A Simulation-Based Power-Aware Architecture Exploration of a Multiprocessor System-on-Chip Design

F.Menichelli [1], M.Olivieri [1], L.Benini [2], M.Donno [3], L.Bisdounis [4]

[1] University of Rome La Sapienza, DIE Rome, ITALY

[2] University of Bologna DEIS Bologna, ITALY

[3] Bulldast s.r.l Torino, ITALY

[4] Intracom S.A. Athens, Greece

Abstract

We present the design exploration of a System-on-Chip architecture dedicated to the implementation of the HIPERLAN/2 communication protocol. The task was accomplished by means of an ad-hoc C++ simulation environment, integrating power models for CPUs, memories and buses used in the design and incorporating software profiling capabilities. The architecture is based on two ARM microprocessors, an AMBA bus and a local bus, DMA unit and other peripherals. Software mapping on the processor has been based on the power/ performance profiling results.

1. Introduction

Technology constraints (e.g. on chip power distribution) and market requirements (e.g. consumer products requiring low cost and low power dissipation), are pushing the demand of low power system design tools. For this reason, power optimization has been considered one of the key features for EDA support. While requiring some degree of accuracy, information on power consumption must be available as early as possible in the design cycle, possibly at the architectural level, in order to reduce the risk of time consuming iterations between high level and low level design phases.

These requirements are emphasized by the growing number of applications migrating toward System-on-Chip (SoC) architectures [1][2]. In these systems multiple processing elements are commonly present, along with multiple bus hierarchies, connected by bus bridges and DMA units.

We have developed a methodology for the architectural exploration of these systems, based on cycle accurate functional simulation and power consumption simulation.

In this article we describe the simulation tool we have developed in the context of the European IST-2000-30093 project "Energy-Aware SYstem-on-chip design of the HIPERLAN/2 standard".

The target of the EASY project is the design and realization of a complete SoC for portable applications that will implement all the functions of the HIPERLAN/2 wireless LAN standard (i.e. the baseband processing, the low-level MAC layer, the DLC layer and the communications with the host bus), as well as critical functionality of the IEEE 802.11a standard.

We describe the architecture level exploration made possible by the tool and we show how we were able to produce indications at the hardware and the software level to reach a higher level of power efficiency.

The backbone of our approach is a simulation tool with functional and power analysis capabilities. With this tool, we are able to obtain a very detailed estimate of the power consumption of the whole system, further subdivided into power consumption of each system component individually and of each software function.

The base of the simulation environment is the ARMulator instruction set simulator (ISS) which is part of the ADS1.2 suite [3]. ARMulator simulates the instruction sets and architecture of various ARM processors. None of the modules incorporated in ARMulator, including the ISS, models power consumption. In order to obtain energy profiles of the code under analysis, we added executable power models of the CPU, cache, memory and buses to the respective modules.

2. The EASY SoC architecture

The methodology and tools for architecture exploration have been primarily experimented on an industrial SoC design within the EASY SoC project. The primary objective of the EASY project is the design of a low-power SoC that will handle the control functions and data processing of the HIPERLAN/2 standard, from the lowest level layers (MAC and baseband) to the highest (DLC), as well as critical functionality of the lower MAC layer of the IEEE 802.11a standard. From one side it will interface directly to the analog IF and RF front-end; from the other side it will integrate both a PCI interface for the communication with the host processor, in the case of

Mobile Terminal operation, and an Ethernet interface, in the case of Access Point operation.

The preliminary architectural template of the EASY SoC that was proposed (Fig. 1) is based on the analysis of the HIPERLAN/2 protocol data processes. The main processor (i.e. protocol processor) used in the EASY SoC is an ARM7 processor with integrated Cache memory. This core processor and other devices, including internal SRAM, external memories' controller, DMA engine, PCI and Ethernet interfaces are connected to an AMBA AHB bus.

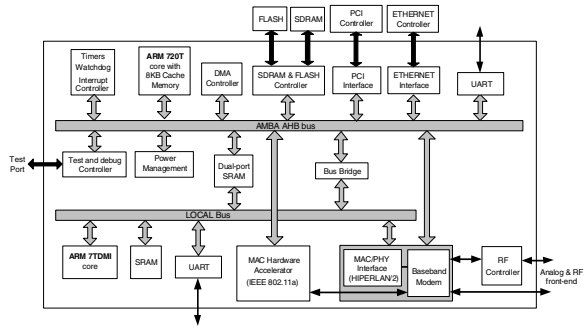

Fig. 1: Architectural template of the EASY SoC

A low-level MAC and baseband processing unit, is seen by the to the protocol processor as a multiple device connected to the AMBA AHB bus through a bus bridge (Fig. 1). It is based on a dedicated ARM7 core that is used to support timing critical tasks and to control the baseband modem. Included in the low-level MAC and baseband processing system, a block called MAC Hardware Accelerator contains modules that will implement critical processes of the IEEE 802.11a standard directly in hardware (i.e. encryption/decryption, fragmentation, timing control, protocol medium access, CRC, etc.).

The space of the partitioning of the functionalities between the three units is not completely free, but partly restricted by preliminary assumptions. These have been made especially on the functions of the MAC hardware accelerator.

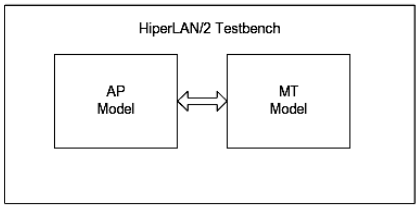

Fig. 2: High Level HIPERLAN/2 model

Partner INTRACOM has developed an executable HIPERLAN/2 C++ specification that implements the functionalities of a Mobile Terminal and of an Access Point [4]. Along with those items, a testbench (tester) has been included, which is used for the validation of the HIPERLAN/2 code functionalities. The tester creates one AP instance and one MT instance and lets them exchange Ethernet packets. The code (without considering the testbench) has basically the same functionalities that will be supported by the EASY SoC. As a consequence of the architectural template of Fig. 1, the whole system's functionalities should be partitioned between protocol processor, modem control processor and MAC hardware accelerator.

While Fig. 1 represents the actual SoC architecture, the high-level software model of the overall system [4] can be run on a single processor architecture. This is particularly useful because we can extract information using a software simulator, being still able to obtain significant indications on software functions consumption of CPU time and power. The criteria at the basis of partitioning are essentially two:

First, real-time constraints. Some of the data processing and control functions have to be performed with strict time constraints (for example, respecting the 10 μs time frame). These critical tasks can be relocated from the protocol processor to the lower-level MAC processor. Some of the task could be even implemented in hardware. From a previous implementation of a hardware accelerator for the IEEE 802.11a standard, some of these tasks are already known to be too timing constrained to run on a general purpose processor (for example encryption and decryption, fragmentation, timing control, protocol medium access and CRC).

Second, less severe but still important: power consumption. The EASY SoC, especially when used in a Mobile Terminal application, will be typically hosted in a system where the total amount of energy is limited, for example a notebook computer or a PDA. Once the most timing critical functions have been planned to be implemented in specialized hardware, the remaining timing constraints could be guaranteed, theoretically, raising CPU clock frequency and memory speed. But blindly raising speed can lead to a significant waste of energy consumption, compared to a more refined strategy, based on the quest for a correct balance between CPU clock frequency, cache and main memory size, specialized hardware support.

3. Design analysis and exploration environment

3.1. General structure and tools

The base of the simulation environment is the ARMulator ISS which is part of the ADS1.2 suite. ARMulator simulates the instruction sets and architecture of various ARM processors.

ARMulator consists of a series of modules, implemented as *Dynamic Link Libraries* (.dll files) for Windows. The main modules are:

- ARM processor core
- The memory used by the processor.

There are alternative predefined modules for each of these parts. One of the predefined memory models, *mapfile*, allows to specify a simulated memory system in detail. It allows to specify narrow memories and wait states.

In addition there are predefined modules which can be used to:

- model additional hardware, such as a coprocessor or peripherals
- extract debugging or benchmarking information (i.e. *Tracer* and *Profiler*).

In addition to these models, taking into account the presence of cache memory in many SoC architectures (e.g. the EASY SoC), we have written a cache memory model for the ARMulator.

The cache memory topologically resides between the ARM core and the program. It is modeled as a set associative memory. The number of sets, the width of the associative part and the size of a single line can be configured. Along with them access time for the first access and for the subsequent sequential accesses (burst accesses) can be specified.

Each simulated module consists of a functional model and two views, a timing model and an energy model, which are used to obtain results on execution time and power consumption of the system.

By integrating all the aforementioned modules in the ARMulator, the system architecture that has been built up in the simulator is shown in Fig. 3.

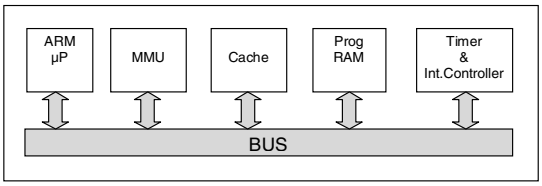

Fig. 3: Simulated system architecture

None of the modules incorporated in ARMulator, including the ISS, models power consumption of the relative component. In order to obtain energy profiles of the code under analysis, power models of the CPU, cache, memory and bus have been added to the respective modules. In this section we will briefly describe their implementation.

3.2. Power Models

CPU: The starting point of the model is inspired by a methodology first proposed by Tiwary et al. [5][6].

These models are based on the association of a cost (in term of energy consumption) to single instructions or sequences of instructions. These data can be obtained from lower level simulations (i.e. RTL) [7] or from physical measurement of the current drawn by a testchip. In our model we used physical measures using an improvement of the Tiwari's setup [8].

Memory: Cache and memory power models are the same, cache energy consumption is treated the same way as any other memory. Two energy consumption states are defined for each type of memory: *active* and *idle*. Details on the model of cache and memory can be found in [9].

The advantage of this model are the fact that it is technology independent, that it does not perform low-level time consuming simulations nor requires details about the circuit characteristics.

Bus: The bus is modeled as a fixed capacitance per line. For each cycle, the transitions on every line are calculated and the bus energy dissipation per cycle is obtained.

3.3. Profiling

Profiling enables us to find out where a program spent execution time and energy as well as which functions called other functions. This information can show which sections of a program are being called more or less often. Profiling is a well-known concept in software development because it can be used to locate which parts of a program are slower than expected, and might be candidates for rewriting to make program execute faster. In our case it has been used to locate parts of the code which could be possible candidates for hardware implementation of their function, in order to increment computational speed and/or to reduce power consumption of the whole system.

Time and power profiling in ARMulator has been implemented by adding a new module, similar to the profiler module already available. The profiler module is activated at every *executed* instruction. It takes as inputs the current instruction address and resolves the function the instruction belongs to. During execution the program call-graph is generated, where cycles (from the delay models) and energy (from the power models) spent in each function are annotated.

4. Optimizing the EASY SoC

4.1. Architectural exploration

Here we present the results obtained from the architectural parameters exploration, and in particular on the exploration of cache size. From Fig. 1 we see that the architecture has an ARM 720T as protocol processor,

which contains an 8kbyte unified cache. The cache contains 512 lines of 16 byte (four words), arranged as a 4-way set-associative cache.

In this analysis we will set the cache size (namely, the number of lines) as a variable parameter in the architecture, and will explore caches of increasing size.

Cache access energy depends on its size, the values have been derived from the CACTI 3.0 tool [10], they are reported in Table 1 for a 0.18um technology.

Size	access energy (nj)
1k	0,765
2k	0,781
4k	0,804
8k	0,869
16k	0,929

Table 1: Cache access energy

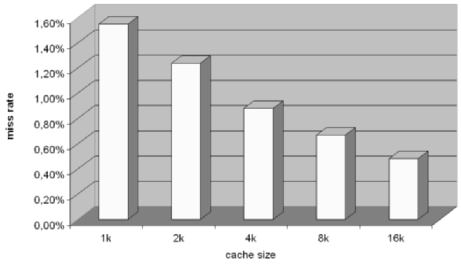

Fig. 4: Miss rate for increasing cache size

Fig. 4 reports the results obtained for miss rate using caches of increasing size. The numbers indicate that the real 8kbyte cache size guarantees an adequate low miss rate, considering that the final application should be larger and more complex (and therefore miss rate is supposed to rise), but the software under test contains both the Access Point and the Mobile Terminal functions, while in the real application they will be split in two separate units.

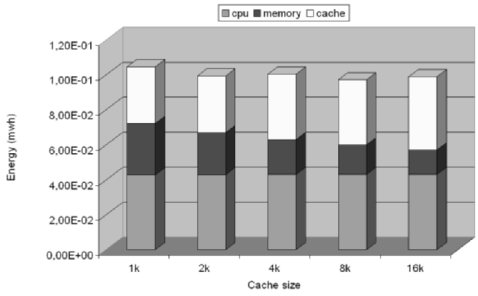

Fig. 5: Energy spent after 50 packets exchange

Fig. 5 shows the energy spent by cpu, memory and cache to send 50 packets. Cache energy increases incrementing its size, while memory energy decreases, because miss rate decreases. This trade-off brings to a minimum in energy consumption. With the above mentioned parameters for cache access energy, this minimum is obtained for 8kbyte cache size.

4.2. Mapping

In this section we illustrate software functions profiling, trying to select the functions that are critical from the time or energy point of view.

Table 2 and 3 present a partial vision of the profiling data, which in their integrity contain about 4000 functions.

The functions are sorted in decreasing order of time and power consumption (self column), so they are the most significant from the time/power consumption point of view. Moreover, pruning has been accomplished to leave out uninteresting data on library functions or testbench functions.

index	cycles	self	children	called	name
4	15535545	9004746	6530799	188	<_ZN19MT_ECL_Sender_Actor9smart_dmaERjPhjjj>:
5	12235730	6955497	5280233	133	<_ZN19AP_ECL_Sender_Actor9smart_dmaERjPhjjj>:
18	1371658	1353810	17848	17	<_ZN11EC_Tx_unackC1ERK7DLCC_IDRK6MAC_IDRKhS7_>:
20	1287077	1179221	107856	26	<_ZN21AP_ECL_Receiver_Actor9smart_dmaERK6EC_dmaRj>:
24	1607147	1049756	557391	25	<_ZN21MT_ECL_Receiver_Actor9smart_dmaERK6EC_dmaRj>:
32	2305221	804927	1500294	106	<_ZN22AP_Frame_Decoder_Actor18transition3_decode>:
38	1438867	572759	866108	79	<_ZN22AP_Frame_Builder_Actor24choicePoint1_DmaListCopy>:
42	935026	538482	396544	810	<_ZN11EC_Rx_unack9Update_CLERj>:
51	2031375	393710	1637665	100	<_ZN22MT_Frame_Decoder_Actor18transition3_decode>:
52	1753062	393294	1359768	92	<_ZN22MT_Frame_Builder_Actor24choicePoint1_DmaListCopy>:
53	627544	391761	235783	568	<_ZN22MT_Frame_Builder_Actor9smart_dmaEPKhPhRKjS4_>:
56	2424221	343388	2080833	97	<_ZN18AP_Scheduler_Actor10create_FCHERKjRKfS3_S3_S3_>:
57	324245	324245	0	487	<_ZN22AP_Frame_Decoder_Actor9smart_dmaEPKhPhj>:
58	314498	314498	0	381	<_ZN22AP_Frame_Builder_Actor9smart_dmaEPKhPhRKjS4_>:
63	790784	260870	529914	59	<_ZN22MT_Frame_Builder_Actor17transition3_build>:
64	728245	258976	469269	321	<_ZN22MT_Frame_Decoder_Actor9smart_dmaEPKhPhj>:
67	1358439	232637	1125802	61	<_ZN16Connection_Table20update_traffic_tableEP13Traffic_TableRj>:
71	431864	198790	233074	692	<_ZN11EC_Rx_unack14Update_DecoderEj>:
74	1588821	195477	1393344	251	<_ZN11EC_Tx_unack9Update_CLEjR6EC_dma>:
78	189849	184302	5547	1081	< memcpy>:

Table 2: First most time-consuming functions ("self" column)

index	energy	self	children	called	name
4	2.56e-002	1.48e-002	1.08e-002	188	<_ZN19MT_ECL_Sender_Actor9smart_dmaERjPhjjj>:
5	2.02e-002	1.14e-002	8.79e-003	133	<_ZN19AP_ECL_Sender_Actor9smart_dmaERjPhjjj>:
18	2.15e-003	2.12e-003	2.90e-005	17	<_ZN11EC_Tx_unackC1ERK7DLCC_IDRK6MAC_IDRKhS7_>:
20	2.05e-003	1.88e-003	1.74e-004	26	<_ZN21AP_ECL_Receiver_Actor9smart_dmaERK6EC_dmaRj>:
24	2.61e-003	1.68e-003	9.30e-004	25	<_ZN21MT_ECL_Receiver_Actor9smart_dmaERK6EC_dmaRj>:
32	3.78e-003	1.34e-003	2.44e-003	106	<_ZN22AP_Frame_Decoder_Actor18transition3_decode>:
38	2.36e-003	9.20e-004	1.44e-003	79	<_ZN22AP_Frame_Builder_Actor24choicePoint1_DmaListCopy>:
42	1.54e-003	8.86e-004	6.54e-004	810	<_ZN11EC_Rx_unack9Update_CLERj>:
51	3.33e-003	6.53e-004	2.68e-003	100	<_ZN22MT_Frame_Decoder_Actor18transition3_decode>:
52	2.87e-003	6.40e-004	2.23e-003	92	<_ZN22MT_Frame_Builder_Actor24choicePoint1_DmaListCopy>:
53	1.03e-003	6.40e-004	3.87e-004	568	<_ZN22MT_Frame_Builder_Actor9smart_dmaEPKhPhRKjS4_>:
56	3.98e-003	5.71e-004	3.41e-003	97	<_ZN18AP_Scheduler_Actor10create_FCHERKjRKfS3_S3_S3_>:
57	5.23e-004	5.23e-004	0.00e+000	487	<_ZN22AP_Frame_Decoder_Actor9smart_dmaEPKhPhj>:
58	5.15e-004	5.15e-004	0.00e+000	381	<_ZN22AP_Frame_Builder_Actor9smart_dmaEPKhPhRKjS4_>:
63	1.29e-003	4.23e-004	8.69e-004	59	<_ZN22MT_Frame_Builder_Actor17transition3_build>:
64	1.19e-003	4.19e-004	7.70e-004	321	<_ZN22MT_Frame_Decoder_Actor9smart_dmaEPKhPhj>:
67	2.23e-003	3.82e-004	1.85e-003	61	<_ZN16Connection_Table20update_traffic_table>:
71	7.10e-004	3.25e-004	3.85e-004	692	<_ZN11EC_Rx_unack14Update_DecoderEj>:
74	2.63e-003	3.22e-004	2.31e-003	251	<_ZN11EC_Tx_unack9Update_CLEjR6EC_dma>:
78	3.29e-004	3.20e-004	8.95e-006	1081	< memcpy>:

Table 3: First most energy-consuming functions ("self" column)

A discussion of significant functions that appear in the tables follows. From Table 2 and 3 we see that there is a correspondence between time consumption and energy consumption (i.e. time consuming function are also the most energy hungry). We will refer to the functions using the *index* column of Table 1 and Table 2.

1. Memory transfer functions (index 4,5,20,24):

These functions perform a copy of Ethernet packets from the source to the destination memory, using CPU instructions. This function can be mapped as a DMA transfer in the final architecture.

2. Memory transfer functions (index 53,57,58,64):

These are all memory transfer functions mapped on the low level protocol processor (in the following referred as ARM2) and the data transfer involve ARM2 and the modem buffer. A DMA transfer between this units should lower the functions cost.

3. Packet decoding functions (index 32, 51)

These functions perform the low level packets decoding before passing them to the DLC queues. These functions are particularly processing intensive and must obey to real-time constraints. They have been mapped on ARM2 in the final architecture.

4. Frame builder functions (index 38, 52, 63)

The class Frame_Builder prepares the data for transmission by the modem. It is mapped in ARM2 in the final architecture. The DmaListCopy methods (index 38 and 52) control the DMA transfer of packets to the modem and, in the current implementation, copy them through software loops. Their impact is supposed to be mitigated by the adoption of hardware DMA transfers.

The build method of the Frame_Builder_Actor class (index 63) performs heavy processing in order to build packets ready to be transferred to the modem. Due to the heterogeneity of elaboration, no hardware solutions seem to be practical. A solution to decrease the cost of this function can be the use of focused manual or automatic code optimization.

5. Connection table update functions (index 67)

This is another heavy computational function. The traffic table is a list that contains the sizes of the DLC queues. It is updated on a frame basis and is used by the scheduler in order to create the map of the HIPERLAN/2 frame and divide the bandwidth among the connections. Decreasing its cost through code optimization can be the most practical solution for this function.

5. Discussion

We have shown how, from a large collection of profiling data, we have pinpointed some hot functions in the HIPERLAN/2 implementation. Following the same procedure other functions can be individuated.

We can argue that a favorable implementation of some functions in hardware is through the utilization of programmable logic directly connected to the CPU bus. Moreover since most of these functions operate on global variables it is required that a dedicated unit incorporates direct memory access functions and therefore may be usefully integrated in the same FPGA or ASIC in which the DMA controller softcore [11] is implemented. Fig. 6 sketches the paths followed by most DMA operations.

On the other hand, some functions have been identified for which the most practical intervention is software optimization.

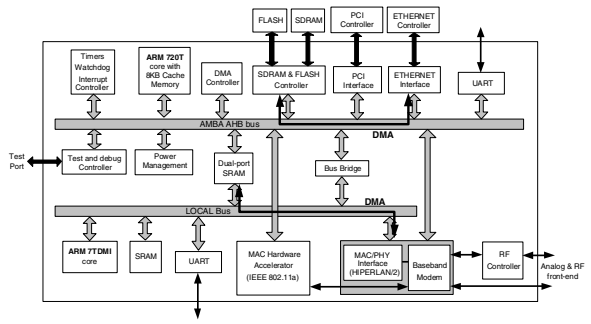

Fig. 6: Direct memory access most frequent paths

5.1. Expected impact of bus contention on mapping

From our simulations the utilization of the local bus accessed by the ARM cache controller is below 25%. In Fig. 7 we report the total execution time (simulated time) of the benchmark and the memory active time, which corresponds to the time the bus is occupied.

These numbers take into account the bus traffic due to DMA operation, emulated by software routines. In the final EASY SoC architecture (Fig. 1) the bus is split into an AMBA bus accessed by the protocol ARM processor and a local bus by low level protocol processor. As a result it is expected that in the final EASY SoC architecture the bus contention does not represent either a performance bottleneck or a source of energy consuming waiting loops in the processors.

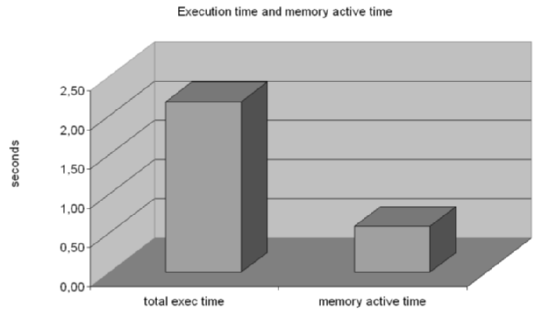

Fig. 7: Total execution time and memory active time

6. Conclusions

The design of the EASY System-on-Chip architecture, starting from a fixed template, was refined and partially modified on the basis of simulation results obtained from a dedicated exploration tool. The mapping of the software routines on the two available processors, in particular, was strongly influenced by the power and performance profiling results. Bus contention was quantitatively analyzed resulting to be not a real problem for the target application. The experimented exploration environment developed for the presented design can be extended and leveraged for other multiprocessor SoC architectures.

7. Acknowledgements

This work was performed within the IST-2000-30093 EASY project, and supported by EU funding programme.

8. References

[1] Philips Electronics, “Nexperia”, http://www.semiconductors.philips.com/products/nexperia/index.html
[2]IBM “Blue Logic” SoC, http://www-3.ibm.com/chips/products/asics/products/soc.html
[3] http://www.arm.com/devtools/ads?OpenDocument
[4] C. Drosos, D. Metafas, L. Bisdounis, “High-level model of the overall system”, IST-2000-30093/EASY Project, Doc. ID: EASY/WP1/ ICOM/DL/I/D12/B1
[5] V. Tiwari, S. Malik, and A. Wolfe. “Power analysis of embedded software: A first step towards software power minimization. IEEE Transactions on VLSI Systems, 2(4):437{445, December 1994.
[6] V.Tiwari, S. Malik, A. Wolfe, “Instruction Level Power Analysis and Optimization of Software”, Journal of VLSI Signal Processing, , 1-18 (1996)
[7] BullDAST s.r.l., “PowerChecker”, http://www.bulldast.com/prod01.htm
[8] N. Kavvadias, P. Neofotistos, S. Nikolaidis, K. Kosmatopoulos, Th. Laopoulos, “Measurements analysis of the software-related power consumption in microprocessors”, IMTC '03, Proceedings of the 20th IEEE, Volume: 2 , 20-22 May 2003.
[9] T. Simunic, L. Benini, G. De Micheli, “Cycle-Accurate Simulation of Energy Consumption in Embedded Systems”, IEEE Transactions on Computer-Aided Design of Circuits and Systems, Vol. 9, No. 1, pp. 15-28, February 2001.
[10] P. Shivakumar, N.P. Jouppi, “CACTI 3.0: An Integrated Cache Timing, Power, and Area Model”, HP Labs Technical Reports, 2001, http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-2001-2.html.
[11] L. Bisdounis, A. Vontzalidis, S. Kougia, L. Tsoura, T. Pagonis, F. Ieromnimon, A. Tatsaki, C. Dre, S. Blionas, P. Delamotte, M.Speitel, O. Abderrahim, “System’s macrocells – Final”, IST-2000-30093/EASY Project, Doc. ID: EASY/WP1/ICOM/PI/I/D32/B1