Improving architecture efficiency of SoftCore processors

Bertrand LE GAL (1), Christophe JEGO (1)
(1) IMS Laboratory - UMR CNRS 5218,
IPB / ENSEIRB-MATMECA, University of Bordeaux, France
{firstname.surname}@ims-bordeaux.fr

1. INTRODUCTION

The growing complexity of applications increases the challenge of the System-On-Chip design. The most efficient way to produce area and power efficient circuits is to design fully dedicated architectures for ASIC or FPGA technologies. However, such approach is unsuited with the "time to market" constraint due to times for design and verification. Moreover, this approach has a lack of flexibility: a modification in the application specification can require new design flow iterations. Another way to implement applications under "time to market" pressure is based on high-end processor usage. Nowadays, *General-Purpose Processors (GPP)* or *Digital Signal Processors (DSP)* provides high computation performance. Moreover, programming languages and compiler tools provide high flexibility degree for these approaches. However, general processors are not relevant in the embedded market.

There are another architectural solutions that combine design, flexibility and time to market. One of these approaches consists to adapt low-cost GPP cores by adding domain specific instructions. Such processor architectures named *Application-Specific Instruction-set Processors (ASIP)* efficiently improve application performances (throughput, latency, etc.) thanks to dedicated hardware instructions.

Several ASIP design automation flows [1, 2, 3] and ASIP case of studies [4, 5, 6, 7, 8] have been proposed. They work on processor customization, identify and instantiate custom instructions in the processor. In this paper, we present an alternative approach. It enables to produce more efficient architectures in terms of hardware complexity. The objective is to obtain area cost reduction compared to original softcore processor. To achieve this objective, application's source code is first analyzed to identify useless processor instructions. Based on this instruction analysis, parts of the processor (useless ones) are removed before the logical synthesis process.

Study presented in this article is based on the SPARC v8 processor for two reasons. The first one is that the ISA is free of patents and thus the customization of the processor instruction set does not need licenses. The second one is that VHDL source codes of the SPARC v8 architecture called LEON-3 are available under the GPL/LGPL licenses. Processor source code allows us to modify the processor architecture according to the proposed methodology and to validate the modifications on FPGA boards.

The paper is structured as follows. In Section 2, we present some ASIP designs, ASIP methodology and discussion about presented work motivations. Section 3 presents the proposed design methodology used to reduce the instruction set. Finally, Section 4 gives experimental results with signal processing benchmarks on ACTEL ProASIC-3 target.

2. RELATED WORKS

A. Processor customization (ASIP)

Nowadays, most application domains - signal and video processing, digital communications, cryptograph, etc. - need fast implementations of their algorithms to respect real time execution. Some works have study these applications and new instruction sets have been proposed to speed-up application execution. In [4, 5] authors have proposed new multimedia instructions to increase execution of video applications. Similar works [6, 7, 8] tried to increase cryptography or digital communication applications. All these works are based on hand made application analysis and hand made VHDL description.

To speed-up the ASIP design time, automatic design flows have also been proposed [1, 2, 3]. Two main automatic ASIP design flows can be cited:

- First one is based on the complete description of the processor core [2]. In such methodology a designer has to fully describe the processor core using a dedicated HDL language. Then a set of automated tools generates the processor description in a HDL language such as VHDL and a complete set of development tools. Using the processor description and the automatically generated compiler, the designer can execute its application on a fully customized processor. This approach enables efficient implementation. But it is complex because the processor core have to be fully described.
- Second one is based on the reuse of a softcore processor [1, 3]. The main idea is to reuse an existing processor core by adding some custom instructions. Associated automatic methodologies perform (Figure 1): instruction

identification from application source code, instruction code generation (VHDL), processor modification and compiler tool chain adaptation. All these steps are specific problems. In practice, for real-life applications, heuristic based techniques are necessary. However these methodologies focus only on processor enrichment, in particularly they never try to optimize the "existing processor core" according to the application requirements.

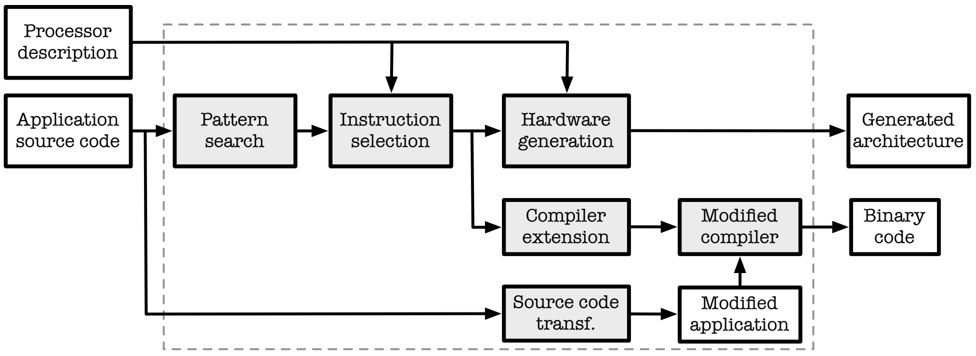

Fig. 1. Design flow for softcore processor customization

B. Work motivation

Hand-made and automatic ASIP methodologies help designers to generated dedicated processors. However, reused processor cores already contain instruction set, functional units, etc. All these resources may become useless for a specific application. These useless resources have drawback effects on processor enrichment possibilities and consequently on processor architecture performances:

1. If reused softcore processor has a large instruction set, only a few custom instructions may be added. In the Leon-3 ISA, there is only 8 free instructions (for arithmetic and logic instructions). This ISA constraint reduces drastically the design space exploration and performances.
2. After the softcore customization process, custom instructions are added to the processor architecture. Due to these custom instructions, some parts of the processor (instructions, functional units) may become useless.

For these reasons, we proposed an automated design flow that removes useless instructions and associated hardware resources from softcore processors. This approach enables:

- A better design space exploration because more instructions can be added in the processor ISA,
- A decrease of architecture cost in terms of hardware complexity for the final ASIP.

2. INSTRUCTION SET REDUCTION METHODOLOGY

A. Optimization methodology for softcore processors

The main objective is to study the application characteristics to (1) identify the useless parts of the processor: instructions, functional units, registers, etc. (2) to automatically remove them from the softcore hardware description. Proposed methodology for this task is shown in Figure 1. The application source code is first transformed into its binary representation using a software compiler dedicated to the processor target (like gcc). The generated executable file produced by the compiler contains the instructions that can be used by the softcore processor to execute the application behavior.

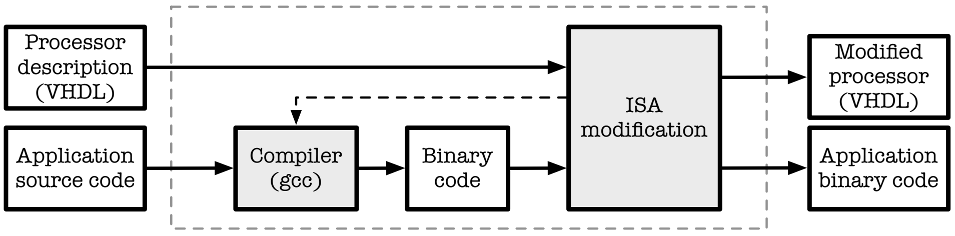

Fig. 2. Proposed design flow for useless instruction detection and hardware removing

The softcore transformation process that enables to reduce the hardware complexity is composed of the following steps:

- Executable file is first disassembled and dead code (unused functions) are removed.
- Instruction occurrences are counted and useless instructions (instruction that are never used) are identified.
- Definitions of useless instruction are removed from the compiler source code for safety reason. Indeed if a designer modifies its application and uses removed instructions, the compiler will generate a compilation error.

- The hardware description of the softcore processor is modified. Source code lines that correspond to instruction processing or functional unit allocation are removed from the source code. By this way, logical synthesis tool won't allocate useless logic and register resources.

Finally, a dedicated version of the processor softcore requiring less hardware resources is obtained. This generic design flow reduces the hardware cost of processors. This technique can be used for softcore processors with or without custom instructions. However, it is important to note that instruction set reduction breaks the processor compatibility with other applications that require some of the removed instructions.

B. Optimization methodology in an ASIP environment

Design flow presented in previous section is applied just before logic synthesis. However, as described in the motivation section, removing instruction from a softcore processor may offer more optimization opportunities during the customization step. The global design flow including the processor customization and the instruction removing stages becomes iterative as presented in Figure 3.

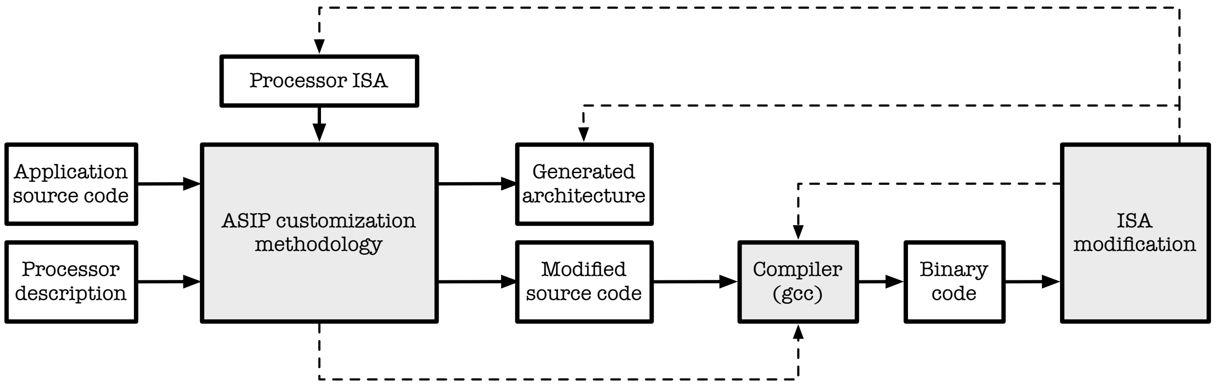

Fig. 3. Global design flow for well-sized ASIP design

Instruction and resource-removing step is executed after the ASIP customization. If the ASIP customization process is stopped due to a lack of free instruction slots, then the processor customization process can iterate after instruction and resource removing.

C. Improving unless instruction detection

The proposed methodology is based on application binary executable file obtained after source code compilation. This generic approach is automatic and need a low computation time. However, obtained results are not optimum. The binary executable file contains the compiled application is composed of:
- A trap table i.e. for the SPARC processors, gcc automatically generate a trap table which is used in software to catch processing errors (divide by zero, memory access error, unknown operation code, etc.).
- Generic low-level functions used for processor and peripheral initialization. These functions and instructions are defined in standard and system libraries. They are automatically included during the linking step.
- The dead code sections. They can be user-defined functions or functions declared in included libraries (stdio, math). This dead code is not removed in a standard compilation process.

These unwanted code pieces could contain floating-point instructions even if the application source code was compiled using software implementation of floating point operations. In practice it won't generate some errors during the application execution because they are unreachable code sections. However we do not have such information during the executable file analysis process. To improve instruction usage detection, a second method is available for processor execution trace files. In particularly, unused instruction detection is improved because unreachable code sections are discarded.

3. EXPERIMENTAL RESULTS

Experimentations given in this section were done from a custom version of the Leon-3 processor. This custom version (fully compatible with the SPARC v8 specification) was modified to speed-up custom instruction insertion. Modified Leon-3 architecture currently up to 1024 custom instructions. Custom instructions can be combinatory (executed in one clock cycle), pipeline or sequential (executed in more than one clock cycle) or asynchronous (working like an independent co-processor). A set of GPL software configuration tools enables to automatically configure the VHDL architecture description. Moreover, this design flow automatically adapts the compiler (GCC) source files to enables custom instruction use.

The experimental results for a Leon-3 softcore processor demonstrate the interest of the proposed approach. To verify the methodology independence from the application domain, a benchmark composed of 26 applications was used. This large set of domains targets cryptography, video and signal processing, digital communications and control based applications. The various application domains require different instructions sets. Most of the application source codes come from literature benchmarks [9-10, 12] and open-source projects like the Helix MP3 decoder [11]. The latest ones are hand made applications that were validated during previous research projects. This application list is detailed in Table 1. Information on application characteristics, operation requirements and source code origins are listed.

Type	#	Application	Source	Complex instr.
Digital communication applications	1	ADPCM enc./dec.	MiBench [9]	Mult, Shift
	2	BCH (31,21) enc./dec.	[12]	Shift, Div
	3	GOLAY enc./dec.	[12]	Shift
	4	LDCP decoder	Hand written	Shift
	5	MP3 decoder	Helix [11]	Mult, Shift, Div
Cryptographic applications	6	CRC 32b	TRAP [10]	Shift
	7	MD5	Fast DES kit	Shift
	8	SHA-1	MiBench [9]	Shift
	9	SHA-2	OpenSSL lib.	Shift
	10	ARC4 enc./dec.	OpenSSL lib.	Shift
	11	DES / 3-DES enc./dec.	Fast DES kit	Shift
	12	AES (128/512) enc./dec.	OpenSSL lib.	Shift
Control applications	13	Engine control	TRAP [10]	Mult, Shift, Div
	14	Data sorting (bubble)	TRAP [10]	Shift
	15	Queens	TRAP [10]	Shift
	16	Pattern matching (text)	TRAP [10]	Shift
	17	Text compresion (v42)	TRAP [10]	Mult, Shift, Div
	18	g3fax - fax reception	TRAP [10]	Shift
	19	LCD controller	Hand written	Shift
Signal processing applications	20	LMS filter processing	Hand written	Mult, Shift, Div
	21	FIR filter processing	TRAP [10]	Mult, Shift
	22	FFT & iFFT (fixed p.)	FFTW src	Mult, Shift
	23	Echo cancellation	LibGSM	Shift, Div
Video processing applications	24	JPEG decoder	TRAP [10]	Mult, Shift
	25	Motion detection	Hand written	Shift
	26	Contrast egalization	Hand written	Mult, Shift, Div

Table 1. Application set used for methodology evaluation.

Validation of the optimized processor designs for these applications has been done by simulation using the ModelSim tool. For each couple (optimized processor core, application), assembler execution traces and functional results have been compared to ones obtained using the initial processor core.

Results[1] provided in Figure 4 shows that more than half of the instruction set of the LEON-3 processor is never used for the different applications that have been investigated. However, as explained in the previous section, we can see that in function of the analysis technique (using the binary executable or profiling information), the useless instruction detection does not provide the same results. Indeed, the binary executable analysis detects from 20% to 58% (average = 52%) of useless instructions. These detection performances are lower compared to the profiling based ones (minimum = 58%, maximum = 74%, average = 69%). However they are obtained more quickly without complex application profiling requirements. If some instructions are never used, most instructions are necessary: cryptographic applications exclusively required logical instructions to the detriment of arithmetic ones though for video applications it is the opposite. Finally, for applications {5, 9, 12}, the assembler analysis step does not generate optimal results. Indeed, these applications contain array of constants. These arrays are transformed into "wrong" processor instruction during the disassembling process. These false instructions produce sub-optimal results during the useless instruction analysis.

Thus, using this knowledge it becomes possible to efficiently reduce the processor cost in terms of hardware resources by removing from the processor architecture useless parts. Hardware saving estimation is obtained after logic synthesis using the Synopsys Synplify Pro tool with an ACTEL ProASIC 3 device. Figure 5 shows the design area in terms of ProASIC elements required.

[1] X-axis values correspond to the application number provided in Table 1

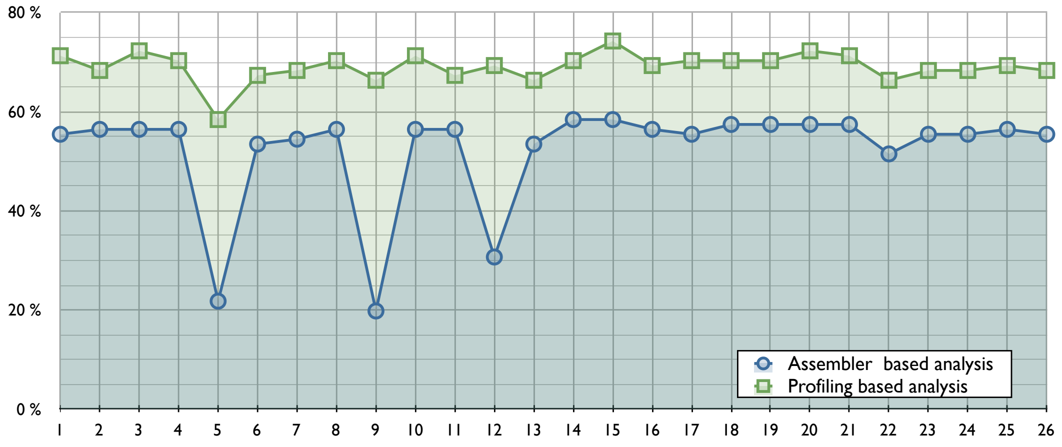

Fig 4. Useless-instruction rate depending on the applications (LEON-3 processor)

Gain in terms of hardware complexity obtained for an ACTEL ProASIC device (Figure 5) is at least 6% for the complete MP3 decoder application and grows up to 48% for the Queens application. Estimated gains for the overall benchmarks are equal to is 33% while considering the profiling-based analysis and to 28% for the assembler-based one.

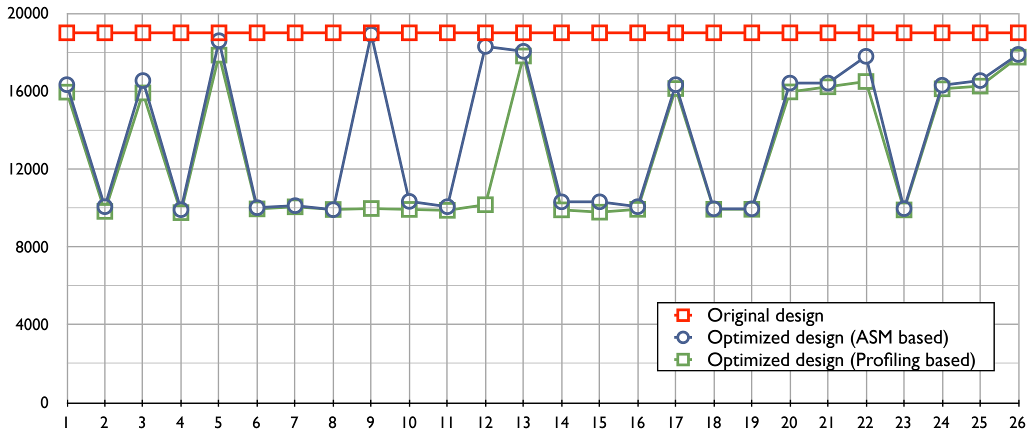

Fig 5. Experimental results (Leon-3 softcore synthetized on a ACTEL ProASIC device)

The most important part of the decrease is due to the costly ALU removing (multiplier, divider and barrel shifter resources). However hardware gains are not only obtained for applications that do not use these resources. Indeed, if we only consider applications that require multiplication, division and shifting resources the average hardware gain is 12%. This interesting result is only due to low-complexity resource removing (instruction decoding logic, etc.).

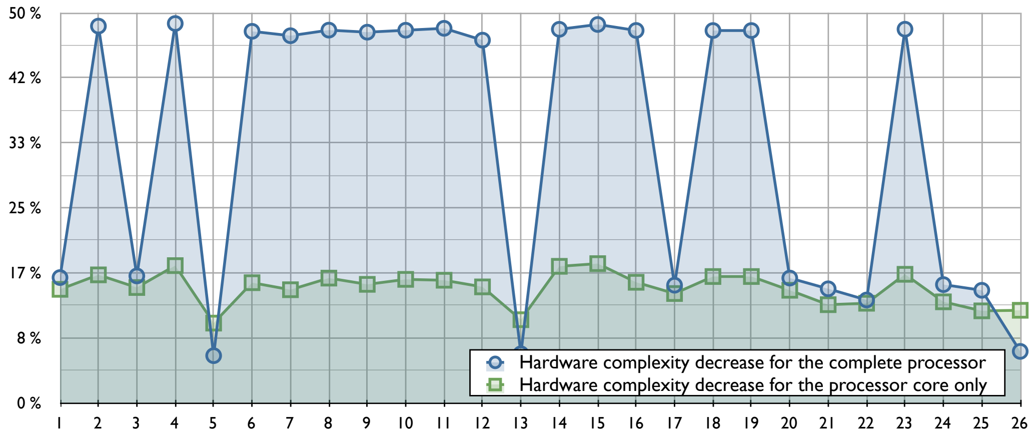

Fig 6. Comparison of hardware gains on the processor core.

Figure 6 shows the hardware gain obtained on the processor core (without the multiplier, divider and shifter ALUs). Results demonstrate the efficient hardware complexity reduction obtained for the processor core (instruction decoding stage, exception management, etc.).

Finally, hardware decrease saving obtained using the proposed methodology also impacts the power consumption of the processor core. Indeed, removing hardware resources in the processor core reduces its power consumption. A lower hardware complexity generates a lower static consumption. By this way, the hardware complexity reduction in the instruction decoder, the pipeline controller, and the exception manager enables to reduce the switching activity during processor execution. This switching reduction minimizes the dynamic power consumption of the processor core.

4. WORK PERSPECTIVES

In this work, we suppose that the processor optimization is performed thanks to a direct methodology. The optimization process is performed after the software compilation step. The extension of this approach to an iterative one may improve its efficiency. Indeed, software compilers were designed to use efficiently the overall processor instruction-set. Processor architecture is considered as fixed irrespective of the generated executable code.

This assumption that can be verified for general-purpose processor and DSP ones helps developers to reduce the algorithm complexity in software compilers. However, coupled with the proposed methodology, this can lead to sub-optimal results in terms of hardware complexity. This sub-optimality is due to the fact that optimized processor hardware complexity depends on the instruction-set used in software application. The reduction of the instruction-set used by the software compiler may reduce the processor hardware cost.

Finally, optimization techniques used in software compiler may lead to an increase of instruction-set usage. For example, during the compilation processor, a software compiler may decide to transform some assembler instructions to improve execution speed: transforming a factor 2 multiplication by a left-shift operation. This usual optimization involves to a processor complexity increase. Let's consider an application requiring only a multiplier and an adder resources. Transforming a constant multiplication to a shift-left operation increase the resource set (shifting resource) while this factor 2 multiplications may be implemented to the multiplier or the adder.

A modification of the software compiler flow to support and extend the proposed methodology was out of the scope of our current work. However, to improve the approach efficiency, software compiler aspects will be investigated.

5. CONCLUSION

In this paper, an original fully automated approach to reduce size of softcore processors (generic and ASIP ones) has been proposed. The methodology uses applications characteristics to remove useless parts of the processor core (instruction, hardware resources, etc.). Its objective is to minimize the hardware implementation cost. Proposed design flow does not required manual modification of the hardware description. Finally, results show that on commonly used applications, about half of Leon-3 instructions are useless. Removing them from the processor core provides area saving from 10% to 25% on the overall processor core (ALU, registers, control, etc.). Future works will target instruction models (more precision on hardware usage) and will consider ASIC targets.

REFERENCES

1. N. Clark, H. Zhong and S. Mahlke, "*Processor Acceleration Through Automated Instruction Set Customization*". In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36), 2003.
2. R. Klemm, J. P. Sabugo, H. Ahlendorf and G. Fettweis, "*Using LISATek for the Design of an ASIP core including Floating Point Operations*", Technical report, 2008.
3. R. R. Hoare et al., "*Rapid VLIW Processor Customization for Signal Processing Applications Using Combinational Hardware Functions*". EURASIP Journal on Applied Signal Processing, vol. 2006 ID 46472, 2010.
4. F. Tlili and A. Ghorbel, "*ASIP Solution for Implementation of H.264 Multi Resolution Motion Estimation*". In the International Journal of Communications, Network and System Sciences, Vol.3 No.5, May 2010.
5. P. Guironnet de Massas, P. Amblard and F. Pétrot, "*On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration*". Journal VLSI Design, vol. 2007, ID 28686, 2007.
6. S. Tillich. "*Instruction Set Extensions for Secret-Key Cryptography*". Poster Presentation at the Ph.D. Forum at the 9th Conference on Design, Automation and Test in Europe (DATE 2006), Munich, Germany, March 6, 2006.
7. F. Naessens, A. Bourdoux, A. Dejonghe, "*A flexible ASIP decoder for combined binary and non-binary LDPC codes*". 17th IEEE Symposium on Communications and Vehicular Technology (SCVT), 24-25 November 2010.
8. G. Kappen, L. Kurz, O. Priebe and T. G. Noll, "*Design Space Exploration for an ASIP/Co-Processor Architecture used in GNSS Receivers*". Journal of Signal Processing Systems, vol. 58 (1), pp. 41-51, 2010.

9. M. Guthaus et al., "*MiBench: A free, commercially representative embedded benchmark suite*", In Proc. IEEE 4th Annu. Workshop Workload Characterisation, Dec. 2001, pp. 3–14.
10. *TRAP benchmark suite for SystemC and TLM based Instruction Set Simulators (ISS)*, http://code.google.com/p/trap-gen.
11. *Helix MP3 Decoder* (open-source), Real Networks Inc. https://helixcommunity.org
12. S. Lin and D. J. Costello, "*Error Control Coding: Fundamentals and Applications*", 2nd edition, Prentice Hall: Englewood Cliffs, NJ, 2004.