Application Specific Processors for Multimedia Applications

Muhammad Rashid, Ludovic Apvrille and Renaud Pacalet
System-on-Chip laboratory
LabSoC, GET/ENST
Sophia-Antipolis, France
muhamad.rashid@thomson.net

Abstract

A well-known challenge during processor design is to obtain best possible results for a typical target application domain by combining flexibility and computational performance. ASIPs (Application Specific Instruction Set Processors) provide a tradeoff between generality of processor (flexibility) and its physical characteristics (computational performance and silicon area).

This paper evaluates an ASIP design methodology based on the extension of an existing instruction set and architecture described with LISA 2.0 language. The objective is to accelerate the ASIPs design process by using partially predefined, configurable RISC-like embedded processor cores that can be quickly tuned to given applications by means of ISE (Instruction Set Extension) techniques. A case study demonstrates the methodological approach for the JPEG algorithm and motion estimation encoding algorithm of H.264 encoding standard.

1. Introduction

An ASIP is an hardware architectural concept meant to fill the gap between ASICs (Application Specific Integrated Circuits) and DSPs (Digital Signal Processors). The formers are highly efficient but lack flexibility. On the other hand, software development on DSPs provide reusable and programmable solutions with less performance and energy inefficiency as compared to ASICs [1]. An ASIP is a microprocessor specialized for a given set of algorithms. By specialized, we mean that its instruction set is designed from scratch or extended from a known microprocessor. The programmability of ASIPs enables the designer to map multiple related applications as well as different generations (versions) of the same application on the same ASIP.

There may be two approaches to design ASIPs. The first approach is to design from scratch: an entirely new instruction set is specifically designed for the target application [2]. The second approach is to customize the instruction set of an existing general purpose partially predefined configurable processor [3] [4] [5]. In this paper, we have designed ASIPs by extending the instruction set of a 32-bit RISC processor. In each of the two approaches, the objective is to design customized processors in order to make the best application-dependent trade off between given criteria [6] such as:

- *Flexibility*: ASIPs should be designed to support at least minor variations of the implemented algorithm. For example, if an ASIP has been designed for a given version of a video decoding algorithm, it should also support next versions of the same decoding algorithms. But adding flexibility also certainly means adding complexity, therefore raising cost and silicon area.

- *System efficiency*: It implies computational power, area and cost.

By partially sacrificing silicon efficiency, configurable processors make ASIPs design more incremental and less complex, since both the hardware architecture and software tools are partially predefined [5].

The main contribution of this paper is to evaluate the LISATek ASIP toolkit (from CoWare) [7] [8] [9]. LISATek assists an ASIP design process by automatically generating the software tool suite (compiler, assembler, linker, simulator) as well as the RTL (Register Transfer Level) description of the designed processor. It covers all phases of the design process from algorithmic specification of the application down to implementation of the micro architecture. As a starting point for model creation LISATek provides a library of sample models which contains processors for different architecture categories like VLIW (Very Large Instruction Word), SIMD (Single Instruction Multiple Data), RISC (Reduced Instruction Set Computer). We extend the instruction set of a 32-bit RISC processor. The reason of 32-bit RISC core selection is the observation that many ASIPs tend to have a RISC-like core architecture and ISA (Instruction Set Architecture) [3]. Two well known applications from multimedia domain serve as case study: 1) The Motion Estimation sub-algorithm of H.264 encoding [13]. 2) The JPEG compression standard [12].

The rest of the paper is organized as follows: Section 2 introduces the evaluation methodology for LISATek design flow starting from the sample model. Section 3 describes ASIPs design for Motion Estimation algorithm in H.264 encoding. Section 4 describes ASIPs design for JPEG algorithm. Section 5 provides simulation and synthesis results. Section 6 comments on strengths and weaknesses of LISA-based design methodology. Section 7 describes related work and section 8 concludes the paper.

2 Evaluation Methodology for LISATek Design Flow

2.1 Evaluation Steps

To evaluate LISATek, we propose the following ASIP design methodology:

- The design flow starts by writing the application specifications in a high level language like C.
- Application specifications written in C are profiled to identify critical parts of the application. Criticality refers to computational intensive parts of the application.
- Customized instructions are identified for critical parts of the application to increase computational performance. These customized instructions are application specific instructions with a higher complexity than generic instructions like ADD, SUB, etc. We further explain this identification step in the paper.
- Customized instructions are integrated into a LISATek predefined configurable processor template to speed up the application.
- Customized instructions are functionally verified and simulated using an adequate instruction set simulator (ISS) generated by LISATek [7] [9]. The simulation also makes it possible to calculate the application speedup in terms of cycle counts.
- After simulation,, an RTL HDL model (VHDL or verilog) of target architecture is generated by LISATek from the corresponding LISA description. It triggers hardware synthesis process via standard logic synthesis tools. As a result, maximum clock rate and silicon area overhead is obtained for the selected CMOS target library.

2.2 Presentation of the Toolkit: Coware LISATek

The LISATek-based processor design flow [7] [9] covers all phases of the design process from algorithmic specification of the application down to implementation of the micro architecture. It improves flexibility of modeling target architectures and significantly reduces description efforts. It provides high level of flexibility to facilitate the description of various processors, such as SIMD, VLIW and RISC type architectures. To describe ASIPs, LISATek is based on a language called LISA 2.0. LISA offers two main features:

- The description of the ASIP structure: registers, pipeline structure, instruction set binary coding, instruction set syntax, etc.
- The description of the behavior of each instruction. This behavior is described with a pseudo C language.

The two main ASIP development phases of the LISA 2.0 based design flow are shown in figure 1. On the left hand side of the figure 1, the architecture exploration phase with the software development tool generation is visualized. On the right hand side, the implementation phase is shown that starts with the automatic creation of an RTL model of the ASIP. These phases are iterative and repeated until a best fit between selected architecture and target application is obtained. Every change to architecture specification requires a completely new set of software development tools. (i.e. C compiler, assembler, linker, simulator). This iterative exploration approach demands very flexible, retargetable software development tools to optimize computational performance, flexibility and silicon area.

2.3 Sample Architecture: 32-bit RISC processor

LISATek provides a library of sample models for different architectural categories. Our case study relies on the LISATek 32-bit RISC processor. The sample architecture has following characteristics:

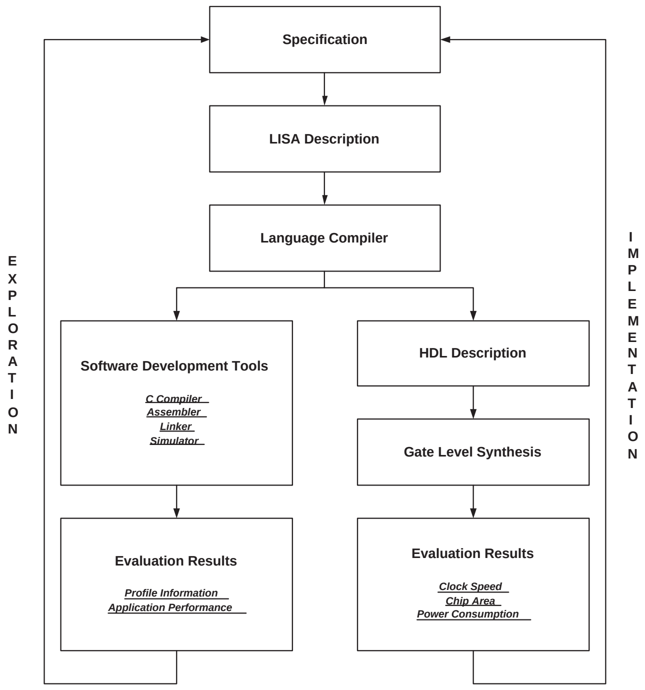

Figure 1. LISATek Design Flow

1. 32-bit instructions with five stage pipeline. (FE, DC, EX, MEM, WB)
   - (a) FE: To fetch instructions from memory.
   - (b) DC: To decode instructions for the next stages (EX, MEM, WB).
   - (c) EX: To execute operations.
   - (d) MEM: To store results in memory.
   - (e) WB: To write results back into registers.
2. Sixteen 32-bit general purpose registers.
3. PC register, Status registers, Pipeline registers and Bypass registers.
4. Six functional units (ALU , Control, DSP, LDST, Shifter and Writeback).
   - (a) ALU: To perform arithmetic and logical operations.
   - (b) Control: To perform branching operations.
   - (c) LDST: To perform load and store operations.
   - (d) Shifter: To perform shift operations.
   - (e) DSP: To perform DSP oriented operations.
   - (f) Writeback: To perform write back operations.

The sample architecture has already a minimal instruction set. There are four types of instructions: Arithmetic and Logical instructions, Branch instructions, Compare instructions and Load/Store instructions.

3 First Case Study: ASIP Design for Motion Estimation Algorithm

To evaluate the strength of LISATek toolkit for ASIP designing, we have performed two case studies: 1) ASIP Design for the Motion Estimation subalgorithm of H.264 encoding process. 2)ASIP Design for the JPEG compression algorithm.

Motion Estimation is the most computationally intensive part in the entire H.264 encoder.

3.1 Application Specification and Profiling Results

The starting point of our design process is a profile step. Profiling results show that the key functional blocks of motion estimation are Half-Pixel Interpolation, Quarter-Pixel Interpolation and Sum of Absolute Differences (SAD) algorithm. Half-Pixel Interpolation takes a 16x16 macro block as input and outputs four macro blocks containing the calculated half-pixels. Quarter-pixel interpolation takes four Half-pixels macro blocks and returns one macro block. SAD takes two macro blocks: a current macro block and a reference macro block. SAD algorithm returns a table (called error table) containing the errors.

3.2 Customization of Sample Model for Motion Estmation

The LISATek RISC processor sample model is provided with an already-defined instruction set. We extend this instruction set by identifying and implementing customized instructions. These dedicated (customized) instructions are identified to accelerate the computational intensive parts of the application. However, an important challenge for these customized instructions is to accelerate the execution of computational intensive parts while being flexible enough to accommodate variations in the algorithm. There are large number of possible instruction set extensions and each set of extensions describes various levels of tradeoffs between flexibility and efficiency. In this paper, we describe only one possible set of instruction set extension.

3.3 Data Memory Organization

H264 [13] encoding needs to compute Half Pixels several times during different encoding steps. Therefore the first step is to compute Half Pixels and store them in memory. Then, SAD is performed for a complete picture. Quarter pixels are computed by a simple average between two numbers stored in memory. Moreover, to compute Half Pixels, the notion of Macroblocks has no importance. Therefore we will compute Half Pixels for the whole image instead of processing macroblocks.

3.4 Customized Instructions

- **HINTER Rdst, Rsrc1, Rsrc2** takes two source registers and one destination registers as parameters. It loads 4 pixels from the second operand (32-bits wide), and multiply each pixel by a coefficient previously loaded in registers. The result of each multiplication is added to an intermediate result loaded from the first source register. Due to 32-bit restriction, we can not pass all the coefficient registers as parameters. We therefore use general purpose registers of the sample architecture as coefficient registers.

- **HSHIFT Rdst, Rsrc1, Rsrc2** divides the result of *HINTER* by 32 (5 bit right shift) to obtain corresponding half pixels. Since each pixel is of 8-bits only, we store the pixels back into memory in the group of four. This instruction right shifts the result of *HINTER* loaded from *Rsrc2* and eventually saturate it. It then writes it to the buffer specified by *Rdst* as shown in figure 2. It also writes the contents of the previous buffer *Rsrc1* left shifted by one pixel (one byte).

- **CSHIFT2** shifts pixels from one register to another This instruction requires that the two involved registers are necessarily *R1* and *R2* (Two general purpose registers of the sample architecture). We may pass the two involved registers as parameters on the cost of additional complex bypass operations.

- **LVW Vdst, Rsrc1, #16immOffset** loads a 32 bit word stored in memory at address (*Rsrc1 + #16immOffset*) into the Vertical Register *Vdest*. The destination registers are necessarily *R1, .. R4*, since we are not able to pass all these registers as parameters due to 32-bit restriction.

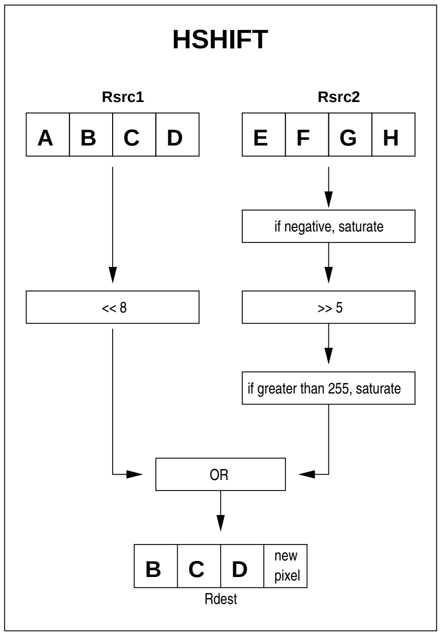

Figure 2. HSHIFT instruction dataflow

3.5 Modifications in Architecture

In order to implement customized instructions, LISA offers the possibility to either put the new instructions in a new *Functional Unit* or in (modified) existing *Functional Units*. In this paper, customized instructions are implemented by modifying existing *Functional Units*. The reason is that the sample architecture already contains some local registers in existing *Functional Units* that we can reuse for our purpose. Indeed, the creation of a new *Functional Unit* implies the creation of new local registers: already available local registers could not be reused. However, even in the existing *Functional Units*, some additional local registers are needed to implement the customized instructions.

We can notice that *LVW* instruction requires five operands: four operands to store the contents of the registers *R1*, *R2*, *R3* and *R4*, and one register to store the address of the word to be loaded from memory. So we have to add two pipeline registers. Moreover this instruction writes results back to memory, we therefore have to add three bypass registers and three writeback registers.

4 Second Case Study: ASIP Design for JPEG Algorithm

JPEG is a general purpose compression standard for still-image applications. The key functional blocks for

JPEG compression are FDCT (Forward Discrete Cosine Transform), Quantization and Entropy Encoding while key functional blocks for JPEG Decompression are IDCT (Inverse Discrete Cosine Transform), Dequantization and Entropy Decoding.

4.1 Application Specification and Profiling Results

The starting point of our design process is a profile step. To perform that profiling, we consider an open source C implementation of the JPEG algorithm [16]. The application code is profiled using the *gprof* GNU profiler [15] on a Pentium machine. Profiling results show that FDCT and quantization are the most computational intensive parts for compression, while IDCT and dequantization are the most computational intensive parts for decompression. 1 D LLM algorithm [19] computes DCT and IDCT with minimum number of operations. The flow graph of the 1 D (8 point) LLM algorithm is shown in figure 3. In figure 3, dots

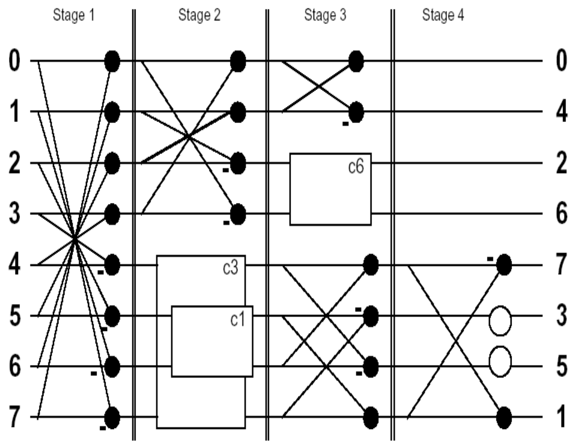

Figure 3. LLM Algorithm Flow Graph for DCT Computations

represent additions or subtractions. Hollow circles represent multiplication by a number. Rectangular boxes represent rotation and its computational cost is 4 multiplications and 2 additions. There are 4 stages in the LLM algorithm for DCT computation. Stage 1 consists of 8 additions/subtractions. In Stage 2, the algorithm splits into two parts. One part is for even coefficients (only additions and subtractions) and the second part is for odd coefficients (rotations). Stage 3 again splits into even and odd parts. The signal flow graph of LLM algorithm for forward and inverse DCTs are mirror images of each another.

4.2 Data Memory Organization

To evaluate the efficiency of customized instructions, the following data memory organization is defined for each input image block.

TmpAddress is used to store partial DCT results and is calculated as:

TmpAddress = (BaseAddress + 64)

Quantization table address *Tabaddress* is calculated as:

TabAddress = (TmpAddress + 128)

CoefAddress shows the starting address of quantized DCT coefficients.

CoefAddress = (TabAddress + 128)

FDCT and Quantization is performed in two steps: First 8x8 block is loaded from memory, 1 D DCT is performed and temporary results are stored into memory. Then these temporary results are retrieved back from memory to compute 1 D DCT column wise, and quantize the results before storing them in memory and so on.

4.3 Customized Instructions

Stage 1 of the LLM algorithm shown in figure 3 loads 8-bit pixels from data memory. This stage outputs 16-bit data. All the subsequent stages (stage 2, stage 3, stage 4) work with 16-bit data input and output. On the basis of the LLM algorithm flow graph and selected memory organization, we have explored various possible instruction set extensions. One possible set of instruction set extension is presented hereafter.

- **DCT01ROW Rsrc1, Rsrc2** computes DCT stage 1 of the LLM algorithm. It computes 08 additions/subtractions. It takes two source operand registers (*Rsrc1, Rsrc2*) and returns results in the registers which are implicitly defined inside the definition of this instruction. The use of implicit registers is due to the 32-bit instruction set restriction.

- **DCT02 Rdst1, Rdst2, Rsrc1, Rsrc2** computes DCT stage 2 of the LLM algorithm. It takes input from two source registers (*Rsrc1, Rsrc2*) and returns results in two destination registers (*Rdst1, Rdst2*).

- **ADDEVEN Rdst1, Rdst2, Rsrc1, Rsrc2** computes upper part in stage 2 and lower parts in stage 3.

- **DCTEVEN Rdst, Rsrc** computes upper part in stage 3 of the LLM algorithm.
- **DCTODD Rdst, Rsrc1, Rsrc2, Rsrc3** computes DCT computations in stage 4.
- **QUANTIZE Rdst, Rsrc1, Rsrc2** computes quantization. First source register stores 2 DCT coefficients (16 bit data). Second source register stores 2 quantization factor (16 bit data). It takes DCT coefficient to be quantized from first register and takes quantization factor from second source register, performs the quantization and stores the result in *Rdst*.

4.4 Architecture Modifications

We have added nine 32-bit registers and fourteen 16-bit registers. In order to execute customized instructions in pipeline, additional pipeline registers are also needed. We have added ten 32-bit registers and four 8-bit registers to pass data between pipeline stages.

4.5 Generation of HDL Code

The LISATek *Processor Generator*[8] [7] tool allows the designer to automatically create an implementation model of the target architecture modelled in LISA 2.0 language. The output of the *Processor Generator* is *VHDL* or *VeriLog* code, which can be processed by standard synthesis tools.

5 Experimental Results

The following experiments have been performed to evaluate the relevance of the proposed LISA-based design flow.

5.1 Simulation Results with Native Instructions

To compute 2D DCT of one input block with only the native instructions of the sample model, 640 cycles are required. Similarly 128 cycles are needed to compute quantization for one input image block of size 8x8. In addition to this, some cycles are consumed in loading pixels from memory, loading quantization coefficient from memory, storing and loading partial results to and from memory respectively and storing quantized DCT coefficients in the memory. It has consumed additional 240 cycles for one input block. Hence total number of consumed cycles are **640+128+240 = 1008**. Computational cost for IDCT and dequantization is same as that of FDCT and quantization.

5.2 Simulation Results with Customized Instructions

To compute 2D DCT of one input block with our ASIP (i.e. customized instructions), 192 cycles are needed. 240 additional cycles are required to compute one input block. So total number of consumed cycles are **192+240 = 432**. Computational cost for IDCT and dequantization is same as that of FDCT and quantization.

The designed architecture is not specialized for this application only. The customized instructions are reusable even if we change the DCT algorithm. Also, some of the instructions could also be reused for other algorithm (For example FFT algorithm).

5.3 Summary of Simulation Results

- The speedup due to new instructions is: **Speedup = 1008/432 = 2.33**. The speedup is obtained at the cost of silicon area. The increase in area is due to the additional pipeline registers and local registers.
- 2.33 is not the maximum possible speedup. The computational efficiency of the designed architecture (Minimum cycle counts) can be increased at the cost of silicon area (additional registers) and flexibility (More specific towards a single application) demonstrating the trade-off between reusability and efficiency.
- The entire design flow for the processor is performed beginning from the functional description of the application down to the hardware implementation within three man-weeks. This time also includes the creation of architecture simulators and production quality software development tools. This short development time demonstrates effectiveness of the design flow.

5.4 Synthesis Results

We have performed logic synthesis by means of Cadence Encounter RTL Compiler using a standard cell CMOS 0.13. The target frequency is 200MHz while external input and output delays are 2.5 ns.

The processor model instantiates three main subblocks: *PipeLine*, *RegisterFile* and *Memories*. The simulation memories are replaced with technology specific vendor memories.

Synthesis results for the entities *Pipeline* and *RegisterFile* are combined to get total area which is 42.5

k-gates at 160 MHz maximum frequency. For JPEG application, the minimum memory needs are:

- Program memory: The minimum size of program memory is the size of ASM code (assembly code) for the application.
- Data memory: The minimum size of required data memory depends on the size of stored image in memory.

6 LISATek Evaluation

6.1 Strengths of LISATek

- As a starting point of model creation LISATek provides a library of sample models which contains processors for different architecture categories. Taking such model as basis has a major advantage to directly have compiler support for the architecture due to the existence of an instruction set. This removes the entry barrier usually caused by new modeling languages and tools.
- It is quite easy to list a set of resources (memory, buses, registers). Operations are described in a hierarchical way, which facilitates reusability and modularity.
- Step by step simulation is quite useable.
- The toolkits *Processor Designer* and *Processor Debugger* [8] [9] creation have a good graphical user interface thus offering ways to design and debug the processor before the generation of its hardware description.

6.2 Weaknesses of LISATek

- Although toolkit *Processor Designer* and *Processor Debugger* has a good graphical user interface but design methodology still lacks the large degree of automation as compared to its counterpart like Tensilica [3] that has more automated approach.
- The LISA language analyzer is quite limited. It means that description errors may occur when compiling the simulation environment. In that case, we must understand gcc errors to correct the LISA description.
- *A*lthough VHDL code can be automatically generated from LISA source code by *Processor Generator* but generation process showed many errors in the generated *VHDL* code. So we have to modify the LISA code in order to remove those errors.
- LISATek profiler [10] provides detailed processor specific information. However, it is bound to specific architectures and not suitable for performance estimation in a general, target processor independent way. In [18], a tool has been propsed that estimates the cycles counts and memory profiles. However, it does not extract inherited spatial parallelism present in the application.
- LISATek based design methodology has no notion for modeling coarse grain reconfigurable architectures. The recent work in this regard is [14]. But it describes only fine grain reconfigurable architecture with static reconfiguration.

7 Related Work

The Xtensa [3] environment from Tensilica is built upon a choice between elements from a predefined set of hardware components which can be adapted to the user requirements. For this reason the design space exploration can be performed efficiently but the designer has not the flexibility of modeling arbitrary ASIPs. The PEAS-III [4] generates not only HDL descriptions but the target compiler and target assembler as well. However, it works with a set of predefined components which limits the resulting flexibility in modeling arbitrary processor architectures. The EXPRESSION [11] language allows the cycle-accurate processor description. It provides the mechanism for capturing the information needed to support ADL (Architecture Description Language) based design space exploration and software toolkit generation methodology. However, currently there is no information whether the implementation step can be done based on this language.

None of the introduced approaches provides the designer with efficient design exploration and implementation capabilities coupled with the required flexibility for the development of arbitrary ASIPs. In this paper, LISA 2.0 based design flow is evaluated to address these issues. We have used a manual approach where custom instructions are identified by the user after profiling. However, the readers are referred to a more automated approach in [5]. In this automated approach advanced profiling tools are used such that custom instructions are not identified by the user but generated automatically from the application code. For Customized instructions implementation, [5] relies on CorXpert (from Coware) tool. CorXpert is a graphical tool for capturing CI (Customized Instructions) of configurable processors. The use of CorXpert as a backend makes the design flow quite generic, since it uses the LISA as the specification formalism for the CI.

8 Conclusion

This paper evaluates LISA 2.0 based methodology to design ASIPs for multimedia applications. We have designed a processor architecture with an extended instruction set based on the profiling results. As far as area overhead and speedup are concerned, our solution is somewhere between pure software implementation and full custom designed ASIPs. Our case study has explored different types of ASIPs for Motion Estimation algorithm in H264 encoding and JPEG algorithm.

A major disadvantage of our approach is the lack of automation in identifying the customized instructions. Design time can be significantly reduced to few hours by making this process automatic. Future work may be the modeling of further real world processor architectures while focusing on the evaluation of efficiency of both the generated RTL code and the efficiency of retargetable C compiler.

Acknowledgments.

This work was sponsored by Texas Instruments Inc. The LISATek evaluation of the motion estimation subalgorithm was performed by Stephane Paris during his internship at LabSoc.

References

[1] P. Ienne and R. Leupers. *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[2] F. Haddad, L. Apvrille, and R. Pacalet. *Comparative Study of Toolkits for the fast Design of ASICs and ASIPs.* (Septembre 2005).

[3] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[4] S. Kobayashi, K. Mita, Y. Takeuchi, and M. lmai. Rapid prototyping of jpeg encoding using the asip development system: *PEAS-III.* In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Hong Kong, April 2003.

[5] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *DATE '06:*, pages 581–586, 3001 Leuven, Belgium, 2006.

[6] A. Hoffmann, F. Fiedler, A. Nohl, and S. Parupalli. A methodology and tooling enabling application specific processor design. In *VLSID '05*, pages 399–404, Washington, DC, USA, 2005.

[7] CoWare. *LISATek Creation Manual*, product version v2005.2.1 edition, February 2006.

[8] CoWare. *LISATek Methodology Guidelines for the Processor Generator*, product version v2005.2.1 edition, February 2006.

[9] CoWare. *LISATek Processor Designer Manual*, product version v2005.2.1 edition, February 2006.

[10] CoWare. *LISATek Profiler*, product version v2005.2.1 edition, February 2006.

[11] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99*, page 100, New York, NY, USA, 1999.

[12] T. international Telegraphic and T. C. Committee. Information technology - digital compression and coding of continuous tone still images - requirements and guidelines. Recommendation T.81.

[13] T. Wiegand et al. Overview of the H.264/AVC Video Coding Standard. In *IEEE Trans. Circuits and Systems for Video Technology*, vol. 13, no. 7, 2003, pp. 560–576.

[14] A. Chattopadhyay, W. Ahmed, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr. Design space exploration of partially reconfigurable embedded processors. In DATE '07, pages 1–6, Apr. 2007.

[15] J. Fenlason and R. Stallman. The gnu profiler.

[16] I. J. group. www.ijg.org.

[17] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with Lisa.* Kluwer Academic Publishers, 2002.

[18] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *DATE '06*, pages 468–473, 3001 Leuven, Belgium,2006.

[19] C. Leoffler, A. Ligtenberg, and G. Moschytz. Practical fast id dct algorithms with 11 multiplications. In *In Proc. IEEE ICASSP*, pages 988–991, Feb 1989.