Automated Functional Verification of Application Specific Instruction-set Processors*

Marcela Šimková, Zdeněk Přikryl, Zdeněk Kotásek, and Tomáš Hruška

Faculty of Information Technology, Brno University of Technology, Czech Republic
{isimkova,iprikryl,kotasek,hruska}@fit.vutbr.cz

Abstract. Nowadays highly competitive market of consumer electronics is very sensitive to the time it takes to introduce a new product. However, the ever-growing complexity of application specific instruction-set processors (ASIPs) which are inseparable parts of nowadays complex embedded systems makes this task even more challenging. In ASIPs, it is necessary to test and verify significantly bigger portion of logic, tricky timing behaviour or specific corner cases in a defined time schedule. As a consequence, the gap between the proposed verification plan and the quality of verification tasks is widening due to this time restriction. One way how to solve this issue is using faster, efficient and cost-effective methods of verification. The aim of this paper is to introduce an automated generation of SystemVerilog verification environments (testbenches) for verification of ASIPs. Results show that our approach reduces the time and effort needed for implementation of testbenches significantly and is robust enough to detect also well-hidden bugs.

1 Introduction

The core of current complex embedded systems is usually formed by one or more processors. The use of processors brings advantages of a programmable solution mainly the possibility of a software change after the product is shifted to the market.

The types of processors that are used within embedded systems are typically determined by an application itself. One can use *general purpose processors* (GPPs) or *application specific instruction-set processors* (ASIPs) or their combination. The advantages of GPPs are their availability on the market and an acceptable price because they are manufactured in millions or more. On the other hand, their performance, power consumption and area are worse in comparison to ASIPs that are highly optimised for a given task and therefore, have much better parameters.

However, one needs a powerful and easy-to-use tools for the ASIPs design, testing and verification as well as tools for their programming and simulation. These tools often use *architecture description languages* (ADLs) [6] for the description of a processor.

* This work was supported by the European Social Fund (ESF) in the project Excellent Young Researchers at BUT (CZ.1.07/2.3.00/30.0039), the IT4Innovations Centre of Excellence (CZ.1.05/1.1.00/02.0070), Brno Ph.D. Talent Scholarship Programme, the BUT FIT project FIT-S-11-1, research plan no. MSM0021630528, and the research funding MPO ČR no. FR-TI1/038.

G. Schirner et al. (Eds.): IESS 2013, IFIP AICT 403, pp. 128–138, 2013.

ADL allows automated generation of the programming tools such as C/C++ compiler or assembler, simulation tools such as *instruction-set simulator* (IIS) or profiler. Moreover, the representation of the processor in *hardware description language* (HDL) such as VHDL or Verilog is generated from this description. If the designer wants to change a design somehow (add a new feature, fix a bug), he or she just change the processor description and all tools as well as the hardware description are re-generated. This allows really fast design space exploration [4] within processor design phases. Examples of ADLs are LISA [2], ArchC [9], nML [1] or CodAL [3].

In some cases, when a special functional unit is added for instance a modular arithmetic unit (in cryptography processors) additional verification steps should be performed. The reason is that description of instructions can be different for simulation/hardware and for the C compiler (e.g. LISA language has separate sections describing the same feature for the simulator and for the C compiler). In this case, one should verify that the generated hardware can be programmed by the generated C compiler. In other words, the C compiler should be verified with respects to the features of a processor. Therefore, it is highly desirable to have a tool that automatically generates verification environments and allows checking all above mentioned properties.

In this paper we propose an innovative approach for automated generation of verification environments which is easily applicable in the development cycle of processors. As a development environment we utilise the Codasip framework [3] but the main principles are applicable in other environments as well. In order to verify a hardware representation of processors with respect to the generated C/C++ compiler we decided to apply functional verification approach as it offers perfect scalability and speed.

The following section shows the state of the art in the development of processors and description of verification techniques which are typically used in this field. Afterwards, the Codasip framework is described in Section 3. Our approach is introduced in Section 4, and Section 5 presents experimental results. In the end of this paper, the conclusions and our plans for future research are mentioned.

2 Verification in the Development Cycle of Processors

The following subsections introduce verification techniques used within processor design phases as well as research projects and companies dealing with processor design.

2.1 Verification Approaches

For verification of processors a variety of options exists: (*i*) formal verification, (*ii*) simulation and testing, and (*iii*) functional verification. However, their nature and preconditions for the speed, user expertise and complexity of the verification process are often a limiting factor.

Formal verification is an approach based on an exhaustive exploration of the state space of a system, hence it is potentially able to formally prove correctness of a system. The main disadvantages of this method are state space explosion for real-world systems and the need to provide formal specifications of behaviour of the system which makes this method often hard to use.

Simulation and testing, on the other hand, are based on observing the behaviour of the verified system in a limited number of situations, thus it provides only a partial guarantee of correctness of the system. However, because tests focus mainly on the typical use of the system and on corner cases, this is often sufficient. Moreover, writing tests is usually faster and easier than writing formal specifications.

Functional verification is a simulation-based method that does not prove correctness of a system but uses advanced verification techniques for reaching verification closure. Verification environments (or testbenches) are typically implemented in some hardware verification language, e.g. in SystemVerilog, OpenVera or *e*. During verification, a set of constrained-random test vectors is typically generated and the behaviour of the system for these vectors is compared with the behaviour specified by a provided *reference model* (which is called *scoreboarding*). In order to measure progress in functional verification (using coverage metrics), it is necessary to (*i*) find a way how to generate test vectors that cover critical parts of the state space, and (*ii*) maximise the number of vectors tested. To facilitate the process of verification and to formally express the intended behaviour, internal synchronisation, and expected operations of the system, *assertions* may be used.

All above mentioned features are effective in checking the correctness of the system and maximising the efficiency of the overall verification process. The popularity of functional verification is claimed by the existence of various verification methodologies, with OVM (*Open Verification Methodology*) [5] and UVM (*Universal Verification Methodology*) [5] being mostly used. They offer a free library of basic classes and examples implemented in SystemVerilog and define how to build easily reusable and scalable verification environments.

2.2 Design and Verification of Processors

There exist several research projects or companies dealing with processor design using ADLs. One of them is an open-source *ArchC* project [9]. A processor is described using ArchC language and the semantics of instructions is described using SystemC constructions [11]. All programming tools as well as IIS can be generated from the description in ArchC. The generation of a hardware representation is currently under development and there is no mention of verifying them so far.

Synopsys company offers *Processor Designer* [8] for designing processors. It uses LISA language. The programming tools, ISS as well as the HDL representation can be generated from the processor description. However, they do not provide any automatically generated verification environments. As mentioned in Section 1, instructions are described in two ways, the first one is used by the generator of the C compiler and the second one is used by ISS. Therefore, if the descriptions of instructions for ISS and the C compiler are not equivalent, then there is no automatic way how to detect it. In that case, C compiler cannot program the target processor properly.

Target company uses an enhanced version of nML language for the description of a processor microarchitecture [12]. The C compiler and ISS can be generated from the description of a processor as well as the HDL representation. According to the web presentation, the verification environment consists of a test program generator. It emits programs in assembly language (note that the assembly language is processor specific).

The generated test program is loaded by the ISS and a third party RTL simulator which evaluates the generated HDL representation. The program is executed and after the testing phase is completed, results are compared. If they are equal, then the test passed, otherwise an inconsistency is found and it needs to be investigated further. Nevertheless, a generator of test programs in some high-level language like C or C++ is missing. Therefore, the C compiler itself is not verified with respect to the processor.

When focusing on automated generation of verification environments in general (not necessary for processors), there already exist some commercial solutions which are quite close to our work. One example is *Pioneer NTB* from *Synopsys* [10] which enables to generate SystemVerilog and OpenVera testbenches for different hardware designs written in VHDL or Verilog with inbuilt support for third-party simulators, constrained-random stimulus generation, assertions and coverage. Another example is *SystemVerilog Frameworks Template Generator* (SVF-TG) [7] which assists in creating and maintaining verification environments in SystemVerilog according to the *Verification Methodology Manual* (VMM).

3 Codasip Framework

The Codasip Framework is a product of the Codasip company and represents a development environment for ASIPs. As the main description language it utilises ADL called CodAL [3]. It is based on the C language and has been developed by the Codasip company in cooperation with Brno University of Technology, Faculty of Information Technology. All mainstream types of processor architectures such as RISC, CISC or VLIW can be described.

The CodAL language allows two kinds of descriptions. In the early stage of the design space exploration a designer creates only the instruction-set (*the instruction-accurate description*). It contains information about instruction decoders, the semantics of instructions and resources of the processor. Using this description, programming tools such as a C/C++ compiler and simulation tools can be properly generated. The C/C++ compiler is based on LLVM platform [13].

As soon as the instruction-set is stabilised a designer can add information about processor microarchitecture (*cycle-accurate description*) which allows generating programming tools (without the C/C++ compiler), RTL simulators and the HDL representation of the processor (in VHDL or Verilog). As a result, two models of the same processor on different level of abstraction exist.

It is important to point out that in our generated verification environments the instruction-accurate description is taken as a golden (reference) model and the cycle-accurate description is verified against it.

The instruction-accurate description can be transformed into several formal models which are used for capturing particular features of a processor. Formal models which are used in our solution are *decoding trees* in case of instruction decoders and *abstract syntax trees* in case of semantics of instructions [14]. All formal models are optimised and then normalised into the abstract syntax tree representation that can be transformed automatically into different implementations (mainly in C/C++ languages). The generated code together with the additional code (it represents resources of processor such as registers or memories) forms ISS.

It should be noted that some parts of the generated code can be reused further in the golden model for verification purposes (more information in Section 4). At the same time as the golden model is generated, connections to the verification environment are established via the *direct programming interface* (DPI) in SystemVerilog. Automated generation of golden models reduces the time needed for implementation of verification environments significantly. Of course, a designer can always rewrite or complement the golden model manually.

The cycle-accurate description of a processor can be transformed into the same formal models as in case of the instruction-accurate description. Besides them, the processor microarchitecture is captured using *activation graphs*. In case of the cycle-accurate description, the formal models are normalised into the *component* representation. Each component represents either a construction in the CodAL language such as arithmetic operators or processor resources or it represents an entity at the higher level of abstraction such as the instruction decoder or a functional unit. Two fundamental ideas are present in this model, (*i*) components can communicate with each other using ports and (*ii*) components can exist within each other. In this way, component representation is closely related to HDLs and serves as an input for the HDL generator as well as for the generator of verification environments.

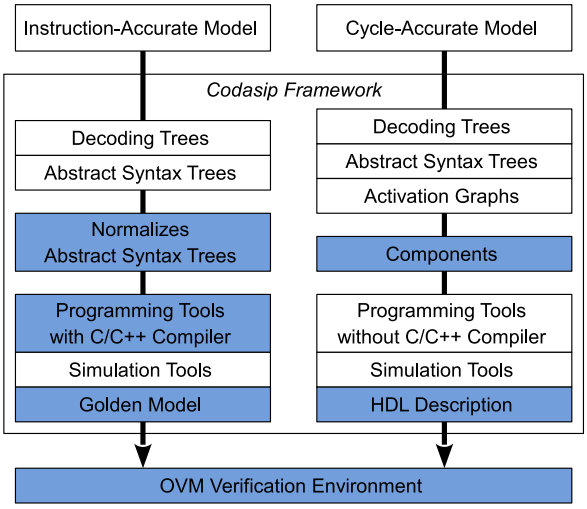

Fig. 1. Verification flow in the Codasip Framework

For better comprehension of the previous text, the idea is summarised once again in Figure 1. Codasip works with the instruction and the cycle-accurate description of a processor and specific tools are generated from these descriptions. The highlighted parts are used during the verification process. It should be noted that the presented idea is generally applicable and is not restricted only to the Codasip Framework.

Verification environments generated from formal models are thoroughly described in the following section.

4 Functional Verification Environments for Processors

The goal of functional verification is to establish the conformance of a design of processor to its specification. However, considerable time is consumed by designing and implementation of verification environments.

In order to comfortably debug and verify ASIPs designed in the Codasip framework as fast as possible and not waste time with implementation tasks we designed a special feature allowing automated pre-generation of OVM verification environment for every processor. In this way we can highly reuse the specification model provided in the CodAL language and all intermediate representations of the processor for comprehensive generation of all units.

Our main strategy for building robust verification environments is to comply with principles of OVM (they are depicted in Figure 2). We have fulfilled this task in the following way:

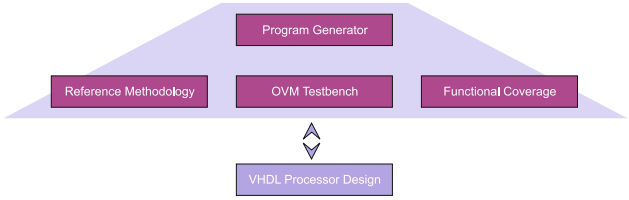

Fig. 2. Verification methodology

1. **OVM Testbench.** Codasip supports automated generation of object-oriented testbench environments created with compliance to open, standard and widely used OVM methodology.
2. **Program Generator.** During verification we need to trigger architectural and microarchitectural events defined in the verification plan and ensure that all corner cases and interesting scenarios are exercised and bugs hidden in them are exposed. For achieving the high level of coverage closure of every design of processor it is possible to utilise either a generator of simple C/C++ programs in some third-party tool or already prepared set of benchmark programs.
3. **Reference Methodology.** A significant benefit of our approach is gained by automated creation of golden models for functional verification purposes. We realised that it is possible to reuse formal models of the instruction-accurate description of the processor at the higher level of abstraction and generate C/C++ representation of these models in the form of reference functions which are prepared for every instruction of the processor. Moreover, we are able to generate SystemVerilog encapsulations, so the designer can write his/her own golden model with advantage of the pre-generated connection to other parts of the verification environment.
4. **Functional Coverage.** According to the high-level description of the processor and the low-level representation of the same processor in VHDL, we are able to automatically extract interesting coverage scenarios and pre-generate coverage points for comprehensive checking of functionality and complex behaviour of the processor. Of course, it is highly recommended to users to add some specific coverage

points manually. Nevertheless, the built-in coverage methodology allows measuring the progress towards the verification goals much faster.

In addition, interconnection with a third-party simulator e.g. ModelSim from Mentor Graphics allows us to implicitly support assertion analysis, code coverage analysis and signals visibility during all verification runs. A general architecture of generated OVM testbenches is depicted in Figure 3 and main components are described further.

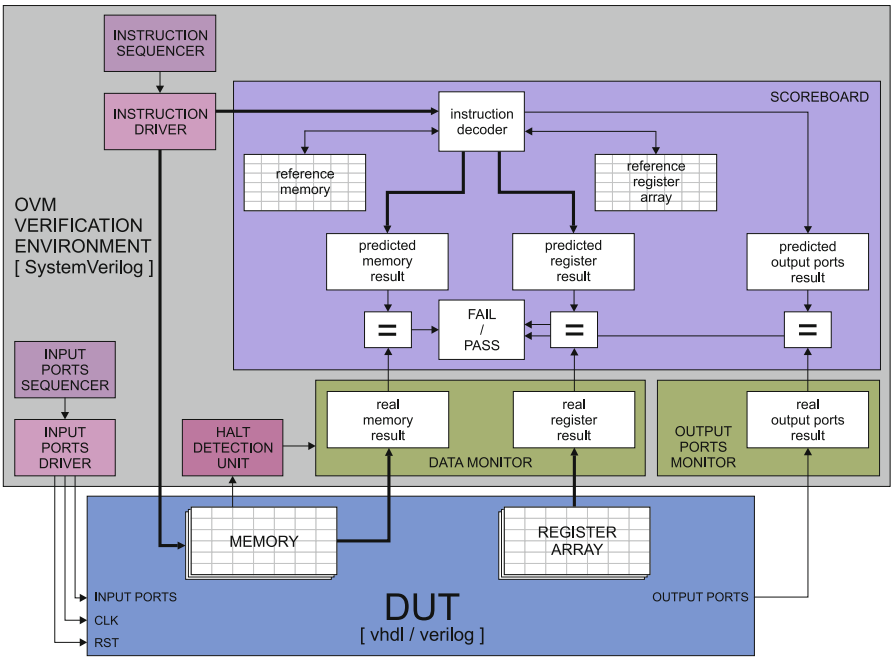

Fig. 3. OVM verification environments for user-defined processors in Codasip framework

- **DUT (Device Under Test)**. The verified hardware representation of the processor written in VHDL/Verilog. According to the type of the processor, different number of internal memories and register arrays is used. The processor typically contains a basic control interface with clock and reset signal, an input interface and an output interface.
- **OVM Verification Environment**. Basic classes/components of generated verification testbenches with compliance to OVM methodology:
  - **Input Ports Sequencer and Driver**. Generation of input sequences and supplying them to input ports of the DUT.
  - **Instruction Sequencer and Driver**. Generation of input test programs or reading already prepared benchmark programs from external resources. Afterwards, programs are loaded to the program memory of the processor as well as to the instruction decoder in Scoreboard.

- **Scoreboard**. This unit represents a self-checking mechanism for functional verification. In order to prepare expected responses of the verified processor to a particular program, Scoreboard uses a set of pre-generated reference functions created with respect to the specification described in the high-level CodAL language. For computational purposes a reference memory and a reference register array are used. As a result, predicted memory result, predicted register array result and predicted output ports result are prepared.
- **Halt Detection Unit**. It is necessary to define a specific time in simulation when images of memories and register arrays of processor are checked. Evidently, it should be done when a program is completely evaluated by the processor. This situation can be distinguished according to the detection of HALT instruction activity in the processor.
- **Data Monitor**. In case of the detection of HALT instruction activity, Data Monitor reads images of memories and register arrays of the processor and sends them to Scoreboard where they are compared with expected responses. If a discrepancy occurs, verification is stopped and a detailed report with description of an error is provided.
- **Output Ports Monitor**. Output ports of DUT are driven and their values are stored for later processing. In contrast with Data Monitor, this monitor works continuously not only in case of HALT instruction activity. Values generated by a reference model in Scoreboard are stored as well. Equivalence between stored values is checked by a function given by user or by default one.
- **Subscribers**. The aim of these units is to define functional coverage points, in other words interesting scenarios according to the verification plan which should be properly checked. These units are not present in Figure 3 although they are generated in every verification environment.

5 Experimental Results

In this section, the results of our solution are provided. We generated verification environments for two processors. The first one is the 16bits low-power DSP (Harward architecture) called Codea2. The second one is the 32bit high performance processor (Von Neumann architecture) called Codix. Detailed information about them can be found in [3]. We used Mentor Graphics' ModelSim SE-64 10.0b as the SystemVerilog interpreter and the DUT simulator. Testing programs from benchmarks such as *EEMBS* and *MiBench* or test-suites such as *full-retval-gcctestuite* and *perrenial testsuite* were utilised during verification. The Xilinx WebPack ISE was used for synthesis. All experiments were performed using the *Intel Core* 2 *Quad* processor with 2.8 GHz, 1333 MHz FSB and 8GB of RAM running on 64-bit *Linux* based operating system.

Table 1 expresses the size of processors in terms of required *Look-Up Tables* (LUTs) and *Flip-Flops* (FFs) on the Xilinx *Virtex5* FPGA board. Other columns contain information about the number of tracked instructions and the time in seconds needed for generation of SystemVerilog verification environment and all reference functions inside the golden models (Generation Time). In addition, the number of lines of programming code for every verification environment is provided (Code Lines). A designer typically

needs around *fourteen days* in order to create basics of the verification environment (without generation of proper stimuli, checking coverage results, etc.), so the automated generation saves the time significantly.

Table 1. Measured Results

Processor	LUTs/FF (Virtex5)	Tracked Instructions	Generation Time [s]	Code Lines
Codea2	1411/436	60	12	2871
Codix	1860/560	123	26	3586

Table 2 provides information about the verification runtime and results. As Codea2 is a low-power DSP processor some programs had to be omitted during experiments because of their size (e.g. programs using standard C library). Therefore, the number of programs is not the same as in case of Codix. Of course, the verification runtime depends on the number of tested programs and if the program is compiled with no optimisation the runtime is significantly longer.

Table 2. Runtime statistics

Processor	Programs	Runtime [min]
Codea2	636	28
Codix	1634	96

The coverage statistics in Table 3 can show which units of the processor have been appropriately checked. As one can see, the instruction-set functional coverage reaches only around fifty percent for both processors (i.e. a half of instructions were executed). The low percentage is caused by the fact that selected programs from benchmarks did not use specific C constructions which would invoke specific instructions. On the other hand, all processors registers files were fully tested (100% Register File coverage). This means that *read* and *write* instructions were performed from/to every single address in register files. The functional coverage of memories represents coverage of control signals in memory controllers. Besides functional coverage, ModelSim simulator provides also code coverage statistics like branch, statement, conditions and expression coverage. According to the code coverage analysis we were able to identify several parts of the source code which were not executed by our testing programs and therefore we must improve our testing set and explore all coverage holes carefully.

Table 3. Coverage statistics

Processor	Code Coverage [%]				Functional Coverage [%]		
	Branch	Statement	Conditions	Expression	Instruction-Set	Register File	Memories
Codea2	87.0	99.1	62.3	58.1	51.2	100	87.5
Codix	92.1	99.2	70.4	79.4	44.7	100	71.5

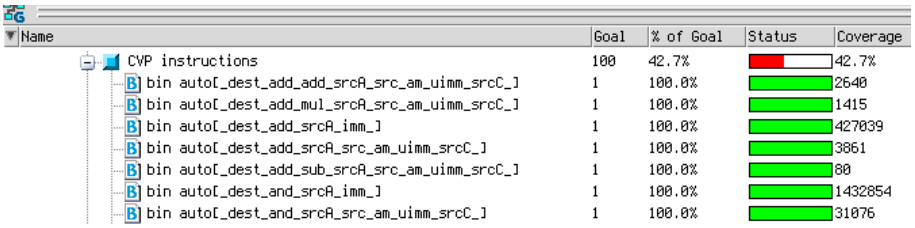

Fig. 4. Coverage Screenshot

Figure 4 demonstrates the status of instruction-set functional coverage for Codix processor after execution of 500 programs in ModelSim.

Of course, the main purpose of verification is to find bugs and thanks to our pre-generated verification environment we were able to target this issue successfully. We discovered several well-hidden bugs located mainly in the C/C++ compiler or in the description of a processor. One of them was present in the data hazard handling when the compiler did not respect a data hazard between read and write operation to the register file. Another bugs caused jumping to incorrectly stored addresses and one bug was introduced by adding a new instruction into the Codix processor description. The designer accidentally added a structural hazard into the execute stage of the pipeline.

6 Conclusion and Future Work

To summarise, implementation of functional verification environment is a manual and highly error prone process. As we wanted to accelerate creation and maintenance of advanced OVM verification environment for ASIPs we implemented a special feature which allows their automated generation. The experimental results show that the automatic generation is fast and robust and we were able to find several crucial bugs during the processors design.

In the future we plan to utilise a sophisticated generator of programs in order to achieve higher level of coverage of verified processors because during experiments we identified several holes in functional coverage and code coverage. Moreover, we want to discover the relation between test-templates and coverage points.

References

1. Fauth, A., Van Praet, J., Freericks, M.: Describing instruction set processors using nML. In: Proceedings of European Design and Test Conference, Paris, pp. 503–507 (1995) ISBN 0-8186-7039-8
2. Hoffmann, A., Meyr, H., Leupers, R.: Architecture Exploration for Embedded Processors with LISA. Springer (2002) ISBN 1402073380
3. Codasip Framework. Codasip (2012), http://www.codasip.com/
4. Martin, G., Bailey, B., Piziali, A.: ESL Design and Verification: A Prescription for Electronic System Level Methodology (Systems on Silicon). Morgan Kaufmann (2007) ISBN 0123735513

5. Mentor Graphics Verification Academy. UVM/OVM (2012), https://verificationacademy.com/topics/verification-methodology
6. Mishra, P., Dutt, N.: Processor Description Languages (Systems on Silicon), vol. 1. Morgan Kaufmann (2008) ISBN 9780123742872
7. Paradigm Works SystemVerilog Frameworks Template Generator (2012), http://svf-tg.paradigm-works.com/svftg/
8. Processor Designer (2012), http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx
9. Azevedo, R., et al.: The ArchC architecture description language and tools. International Journal of Parallel Program 33(5), 453–484 (2005) ISSN 0885-7458
10. Synopsys. Pioneer NTB (2012), http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/Pioneer-NTB.aspx
11. SystemC Project (2012), http://www.systemc.org/home/
12. Target (2012), http://www.retarget.com/
13. The LLVM Compiler Infrastructure Project (2012), http://llvm.org/
14. Přikryl, Z.: Advanced Methods of Microprocessor Simulation. Information Sciences and Technologies, Bulletin of the ACM Slovakia 3(3), 1–13 (2011) ISSN 1338-1237