be measured due to source and target register switching. This power value $P_{\mathrm{base}}$ is used as reference for all following measurements *P*. During the next step the source operand values of the first instruction are the same as in step one, but the source operands of the second instruction are changed. Thus the data path is brought back to an initial state every second instruction. The data dependent power consumption $P_{\mathrm{d}}$ is the difference between the measured power value *P* and $P_{\mathrm{base}}$. A footprint of each instruction can be drawn up by using this methodology. The footprints can be used to identify instructions sharing the same functional units. For example, the instructions *add*, *add immediate* and *sub* have the same power behaviour due to sharing the same adder. Of course, the mathematical functionality has to be taken into consideration. All instructions with the same behaviour are arranged in groups.

The next step is the identification of internal buses shared among different groups of instructions. Again a loop of alternating pairs of instructions is used. First, the first source operands of the two instructions are the same (e.g. 0xaa...) and second, they differ with maximum hamming distance (e.g. 0xaa... and 0x55...). The difference between the power values are used for bus switching characterization.

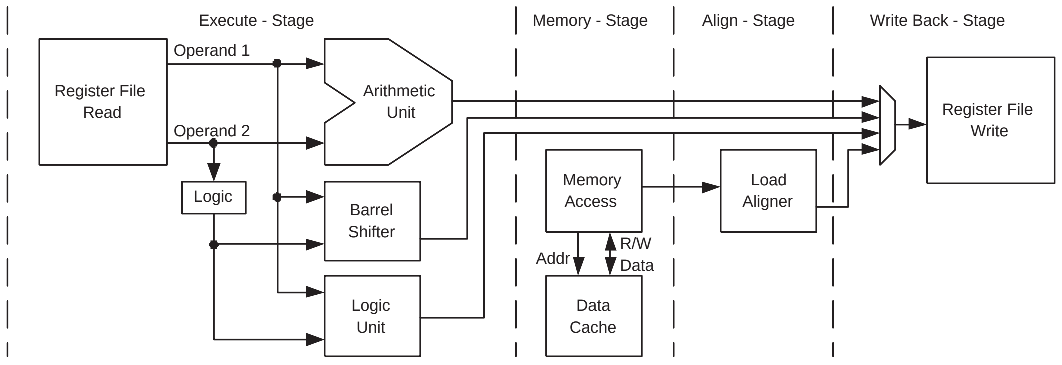
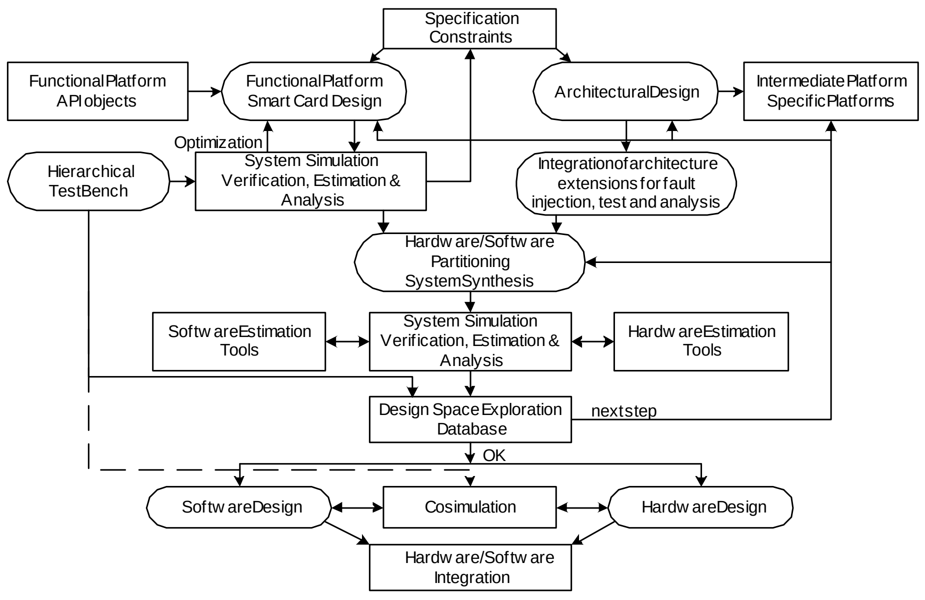


Figure 8: A data path energy model based on the instruction set characterization.

This methodology applied to the considered smart card CPU results in an architectural model for the IU-pipeline as presented in Figure 8. This model was integrated into the pipeline model presented in Figure 7. Each instruction has its own data dependent energy model, which contains references to the used shared energy models for each pipeline stage. This is a very flexible approach because data dependent power models can be extended or refined by referencing additional or other shared models.



Specification Constraints
Functional Platform API objects
Functional Platform Smart Card Design
Architectural Design
Intermediate Platform Specific Platforms
Optimization
Hierarchical TestBench
System Simulation Verification, Estimation & Analysis
Integration of architecture extensions for fault injection, test and analysis
Hardware/Software Partitioning System Synthesis
Software Estimation Tools
System Simulation Verification, Estimation & Analysis
Hardware Estimation Tools
Design Space Exploration Database
next step
OK
Software Design
Cosimulation
Hardware Design
Hardware/Software Integration

Figure 9: Developed design flow for power aware smart cards.

## 4.0 Design Flow

This section shows the design flow used together with the models presented above. The entry point for the design flow presented in Fig. 4 is an arbitrary non-executable behavioral description. During the first step an executable functional model has to be developed. This model can be optimized and tested until it fulfills all requirements. Because of the golden model character for further implementation this is an important milestone.

The purpose of the following stage is the mapping of the functional model on a transaction-level architecture, which has to be done by the user. Partitioning comprises mapping of functional modules, global memory map definition, and configuration of master and slave adapters. A synthesis system processes the functional model, the architecture description, and configuration information and produces an executable SystemC model. The design space can be explored by changing the target architecture, transforming the functional mapping, and reconfiguring the system mapping. When an architecture and mapping configuration with a high potential to meet the constraints have been found, integrated tools for software [15] and hardware power estimation can be used to get accurate energy results. The best solution can be selected out of a set of solutions which acts as a golden model for the further implementation. Subsequent stages are the standard SW and HW design flows.

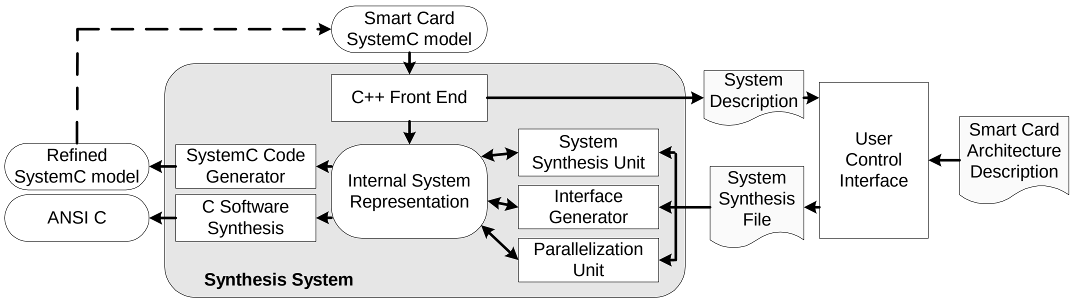


Figure 10: Synthesis system to process SystemC models and transaction-level architectures.

The implementation of the synthesis system is shown in Fig. 10. The SystemC model is processed by a commercial C++ front end provided by EDG [4]. An analysis unit was integrated into the front end to generate a system description in XML. The user front end analyzes the file and generates a graphical representation. The same is done for the architectural description which has to be provided by the user in XML. Mapping information is written to a system configuration file which is read by the synthesis tool. Three main blocks have been implemented to transform the internal representation of the SystemC model. The system synthesis unit is able to manipulate the entire system and the interface generator is responsible for HW/SW interfaces and corresponding adapters. The last unit, called parallel unit, is necessary to replace two modules executed sequential with two parallel processes, process synchronization, and process intercommunication. Two back ends allow the generation of a new SystemC model and alternative ANSI C code for cross-compilation.

## 5.0 Evaluation

The presented smart card system-level design approach was evaluated using the smart card operating system Java Card™ [11]. Java Card is a Java virtual machine customized for smart cards. First of all, a functional model was developed, which is presented in Fig. 11. The *command dispatcher* is the only active module in this model. It receives messages from the UART, processes them and calls corresponding methods of other modules. The *linker/loader* is responsible for post-installation of Java Card Applets. The *applet/system manager* initializes and controls the virtual machine. A simple software interpreter is used to interpret all Java Card bytecodes. The operand and local variables stack was implemented based on a RAM module.

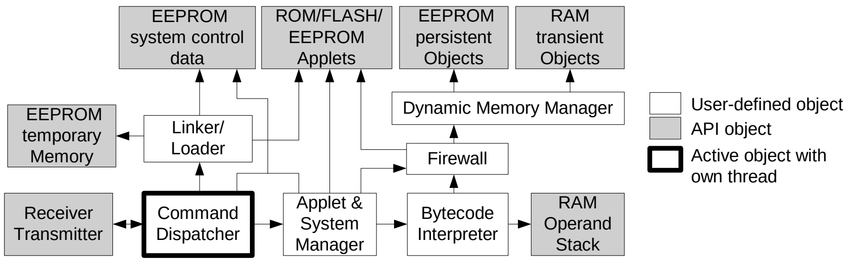


Figure 11: Functional Java Card™ model.

Table 1 shows the results of switching activity optimization at functional level and the effects at transaction-level. The results show that there is a strong correlation between the switching activities at both levels of abstraction. However, results demonstrate that optimization can be done at functional level, where the complexity and necessary effort to change models is low. The large influence of EEPROM programming cycles can be demonstrated at functional level.

Table 1: Bus switching activity (SA) evaluation for a functional model (Funct, noopt) and an optimized version (Funct, opt). These models have been mapped on an architecture (Arch, noopt; Arch, opt). Results are shown for binary bus encoding, T0 encoding and bus invert code (I).

| | Memory Accesses | Addr SA | Addr SA(T0) | Data SA | Data SA (I) |
|---|---|---|---|---|---|
| Funct, noopt | 221071 | 415333 | 428627 | 680335 | 663747 |
| Funct, opt | 203832 | 409491 | 436167 | 608183 | 591025 |
| Arch, noopt | 233832 | 686290 | 700040 | 781682 | 769463 |
| Arch, opt | 215817 | 636242 | 651088 | 672152 | 654668 |

Figure 12(a) presents the number of programming cycles necessary to install and execute a Java Card applet dependent on the EEPROM page size for all memory blocks. The JC model was mapped on two transaction-level architectural models, one containing a single EEPROM block and a second with two blocks. The impact on the necessary programming cycles is shown in Fig. 12(b).

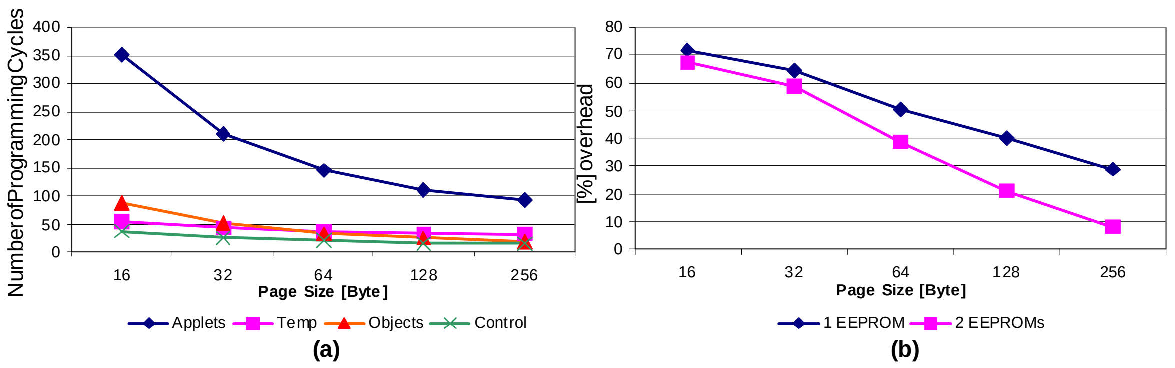


Figure 12: Exploration of non-volatile memory: (a) programming cycles dependent on EEPROM page size; (b) programming overhead due to memory block sharing.

At functional platform the aim is to minimize energy dissipation by algorithmic exploration and architecture-level low energy techniques. At cycle-accurate platform software has to be cross-compiled to estimate energy dissipation accurately. The effect of energy optimization is small compared to the more abstract levels. But at this level software power profile optimizations can be performed. As mentioned in the introduction, the power profile is important for the stability of RF-field powered embedded systems. For instance, a power profile optimization of a part of a DES algorithm software implementation shows the following Fig. 13. Optimization results are due to instruction re-ordering based on the energy characterization of the instruction-set. The graphical view shows the elimination of several peaks and Table 2 depicts the values for the average power dissipation and the standard deviation. The result is a more flat power profile which can be seen by the reduced standard deviation, but the median power dissipation as reduced insignificantly.

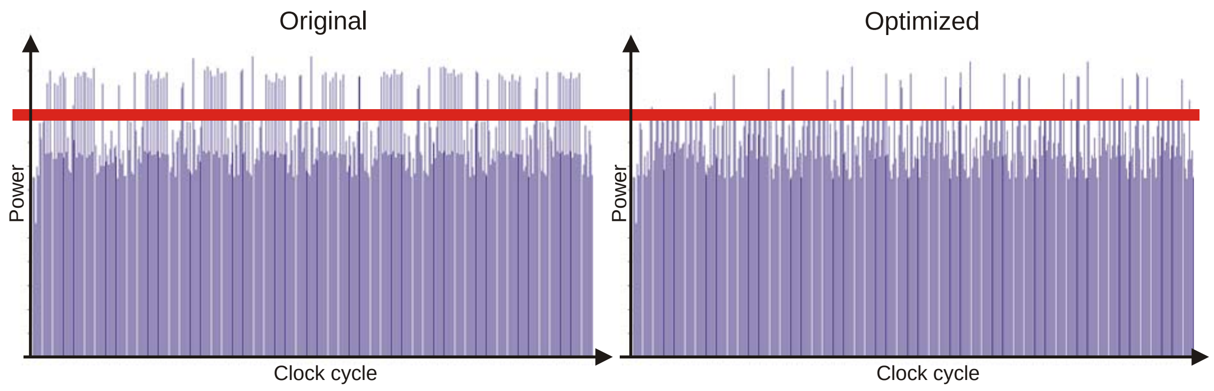


Figure 13: Instruction-level power profile optimization for a small part of the DES algorithm.

Table 2: Median power dissipation related to the power dissipation of a NOP instruction and standard deviation for the example presented in Fig. 13.

| | Median | Standard Deviation |
|---|---|---|
| Original Code | 191,5% | 30,6% |
| Optimized Code | 184,5% | 20,6% |

## 6.0 Conclusions

This paper has presented the abstraction of smart card designs to get the benefit of system-level optimization. Due to the intensive usage of non-volatile memory in smart card applications, the memory system is represented in detail. Furthermore the design flow and the basic concept of implemented tools were discussed. Preliminary results were presented and discussed.

## 7.0 References

[1] A. Haverinen, M. Leclercq, N. Weyrich, D. Wingard, "SystemC™ based soc communication modeling for the ocp™ protocol," white paper, *www.ocp-ip.org*, 2002.

[2] A. Macii, L. Benini, M. Poncino, *Memory Design Techniques for Low Energy Embedded Systems*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.

[3] A. Raghunathan, N. K. Jha, S. Dey, *High-Level Power Analysis and Optimization*, Kluwer Academic Publishing, Boston/Dordrecht/London, 1998.

[4] Edison Design Group (EDG), *C++ Front End Documentation*, www.edg.com, 2003.

[5] F. Balarin, *et.al.*, Hardware-Software Co-Design of Embedded Systems, Kluwer Academic Publishers, Boston/Dordrecht/London, 5th Printing 2003.

[6] L. Benini and G. De Micheli, "System-level power optimization: techniques and tools," *Proceedings of Int. Conference on Low Power Electronics and Design*, 1999.

[7] L. Benini *et. al.*, "Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems," *IEEE/ACM Great Lakes Symposium on VLSI*, March 1997.

[8] M. B. Stan, W. P. Burleson, "Bus-Invert Coding for Low-Power I/O" *IEEE Trans. on VLSI Systems*, 1995.

[9] Open SystemC Initiative (OSCI), "SystemC 2.0.1 LRM," Revision 1.0, www.systemc.org, 2003.

[10]R. Dömer, A. Gerstlauer, D. Gajski, „SpecC Language Reference Manual,“ Version 2.0, *SpecC Technology Open Consortium*, www.specc.org, December 2002.

[11]Sun Microsystem, Inc., "Java Card™ Virtual Machine Specification," Version 2.2.1, www.sun.com, 2004.

[12]U. Neffe *et. al.* "Energy Estimation based on Hierarchical Bus Models for Power Aware Smart Cards," *IEEE Design, Automation and Test in Europe Conference and Exhibition*, Designer's Forum, France, 2004.

[13]U. Neffe *et. al.*, "SystemC Based Design Space Exploration for Power Aware Smart Cards," *Proceedings Forum on Specification and Design Languages (FDL)*, France, 2004.

[14]U. Neffe *et. al.*, "A Flexible and Accurate Energy Model of an Instruction Set Simulator for Secure Software Design" *Proc. 14th Int Workshop on Power and Timing Modeling, Optimization and Synthesis*, Greece, 2004.

[15]W. Rankl, W. Effing, S*mart Card Handbook.* 4th Edition, Hanser-Verlag, Munich, 2002.

[16]V. Tiwari et.al., "Power Analysis of Embedded Systems: A First Step towards Software Power Minimization", IEEE Trans. VLSI Systems, Vol. 2, No. 4, 1994.

[17]M. Sami et.al., "An Instruction-Level Energy Model for Embedded VLIW Architectures", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 21, No. 9, 2002.