

Instruction Set Selection for ASIP Design

Michael Gschwind*

mikeg@watson.ibm.com

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

We describe an approach for application-specific processor design based on an extendible microprocessor core. Core-based design allows to derive application-specific instruction processors from a common base architecture with low non-recurring engineering cost.

The results of this application-specific customization of a common base architecture are families of related and largely compatible processor families. These families can share support tools and even binary compatible code which has been written for the common base architecture. Critical code portions are customized using the application-specific instruction set extensions.

We describe a hardware/software co-design methodology which can be used with this design approach. The presented approach uses the processor core to allow early evaluation of ASIP design options using rapid prototyping techniques.

We demonstrate this approach with two case studies, based on the implementation and evaluation of application-specific processor extensions for Prolog program execution, and memory prefetching for vector and matrix operations.

1 Introduction

New application areas for microprocessor-based systems have diverse and often conflicting requirements on the microprocessors at the core of these systems. These requirements can be low power consumption, performance in a given application domain, guaranteed response time, code size and overall system cost.

To meet these criteria, processor characteristics can be customized to match the application profile. Customization of a processor for a specific application holds the system cost down, which is particularly important for embedded consumer products manufactured in high volume.

The trade-offs involved in designing application-specific processors differ considerably from the design of general purpose processors, because they are not optimized for “application mix” (such as represented in the SPEC benchmark suite) but target a specific problem class. The primary aim then is to satisfy the design goals for that problem class, with performance on general purpose code being less important.

In designing application specific processors, different design solutions and trade-offs between hardware and software components constituting a system have to be explored. As a consequence, application-specific processor design is most naturally defined as

a hardware/software co-design problem. An overview of design issues for application specific processors is given in [4].

Hardware/software co-design of application-specific processors identifies functions which should be implemented in hardware and which in software to achieve the requirements of the application, such as code size, cycle count, power consumption and operating frequency. Functions implemented in hardware are incorporated in an application specific processor either as new instructions and processor capabilities, or in the form of special function units. These special function units may then either be integrated on the chip or implemented as peripheral devices.

In the past, such support tools have been available mostly for the evaluation of software aspects, based either on trace collection and evaluation of the design or a cycle-accurate timer model of the architecture. This availability of software tools has led to a situation where studies focus intensely on software issues which are easily accessible with the methodologies and tools at hand, while neglecting the hardware aspects such as design size, power consumption and achievable processor frequency.

We have tried to investigate possibilities for achieving equally good evaluation capabilities of hardware aspects of instruction set design. In the course of this, we have turned to reconfigurable processor cores described in a hardware description language and synthesis to evaluate the impact a design decision may have on hardware complexity.

In this work, we show how proposed instruction set extensions can be evaluated for both its impact on program performance and on hardware efficiency. We describe *hardware/software co-evaluation*, as presented in [6], and how this methodology can be applied to explore the design space and generate optimized implementations for specific application requirements.

We have developed a processor core based on the MIPS RISC architecture [15] which serves as base for hardware/software co-design space exploration [7]. We have synthesized the VHDL description of the processor core for several target technologies, such as MHS 0.6 μ and AMS 0.6 μ ASIC and the Xilinx XC4000XL FPGA family [7, 10]. In this work, we have used the 1.5 μ LSI 10k ASIC process as target technology to facilitate comparison to the original MIPS-I implementations.

We have used the MIPS processor core to explore several application-specific instruction set extensions to implement a memory prefetching mechanism and other performance enhancing extensions, including tag support for dynamically typed languages such as Prolog [6], vector processing [9], and fuzzy processing [20].

This paper is structured as follows: We present related work in section 2. Section 3 introduces the hardware/software co-design methodology used for design evaluation. Section 4 shows how to extend the processor core to include new operations. Sections 5 and 6 give a description of the instruction set evaluation performed for Prolog and prefetching architectures, respectively. We draw our conclusions in section 7.

* This work was performed while the author was with Technische Universität Wien, Vienna, Austria.

2 Related Work

A number of reconfigurable processor cores have been made available commercially recently. These include a MIPS-I processor core by Lexra which is similar in many ways to the core used in this work [24]. Lexra allows its licensees to add execution units and/or instructions to the base CPU by tapping directly into the CPU core. Lexra's synthesizable model breaks out all the signals – including a 12-bit opcode-field, two 32-bit operand buses, a 32-bit result bus, and synchronization signals – needed to attach extra execution units. Any unused MIPS opcodes may be used to enable the extra instructions.

Another commercial offering for reconfigurable processor cores is the ARC processor core [1, 23]. Unlike Lexra's MIPS core, the ARC core is not compatible with any existing architecture. The processor core is supported by a "configuration wizard" which allows to parameterize various aspects of the processor core and select from predefined instruction set extensions. Another extendible core specifically designed for DSP applications is reported in [18].

An alternate approach to processor configurability is taken by Siemens' Carmel architecture which is aimed at the DSP market [22]. While the processor architecture is fixed, users can define complex instruction words which execute up to 6 operations in parallel. This approach is quite similar to variable length VLIW instruction formats.

Instruction set definition has previously been addressed in a number of publications, but the authors have treated instruction set design and instruction set selection mostly as a scheduling problem of operations [12, 13].

An alternative approach treats instruction set selection either as a module selection or operation coupling problem [21, 25]. This approach is driven by compiler statistics generated during the code generation phase and cannot take into account accurate hardware cost estimates. This approach is closely related to reconfigurable compiler and simulator generation based on instruction set descriptions.

Both the pipeline scheduling and module selection approaches cannot generate new logic resources. An approach which can actually generate new logic capabilities for a processor has been presented in [2] for an adaptive machine architecture. Here, the compiler extracts functionality from a high-level languages description and implements it in field-programmable gate arrays (FPGAs) attached to a processor. However, this approach suffers from high communication overhead between the processor and the attached FPGAs and also the idioms recognized by the system seem rather limited.

3 Instruction Set Design as a Co-Design Problem

In the past, hardware/software co-design has been applied mostly to the partitioning of embedded systems [26, 3, 5]. In these systems, the software part executes on a processor with a fixed instruction set and the hardware part is implemented as coprocessors. Hardware/software co-design process involves a partitioning step into a software component executing on a microprocessor and a hardware component. Typically, the hardware component consists of one or more ASIC or FPGA support chips.

These hardware components are more or less tightly coupled to the microprocessor, involving different synchronization costs. In such environments, the selection of an appropriate communication and synchronization method between the software partition and the coprocessors is an important factor which also influences possible partitioning [11, 14].

In our work, the hardware partition is not in a coprocessor, but the modified microprocessor. Thus, communication and synchronization are solved implicitly. Communication between hardware

and software takes place via the processor state, which can be modified by both the hardware and software part. Consequently, the major problem in hardware/software co-evaluation is partitioning to reduce a given cost metric and closely related to instruction set generation.

In application specific processors, the hardware component of a co-design solution is integrated on the chip and directly controlled by the instruction stream. Because of this tight coupling, the attached hardware can access all processor resources such as register files and other processor state, which virtually eliminates the cost of synchronization and data transfer. The interface and synchronization between hardware components is more or less statically defined by the instruction stream.

In the design of such processors, interface synthesis between the hardware and software components takes the form of instruction set definition. This tight coupling of hardware and software in problem-optimized instruction set architectures has been taken advantage of in a number of design solutions, specifically in areas such as signal and multimedia processing.

An important design point in partitioning functionality into hardware and software components is the interface between these system parts. In application specific processors, where the application-specific hardware components are tightly coupled to the processor, this interface is usually in the form of special-purpose instructions embedded in the instruction stream. This allows for natural synchronization, tight coupling and fast communication between software and hardware components.

Tight coupling implies that extensions must run at processor speed, and when introducing new instructions, the impact on hardware complexity and clock frequency has to be considered. In environments where power consumption is constrained, higher processor frequency is rarely desirable: a lower processor frequency with fewer cycles and a smaller code footprint may provide competitive performance with lower power consumption.

We base our hardware/software co-design methodology on the usage of a processor core as a starting point in design process. The core is extended with new functionality to tune the processor to a specific application. Developing a full custom processor for each application may result in a more efficient design, but full investment, higher developing costs, longer time to market and higher risk make designing from scratch a dedicated processor for a specific application prohibitive.

Starting from an existing "thin" RISC processor core reduces overall engineering costs. In this approach, the extendible processor core defines the processor framework and design efforts can be focused on defining its application-specific enhancements. Several variations of an extended processor may be evaluated before the optimal design is selected. In our approach, we use these prototypes to evaluate hardware, and develop software to evaluate the current design. This allows exploration of the design space to define the final architecture without incurring unreasonable cost.

The core-based co-design process consists of the following steps: instruction definition, cycle-level simulation, processor core extension and evaluation.

4 Extending The Processor Core

To allow for new functionality present in extended instructions, the available resources can be parameterized. This includes adding new function blocks, extending existing function blocks (such as adding additional functionality to the ALU) and introducing new interfaces between blocks.

The additions required to implement most instruction set extensions are localized to the IF and EXE pipeline stages. Consider the example of implementing the difference or zero `doz` instruction.

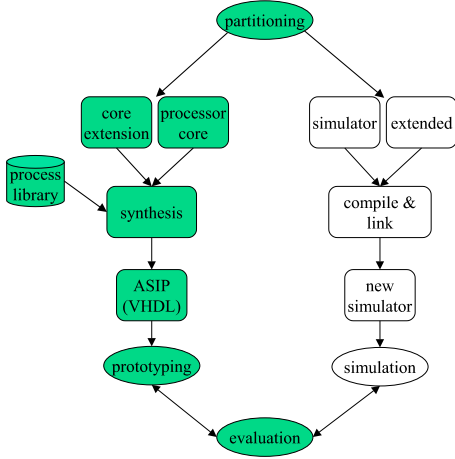


Figure 1: Design flow in the hardware/software co-design approach in designing application-specific processors.

An operation code has to be assigned to the `doz` operation in the instruction set architecture, the instruction decode stage must be modified to recognize this instruction, and decode it appropriately for the ALU. These are trivial changes to the processor, which pre-decodes all common signals for known instruction formats.

In the execution stage, the difference or zero function is then implemented as part of the ALU. This is performed by extending a case statement for the operation code with a clause implementing this particular operation. All input signals to the ALU are generated by the surrounding logic, so the ALU only needs to implement the actual computation:

```

when doz_op =>
  if signed(fw_data_rs1) < signed(fw_data_rs2) then
    -- result is 0
    data_rt <= conv_std_logic_vector(0, 32);
  else
    -- result is difference
    data_rt <= signed(fw_data_rs1) -
      signed(fw_data_rs2);
  end if;

```

Similar extension capabilities are available for other aspects of the processor design, such as the branch architecture. This allows implementation of special purpose branching structures such as counter loops or multi-way branches. Consider the following three-way branch which branches to different addresses depending on the `sgn()` function of the argument register:

```

when bnp =>
  -- three way branch, fall through if reg == 0
  if negative = '1' then -- reg < 0
    branch_address <= address(pipe_out.addr_br1);
    take_branch <= '1';
  elsif zero = '0' then -- reg > 0
    branch_address <= address(pipe_reg.addr_br2);
    take_branch <= '1';
  end if;

```

In addition, the use of communicating state machines for synchronizing pipeline control make even complex control flow in the pipeline easy to implement. Such more complex control adaptation was used for implementing scoreboarding as used in prefetching.

5 Case Study: Prolog Support

We have evaluated instruction set extensions targeted at improving Prolog program performance [6], which have been developed

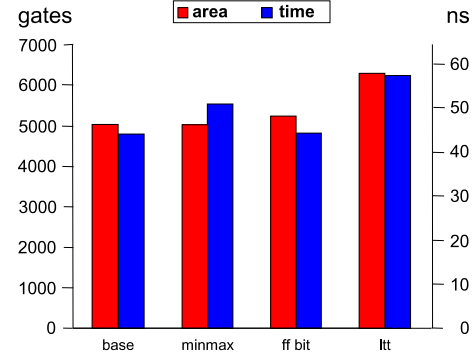


Figure 2: Area usage and critical path for instructions implemented in the ALU.

to support the high-performance VIP Prolog system designed at Technische Universität Wien [17].

5.1 Architectural support for Prolog execution

Based on the performance characterization of the VIP Prolog system, critical operations were identified to efficiently support the Prolog interpreter, the runtime system, and code generated by the Prolog compiler [16]. The functionality identified as critical for efficient execution of Prolog, and other dynamically typed languages, was appropriate tag support to facilitate efficient decoding and dispatching of different functions based on the data type.

In addition, since Prolog is an interpreted language based on a byte code intermediate representation, support for fetching, decoding and dispatching the byte codes of the Prolog Virtual Machine was identified as important.

Finally, support for automatic stack and memory management, and for the implementation of efficient garbage collection, should be considered [16].

Tag handling is critical because in dynamically typed languages, the actual operation to be performed is only identified at program run time based on the data types presented to it. The meaning of operations is “overloaded” and only resolved at run time based on the operand types.

To reduce the path length of this decoding, different forms of multiway branching can provide the appropriate amount of branch bandwidth to dispatch operations in a single cycle. Thus, we considered different implementations of multiway branching, such as 3-way branch (b3w), 4-way branch (b4w), and computed (jmptag) and table-based n-way branches. To remain flexible, position and size of the tags used by these instructions should be a parameter of the operation.

We also experimented with additional addressing modes for branching (like conditional register indirect branching) and with conditional function calls, both of which are useful for interpreter implementation.

Memory management (i.e., Prolog stack management and garbage collection) is supported by the operations min, max, and find first bit.

5.2 Evaluation

Table-based n-way branch proved to be very complex because the data flow did not map well on the processor pipeline. Ensuring consistent pipeline synchronization (two branch delay slots are necessary to cover the costs of table lookup and instruction fetch) would have increased control complexity unreasonably. To simplify implementation and increase achievable frequency, we implemented a

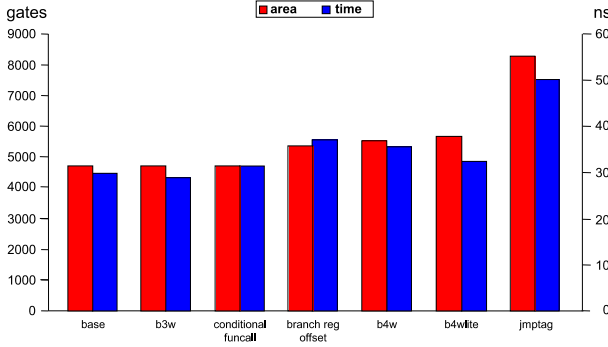


Figure 3: Area usage and critical path for branch operations.

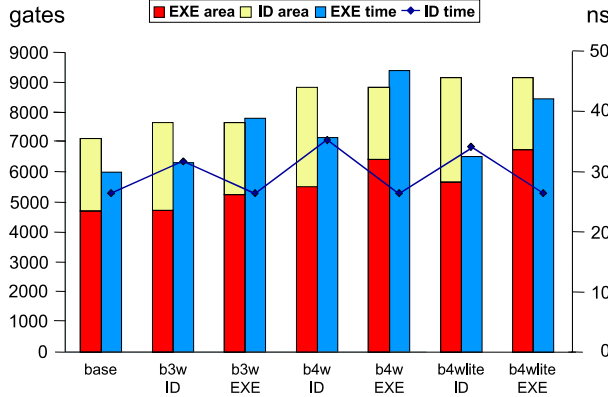


Figure 4: This figure shows the effect of different microarchitecture decisions on the critical path and resource usage. Precomputing results in the instruction decode stage reduces the overall critical path for the branches.

'load tag table' instruction (l_{tt}) which performs the table lookup for an n-way branch based on a tag. The branch can then be performed with a normal register-indirect branch instruction.

This design achieves the same functionality while fitting on the existing pipeline flow. In addition, the tag table lookup instruction primitive (l_{tt}) can also be used in other contexts where tags are used to index data. While this design decision leads to some degradation in achievable CPI, it results in better overall performance due to higher clock speed. We also investigated a four-way branch with fixed tag position (b4w|lte) to reduce delay associated with extracting the tag of the source register. Arbitrary tag positions can be supported by shifting the tag appropriately.

The results for instruction set evaluation are classified by modified processor components, i.e., ALU or branch unit. Although l_{tt} is a memory operation, the only modifications required were to the ALU for address computation of the memory access. The memory access is a regular 32 bit read operation from an effective address.

Figure 2 gives the gate count and critical path delay for extensions to the execution stage. Only data for execution stage is given, as only the ALU was modified and the other modules remained unchanged.

Figure 3 gives the gate count and critical path delay for new branch instructions. These change only the branch address and branch condition evaluation in the execution stage, the interface between EXE and the AT stage which contains the program counter being just an address and a branch flag (indicating whether to branch to that address or not). The critical paths reported are those for the interface to the AT stage, not those for other interfaces.

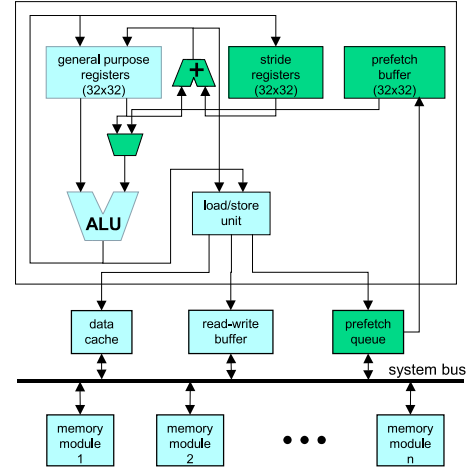


Figure 5: Prefetch architecture.

In exploring the design space for efficient implementation of the multiway branches, we have experimented with precomputing partial results in the instruction decode stage. This distributes the amount of logic required and reduces the critical path in the execution stage. The results of this experiment are shown in figure 4.

6 Case Study: Vector Prefetching

Using the presented approach, we also tried to evaluate a scheme for prefetching vector data with regular data layout. The scheme is based upon stride-directed prefetching, where the stride is the distance between successive accesses. This is especially good for vector and matrix manipulation, or for multimedia data which have poor or no locality, or a working set which is too large to fit in the cache.

Stride-directed stream prefetching is particularly useful for embedded applications where L2 caches are too expensive or which perform multimedia processing with a high number of compulsory misses due to large working sets. In fact, streaming data is often accessed only once in these applications (or far enough apart such that a previously loaded value would already have been displaced from the cache from further accesses.)

6.1 Prefetch architecture

Figure 5 gives an overview of the system architecture with stream prefetching. The processor core is extended by two additional register files, the stride register file and the prefetch buffer. The stride register file holds the distance between two successive accesses, and the prefetch buffer is the destination for prefetch requests.

Operations can directly access the prefetch buffer as one of the source operands. When the prefetch buffer is accessed, the next asynchronous prefetch access initiated. Prefetch addresses are computed by a dedicated address generation unit. The prefetch unit and the CPU are synchronized through the use of scoreboarding on the prefetch buffer.

The CPU accesses a split transaction bus, and multiple memory modules can be used to supply sufficient memory bandwidth. The prefetch architecture is described in more detail in [9].

6.2 Performance

For performance characterization, we have used Livermore vector processing kernels [19] with varying problem size. We have evaluated configurations for cold- and warm-started caches, as well as

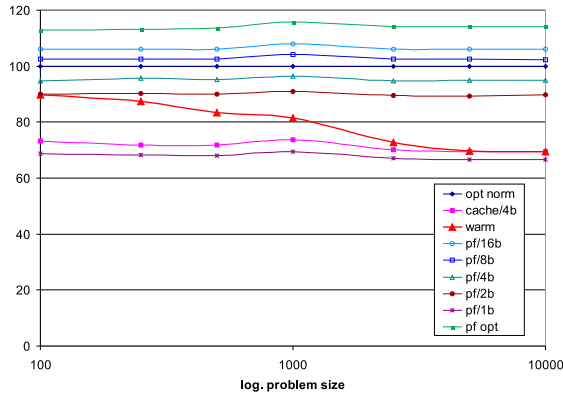


Figure 6: Livermore kernel performance for different memory access schemes, as a function of problem size.

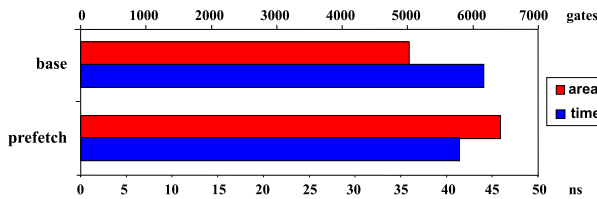


Figure 7: Area usage and critical path of the execution unit after implementing the required functionality for data streaming.

prefetching with different memory subsystems. The normalized geometric mean for cycle count as a function of problem size is shown in figure 6. Additional results for some graphics code kernels are reported in [8].

Figure 7 shows resource usage and critical path for the execution stage of a prefetch implementation. The area usage in the execution stage is increased essentially by the size of an adder needed to compute the next memory access address by adding the stride to the current address. The critical path of the execution stage is actually shortened, because we have streamlined the implementation. While memory addresses were previously computed by the ALU, all addresses are now generated by a single address adder, simplifying the internal structure.

7 Conclusion

In this paper, we have demonstrated how to apply hardware/software co-evaluation to instruction set definition. We have defined and evaluated instruction set extensions for two application areas. The instruction set extensions have been added to a RISC processor core based on the MIPS instruction set architecture. In this work, we evaluate instruction set architecture extensions optimized for several application-specific problems.

A core-based design approach allows early design evaluation with low non-recurrent engineering costs to reduce overall design risk. As this method gives quick feedback on the performance, efficiency and price of the design, several design choices from the design space can be evaluated in a very short time until the optimal design is specified.

Acknowledgments

The author wishes to thank Valentina Salapura who has contributed to the quality of this work by numerous valuable suggestions.

References

- [1] ARC. Argonaut RISC cores. <http://www.riscscore.com>.
- [2] ATHANAS, P. M., AND SILVERMAN, H. F. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer* 26, 3 (March 1993), 11–18.
- [3] BUCHENRIEDER, K. *Hardware/Software Co-Design: An Annotated Bibliography*. IT Press, Chicago, IL, 1994.
- [4] DE MICHELI, G. Computer-aided hardware-software codesign. *IEEE Micro* 14, 4 (August 1994), 10–16.
- [5] ECKER, W. Using VHDL for HW/SW co-specification. In *Proc. of the 1993 European Design and Automation Conference with EURO-VHDL '93* (Hamburg, Germany, September 1993), R. Camposano, Ed., GI, IEEE Computer Society Press, pp. 500–505.
- [6] GSCHWIND, M. *Hardware/Software Co-Evaluation of Instruction Sets*. PhD thesis, Technische Universität Wien, Vienna, Austria, July 1996.
- [7] GSCHWIND, M., AND MAURER, D. An extendible MIPS-I processor kernel in VHDL for hardware/software co-design. In *Proc. of the European Design Automation Conference EURO-DAC '96 with EURO-VHDL '96* (Geneva, Switzerland, September 1996), GI, IEEE Computer Society Press, pp. 548–553.
- [8] GSCHWIND, M., AND PIETSCH, T. A smart cache for improved vector performance. In *Proc. of the First International Meeting on Vector and Parallel Processing* (Porto, Portugal, September 1993).
- [9] GSCHWIND, M., AND PIETSCH, T. Vector prefetching. *ACM Computer Architecture News* 23, 5 (December 1995), 1–7.
- [10] GSCHWIND, M., SALAPURA, V., AND MAURER, D. FPGA prototyping of a RISC processor core for embedded applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (1998). submitted.
- [11] GUPTA, R. K., COELHO, C. N., AND MICHELI, G. D. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of the 29th Design Automation Conference (DAC '92)* (Anaheim, CA, June 1992).
- [12] HOLMER, B. K. A tool for processor instruction set design. In *Proc. of the 1994 European Design Automation Conference with EURO-VHDL '94* (Grenoble, France, September 1994), IEEE Computer Society Press.
- [13] HUANG, I.-J., AND DESPAIN, A. M. Synthesis of instruction sets for pipelined microprocessors. In *Proc. of the 31st Design Automation Conference (DAC '94)* (San Diego, CA, June 1994), ACM.
- [14] ISMAIL, T. B., ABID, M., O'BRIEN, K., AND JERRAYA, A. An approach for hardware-software codesign. In *Proc. of 5th International Workshop on Rapid System Prototyping* (Grenoble, France, June 1994), IEEE, pp. 73–80.
- [15] KANE, G., AND HEINRICH, J. *MIPS RISC Architecture: reference for the R2000, R3000, R6000 and the new R4000 instruction set computer architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [16] KRALL, A. An extended Prolog instruction set for RISC processors. In *VLSI for Artificial Intelligence and Neural Networks* (New York, NY, 1991), J. G. Delgado-Frias and W. R. Moore, Eds., Plenum Press, pp. 101–108.
- [17] KRALL, A. The Vienna Abstract Machine. *Journal of Logic Programming* 29(1-3) (1996), 85–106.
- [18] KUULUSA, M., NURMI, J., TAKALA, J., OJALA, P., AND HERRANEN, H. A flexible DSP core for embedded systems. *IEEE Design and Test of Computers* 13, 4 (October 1997), 60–68.
- [19] MCMAHON, F. H. Lawrence Livermore National Laboratory FORTRAN Kernels Test: MFLOPS. FORTRAN source code, September 1991.
- [20] SALAPURA, V., AND GSCHWIND, M. Hardware/software co-design of a fuzzy RISC processor. In *Proc. of the Design, Automation and Test in Europe Conference DATE '98* (Paris, France, February 1998), EDAA, IEEE Computer Society Press, pp. 875–882.
- [21] SATO, J., ALOMARY, A. Y., HONMA, Y., NAKATA, T., SHIOMI, A., HIKICHI, N., AND IMAI, M. PEAS-I: A hardware/software codesign system for ASIP development. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science E77-A*, 3 (March 1994), 483–491.
- [22] SUCHER, R. Carmel: A configurable long instruction word DSP core. In *Microprocessor Forum 1998* (San Jose, CA, October 1998), MicroDesign Resources.
- [23] TURLEY, J. ARC getting full-30 licensees aboard. *Microprocessor Report* 12, 15 (November 1998).
- [24] TURLEY, J. LXR-4080 is independent implementation of basic MIPS-I processor core. *Microprocessor Report* 12, 2 (February 1998), 13.
- [25] VAN PRAET, J., GOOSSENS, G., LANNEER, D., AND DE MAN, H. Instruction set definition and instruction set selection for ASIPs. In *Proc. of the 7th International Symposium on High-Level Synthesis 1994* (Niagara on the Lake, ON, Canada, 1994), IEEE, pp. 11–16.
- [26] WOLF, W. H. Hardware-software co-design of embedded systems. *Proceedings of the IEEE* 82, 7 (July 1994), 967–989.