FlexSoC: Combining Flexibility and Efficiency in SoC Designs

John Hughes‡, Kjell Jeppson†, Per Larsson-Edefors, Mary Sheeran‡, Per Stenström, Lars "J." Svensson

Computer Engineering, †Microelectronics, ‡Computing Science
Chalmers University of Technology, SE-412 96 Göteborg, Sweden

`http://www.cs.chalmers.se/~flexsoc`

Abstract

The FlexSoC *project aims at developing a design framework that makes it possible to combine the computational speed and energy-efficiency of specialized hardware accelerators with the flexibility of programmable processors.* FlexSoC *approaches this problem by defining a uniform programming interface across the heterogeneous structure of processing resources. This paper justifies our approach and also discusses the central research issues we will focus on in the areas of VLSI design, computer architecture, and programming and verification.*

1 Introduction

Dedicated, application-specific, single-task processing elements (or specialized blocks, or accelerators) can improve overall computation speed and energy efficiency when used in addition to general-purpose processor (GPP) cores in a System-on-a-Chip (SoC). Most SoC visions rely on such specialized blocks to carry out compute-intensive tasks, such as filtering, transforms, or encryption. Then, clear performance benefits in terms of processing speed and energy-efficiency are achievable when compared to software implementations hosted on GPP or digital signal processor (DSP) cores. These benefits may vanish, however, if a slight variation of the original algorithm is to be implemented instead. Thus, efficiency and programmability would seem to be at odds. Moreover, SoCs containing both GPP cores and accelerators do not interface well with traditional software development tools, such as compilers and debuggers. This makes software development challenging.

The objective of the FlexSoC research effort is to combine the efficiency of special-purpose hardware with the programmability offered by a GPP. We envision processors that simultaneously offer both programmability similar to that of a GPP and efficiency similar to that of special-purpose hardware. Our research will be directed towards *a homogeneous way to handle heterogeneous processor architectures,* including the following benefits, which are all taken for granted by users of traditional GPPs and DSPs:

- A semi-opaque processor architecture, which the programmer may, but does not have to, know in detail.
- A software-independent way to augment the underlying processor architecture for increased computational capacity.
- Unified mechanisms to disable unrequired hardware resources during periods of light computational load to save energy.
- Improved resource utilization by automatic re-use of hardware resources across unrelated computations.

The authors gratefully acknowledge the support of the Swedish Foundation for Strategic Research (SSF).

Our work will also provide a compact representation of the instruction stream, which will serve to reduce application memory-footprint and instruction-bandwidth requirements, leading to improved area and energy efficiency.

We will first outline the general approach of the FlexSoC project in Section 2. We then discuss the central research issues we plan to address and the methodological approach chosen in Sections 3 and 4, respectively, before we conclude in Section 5.

2 Approach

In traditional GPPs, a fixed Instruction Set Architecture (ISA) provides a hardware/software interface that is the same for *all* applications. For a heterogeneous SoC processor, the use of a single hardware/software interface is not expected to work well: since such an ISA would contain a significantly larger number of instructions, the resulting code would consume large amounts of memory and instruction bandwidth. Moreover, maintaining code compatibility would make it difficult to add new hardware structures. Furthermore, the fixed-ISA concept does not fit well with the heterogeneous processors foreseen for high-functionality SoCs. The specialized blocks used for filtering, encryption, and other compute-intensive tasks are rarely controlled directly via an instruction set but rather indirectly, through explicit configuration. Thus, the price paid for a fixed-ISA hardware/software interface in this context is inefficiency on both sides of the interface.

FlexSoC aims to mitigate these inefficiencies through a novel hardware-software interface concept. Our focus on processors embedded on SoCs makes it possible to depart from the traditional, compatibility-constrained ISAs. Thus, a FlexSoC processor has a *native* ISA (N-ISA), capable of fine-grain control of all computational resources,

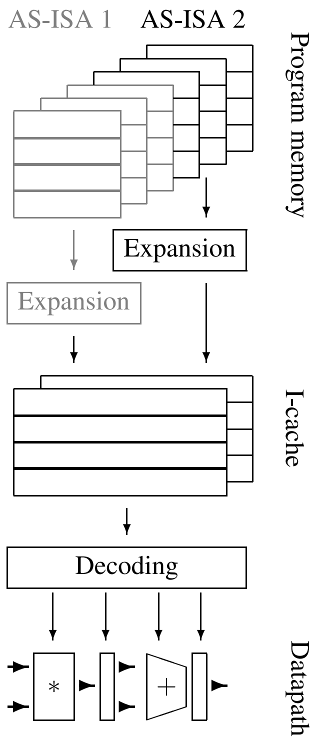

Figure 1. Instructions in two different AS-ISAs are expanded into the N-ISA in the I-cache.

as well as one or more *application-specific* ISAs (AS-ISAs), each of which could be tuned to a class of applications with similar computational requirements. Translation from each AS-ISA to the N-ISA is carried out on-the-fly by reconfigurable logic circuits. Thus, AS-ISAs can be defined to fit a new application; the N-ISA and the translation process stay fixed, providing the foundation on which the AS-ISAs are built.

To efficiently accomplish all their tasks, SoC processors will include special-purpose hardware blocks to support time-consuming computational kernels. The following observation has far-reaching consequences: *Each task will only use a small subset of the functionality offered by the complete processor.* A properly selected AS-ISA can therefore be much more densely coded than the N-ISA. In fact, a recent study [1] that characterized the instruction usage in Java bytecodes observed that only 45 out of 256 bytecodes account for 90% of the execution. This suggests that executable code will occupy less memory and require a lower instruction bandwidth using the FlexSoC approach. In fact, the selection of the AS-ISA adds another dimension to the optimization space available to the compiler: it may be preferable to use a sequence of instructions which are already available instead of a single instruction that would expand the AS-ISA.

A possible controller strategy to implement the AS-ISA-to-N-ISA translation is illustrated in Figure 1. To reduce the influence of the time spent to translate an application-specific instruction to a native instruction, our intention is to carry out this translation on instruction cache misses only, and cache the translated instructions.

Figure 1 shows only a simple translation case. Several extensions of this scheme readily suggest themselves:

- For a multi-level instruction cache, translation from the AS-ISA to the N-ISA could be distributed across each cache level. As a consequence, the two ISA levels would be replaced by a deeper hierarchy.
- If one AS-ISA instruction is allowed to correspond to a *sequence* of N-ISA instructions, the expressive power per byte in the AS-ISA grows even more, further improving instruction density. The allowable clock rates at different cache levels will naturally influence the maximum length of such instruction sequences.
- A single instruction cache hierarchy may support several instruction streams, in much the same way as a traditional GPP may share caches with a co-located DSP in a heterogeneous multiprocessor [2]. Provided that the synchronization issues can be handled, a FlexSoC processor can additionally re-use datapath blocks *across instruction streams*, as several AS-ISAs may map to the same N-ISA.

Since the N-ISA is completely hidden to the application code, it can be changed if functionality is added. Thus, new hardware functionality can be added without being constrained by a fixed-size ISA. The N-ISA can also provide a more expressive interface to the compiler. This can be exploited by offering more powerful primitives to control the heterogeneous processor with the goal of saving power and adapting resources to the application needs.

3 Research issues

While the hardware/software interface outlined above clearly offers flexibility to embedded heterogeneous processors, there are many open questions concerning how to implement the interface efficiently. These questions motivate research on both sides of the interface in a concerted fashion. We outline some of the issues below.

3.1 VLSI design issues

The success of the FlexSoC program depends on solving several challenging circuit-design and micro-architecture problems. The implementation of the instruction-cache hierarchy is quite different from that of most traditional GPPs and DSPs. First, instruction decoding is distributed across the memory hierarchy, with a minimum of two decoding stages. Second, the decoding circuitry itself is to be reconfigurable to support several AS-ISAs. Third, for fast AS-ISA swaps, several configuration-data sets should be stored as close to the decoding logic as possible. Fourth, if one AS-ISA instruction is allowed to correspond to a sequence of N-ISA instructions, multi-rate clocking is required.

Instruction decoding distributed across the first-level I-cache is a well-known technique. For example, modern implementations of the ubiquitous Intel architecture decode the x86 instructions to "micro-ops" (RISC-style instructions) which are then stored in the instruction trace caches. In FlexSoC, the technique is generalized to several levels of on-the-fly translation; selecting the right number of levels is an important design task. The "best" number of decoding stages will likely be found only through thorough architectural analysis and simulation, where detailed performance models of the critical VLSI blocks determine the optimum parameter values.

Several recent academic research efforts have focused on the use of reconfigurable logic in the datapaths of general-purpose processors (Zhou and Martonosi [3] present a good overview). The FlexSoC effort is complementary to these approaches, in that reconfigurability is restricted to the instruction stream processing, where latency can be partly hidden under cache misses.

Configuration-data storage is an important consideration. To load the data sets from main memory every time would reduce the "agility" of the processor—its ability to quickly adapt to a new workload. A high AS-ISA swap latency can be expected to result in AS-ISA bloat, since the compiler layer would attempt to extend an existing AS-ISA rather than introduce a new AS-ISA when needed. A natural approach seems to be to save the configuration data in the cache memory array itself, essentially treating it like other dynamic data, which will be loaded on demand, locked in memory if AS-ISA switch latency is important, and flushed from memory when no longer needed. Thus, instruction cache space can be seamlessly traded off for configuration data space through adaption of the common memory resource.

3.2 Computer architecture issues

On-the-fly translation, code compression, and avoidance of code expansion in using compiler optimizations all stress how to support the translation process in an efficient yet flexible way. FlexSoC will focus on this issue.

On-the-fly instruction translation finds many applications beyond the FlexSoC approach. One example is in supporting virtual machines in Just-In-Time (JIT) compilation of Java code [4]. In a JIT compiler, code is translated on the fly by software. In contrast, in our approach, most of the translation and optimization is done at compile time and a much less compute-intensive translation between AS-ISA and N-ISA instructions is postponed to run-time. Yet, a critical issue is to find schemes to support the on-the-fly translation process which impose the least possible overhead in terms of execution time and energy. As discussed above, it is desirable to carry out the instruction translation as close to the processors as possible as that will be most effective in preserving memory space and instruction bandwidth. However, the closer the translation is, the more execution overhead it can cause. One way to reduce the overhead would be to perform the translation in a speculative manner using branch prediction. The question is how effectively that can be done.

A second example of on-the-fly translation is when instructions are compressed in memory so as to reduce the memory space needed to store the program [5]. Since an AS-ISA instruction can represent a sequence of N-ISA instructions, the FlexSoC framework enables powerful compression schemes guided by static (compiler) and dynamic analysis (*e.g.,* profiling). The compiler could for example identify recurring code sequences [6] and semantically associate an AS-ISA instruction with them, just like a procedure call. When such an instruction is encountered, it is expanded so that it acts as if the procedure is inlined. The power of such compression approaches was recently demonstrated in the DISE project [7, 8]. DISE applies static analysis to the binary only; FlexSoC enables much more aggressive compression as the compiler can have full control of the coding.

Many compiler optimizations increase the static code size and the FlexSoC approach can afford them without this code expansion. In addition to the abovementioned procedure inlining, loop unrolling is another example. A loop to be unrolled could be associated with a special AS-ISA instruction. That instruction could in the translation process unroll the loop and load it into the first-level cache.

3.3 Programming and verification issues

The union of requirements of FlexSoC programmers is likely to be very large and difficult to fulfill in its entirety, just as is the case for GPPs. At one extreme, a programmer may insist on detailed control of hardware usage, and at the other extreme, the programmer may prefer the compilation system to make all hardware allocation decisions. The challenge is to provide a way to satisfy both these demands, and develop a programming paradigm where what can be automated well, is automated, and what cannot be, can be specified by the programmer in a high-level way.

We plan to develop this paradigm in a series of stages. The first stage is to develop a *configuration design language*, whereby processor configurations—the translation rules from AS-ISA to N-ISA—can be described by a programmer, analyzed for performance and power properties, and verified for correctness. The second stage is to use these configuration descriptions together with application code to generate AS-ISA code. In the third stage, we will automate appropriate aspects of the configuration problem: identify aspects suitable for automation, and find good optimization routines to handle those. We will consider using simulation feedback to direct the configuration selection, if appropriate.

Our approach to research this FlexSoC programming interface is to work with *domain-specific embedded languages,* implemented as libraries in the functional programming language Haskell [9]. The approach combines the advantages of a very-high-level programming language with fine low-level control, and above all enables us to experiment with the design without the overheads of developing a special compiler. We are following the approach of Elliott, Finne and de Moor's Pan system [10], which generates optimized C code for graphics transforms from high-level Haskell descriptions, or Claessen and Sheeran's Lava system [11], which generates FPGA configurations from Haskell descriptions.

The flexibility inherent in the FlexSoC approach to processor design means that performance trade-offs must be made across several levels of abstraction, including hardware and software levels. To verify the equivalence of the choices being considered, it will be necessary to develop ways to reason about processor configurations, that is, about the logic functions used to translate from an AS-

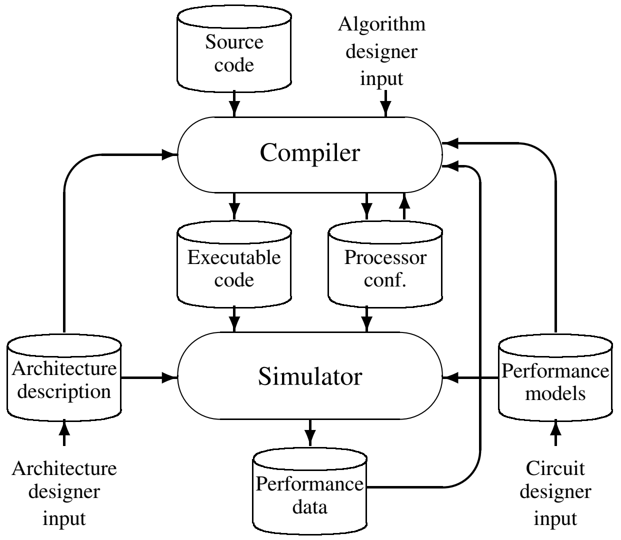

Figure 2. The experimental framework.

ISA to the N-ISA of the FlexSoC processor. The Lava system [12], developed at Chalmers and Xilinx, is coupled to a variety of formal verification tools, with particular emphasis on automatic verification using satisfiability solvers. Several extensions of the Lava system will be necessary to handle the full range of FlexSoC processors, such as those using multiple clock rates.

4 Methodological Approach

Our approach draws on novel cache-memory circuit design ideas; unique use of reconfigurable logic for instruction decoding; novel ways to distribute the instruction decoding logic across the datapath and memory subsystem; new principles for on-the-fly translation; and supporting methods for program compilation, transformation, and verification. An experimental framework (Figure 2), built around an *architecture-simulator core,* will allow us to capture and verify functional, performance, and power behavior of a FlexSoC processor. Parameterized *timing and power models for hardware blocks,* based on VLSI-design experiments, will provide the data necessary for performance evaluation. A *programming interface* will allow co-evaluation and co-verification of configurable hardware, configuration data, and the executable code.

In constructing the architecture-simulator core, we will draw on our experience in implementing highly efficient architecture simulation systems. In recent work [13], we have designed a simulation framework in which processor and memory architecture simulators can be constructed for functional, timing, and power modeling of processors at the register transfer level. We will extend this framework with a configuration module that models the on-the-fly translation architectural mechanisms.

The hardware-block performance models will be based on circuit simulations; selected blocks will be fabricated and tested to verify the analysis and to calibrate the simulations. The construction of abstract register-transfer level models from large amounts of simulation data is a topic of ongoing research within the group [14].

Our driving application examples will, in addition to GPP-like functionality, also include specialized hardware blocks to support time-consuming computational kernels. We intend to select examples similar to those found in the MediaBench suite [15].

5 Conclusions

The FlexSoC initiative addresses the efficient design of embedded programmable processors that include special-purpose datapath components together with the fundaments of the programmable processor. The problem complex has attracted much attention in academia as well as in industry. Compared with previous efforts, the FlexSoC initiative brings a unique focus on instruction-stream management and the cross-disciplinary competence needed to consider the problem in its entirety.

References

[1] Radhakrishnan et al. Java run-time systems: Characterization and architectural implications. *IEEE Trans. on Computers,* vol. 50, no 2, Feb 2001, pp. 131–146.

[2] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, vol 30, no 9, Sep 1997, pp. 79–85.

[3] X. Zhou and M. Martonosi. Augmenting modern superscalar architectures with configurable extended instructions. Presented at the *Reconfigurable Architectures Workshop (RAW 2000),* Cancun, Mexico. Available at `http://ipdps.eece.unm.edu/2000/raw/18000943.pdf`

[4] Cramer et al. Compiling Java just in time. *IEEE Micro*, vol. 17, no 2, March/April 1997, pp. 45–53.

[5] C. Lefurgy, E Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Proc. of 6th Int. Symp. on High-Performance Computer Architecture*, Jan 2000, pp. 218–227.

[6] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler Techniques for Code Compaction. *ACM Trans. on Programming Languages and Systems*, Vol. 22, No. 1, 2000, pp. 378–415.

[7] M.L. Corliss, E.C. Lewis and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proc. of 30th IEEE/ACM Symp. on Computer Architecture*, Jun. 9-11, 2003.

[8] M.L. Corliss, E.C. Lewis and A. Roth. A DISE implementation of dynamic code decompression. In *Proc. of the ACM/SIGPLAN 2003 Int. Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES-03),* Jun. 11–13, 2003.

[9] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse,* IEEE Computer Society, June 1999.

[10] C. Elliott et al. Compiling embedded languages. In *Semantics, Applications and Implementation of Program Generation workshop (SAIG 2000)*, Springer Verlag LNCS, 2000.

[11] P. Bjesse et al. Lava: Hardware design in Haskell. In *Proc. Int. Conf. on Functional Programming (ICFP'98)*, ACM Press, 1998.

[12] K. Claessen, M. Sheeran, and S. Singh. The design and verification of a sorter core. In *Proceedings of the International Conference on Correct Hardware Design and Verification Methods, CHARME'01*, Springer Verlag, LNCS, 2001.

[13] J. Chen, M. Dubois, and P. Stenström. SimWattch: An Approach to Integrate Complete-System with User-Level Performance/Power Simulators. In *Proc. of IEEE ISPASS-2003*, March 2003.

[14] D. Eckerbert and P. Larsson-Edefors. A Deep Submicron Power Estimation Methodology Adaptable to Variations Between Power Characterization and Estimation. In *Proceedings of Asia South-Pacific Design Automation Conference (ASPDAC),* Kitakyushu, Japan, Jan. 21–24, 2003, pp. 716–719.

[15] C. Lee, M. Potkonjak, W.H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of 30th IEEE Ann. Int. Symp. on Microarchitecture,* 1997, pp. 316–321.