Hardware/Software Co-Design of a Fuzzy RISC Processor

Valentina Salapura

valentina@vlsivie.tuwien.ac.at

Technische Universität Wien
Treitlstraße 1-182-2
A-1040 Vienna, Austria

Michael Gschwind[*]

mikeg@watson.ibm.com

Technische Universität Wien
Treitlstraße 1-182-2
A-1040 Vienna, Austria

Abstract

In this paper, we show how hardware/software co-evaluation can be applied to instruction set definition. As a case study, we show the definition and evaluation of instruction set extensions for fuzzy processing. These instructions are based on the use of subword parallelism to fully exploit the processor's resources by processing multiple data streams in parallel. The proposed instructions are evaluated in software and hardware to gain a balanced view of the costs and benefits of each instruction. We have found that a simple instruction optimized to perform fuzzy rule evaluation offers the most benefit to improve fuzzy processing performance.

The instruction set extensions are added to a RISC processor core based on the MIPS instruction set architecture. The core has been described in VHDL so that hardware implementations can be generated using logic synthesis.

1 Introduction

In this work, we analyze how fuzzy processing can be implemented efficiently on general purpose CPUs and what functionality is required to achieve peak performance.

Instruction sets are often optimized for some software metric such as a minimum number of clock cycles, but this approach neglects the hardware impact of the proposed instructions. While some instructions may reduce the cycle count, they may also lengthen the cycle time and even result in a net performance loss. To obtain a more balanced view of software and hardware implications of instructions, a co-design approach to instruction set definition is necessary.

In this work, we show how proposed instruction sets can be evaluated for both its impact on program performance and on hardware efficiency. This approach is demonstrated with the evaluation of fuzzy instruction set extensions using the hardware/software co-evaluation method.

Fuzzy computation can be implemented on any general purpose processor. However, because instruction set architectures were not designed with fuzzy computation in mind, the available primitives can result in an inefficient implementation. A possible approach to rectify this situation is to introduce additional instruction set primitives to efficiently support fuzzy processing.

The proposed fuzzy instruction set extensions have been designed to optimally exploit the available processing resources by using subword parallelism. Because fuzzy computation operates with short data types, multiple data can be packed in a single 32 bit processor word.

As a starting point for optimizing fuzzy performance, we use a RISC processor core based on the popular MIPS-I instruction set architecture [1]. Both the processor core and the added application specific instructions have been described using the VHDL hardware description language [2], and synthesized using logic synthesis.

This paper is organized as follows: section 2 introduces the hardware/software co-design methodology we have used for instruction set architecture evaluation and section 3 gives an overview of related work. In section 4, we describe the MIPS RISC processor core which serves as starting point for our design. We give an overview of fuzzy issues and the proposed extension of the processor in section 5. The hardware/software co-evaluation of the fuzzy instructions is given in section 6 and we draw our conclusions in section 7.

2 Hardware/Software Co-Design of Instruction Sets

The evaluation of the instruction set architecture is a major issue in the design of application specific instruction processors [3]. To optimize processor performance for a particular application, a common approach is to extend the instruction set by application specific instructions. Many instruction set extensions have been proposed, such as for signal processing or multimedia processing.

Adapting an instruction set to a particular problem is a

[*] Dr. Gschwind is currently with the IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

difficult task, as many unknown issues have to be explored. Due to the many factors involved in performance optimization, suggested optimization solutions often minimize only the number of instructions necessary to solve a problem, or at best the number of cycles.

However, many of the suggested special purpose instructions are complex. As a result, it may not be possible to clock an extended processor at the same frequency as the original design. This is often neglected by studies, as the processor characteristics can be difficult to predict, and as a full implementation of a processor is often out of scope. Thus, studies most often use instruction or cycle level simulators such as SPIM [4] to predict performance. While these instruction set emulators can be used to test software, generate traces and gather statistics, they do not allow to predict the effects of the extended instruction set architecture on the processor design itself.

In this work, we evaluate the instruction set architecture (ISA) optimized for fuzzy computations using *hardware/software co-evaluation* of instruction set extensions to gain a more balanced view of the benefits of different instruction sets.

Proposed instructions are evaluated in hardware and software to establish the performance impact of each instruction:

- Software evaluation can be performed using program traces, instruction set simulators or object code instrumentation. During this step, the cycle count of benchmarks is established.
- Hardware evaluation is performed using rapid prototyping based on logic synthesis. To evaluate hardware effects of instruction set extensions, we have designed an extendible RISC processor core. Instructions are implemented and the extended processor architecture is synthesized to establish cycle time and chip area.

Using the information derived from these steps, the benefits of each proposed instruction can be evaluated in order to decide whether to implement a particular function in hardware or in software. Effectively, this process moves functional blocks from software to hardware or vice versa to optimize performance and cost.

Using this co-evaluation approach, we have evaluated 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 [5]. In this work, we evaluate instruction set extensions optimized for fuzzy computation.

Previously presented automatic instruction set definition approaches have used either pipeline scheduling or module selection to define an "optimal" instruction set. Alas, these automatic approaches cannot consider or optimize data layout based on such methods as subword data parallelism which require human intervention to adapt the data layout to a particular problem set.

3 Related Work

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 [7], [8], or as a module selection problem [9].

Scheduling approaches derive the best combination of operations to be executed in a pipeline or where to put them in a pipeline. One starts from a fixed pipeline and tries to schedule operations found in application programs to achieve high resource utilization of different functional blocks available in the pipeline [7]. A different approach is the partitioning of instructions on different pipeline steps while the instruction set is mostly fixed and the operations are mapped on the different pipeline stages to reduce delay.

The module selection approach [9] uses frequency analysis of software traces to determine the types of instruction to be supported by processor. The n most frequent operations are implemented in hardware selecting from a fixed set of modules, but no thought is given to the impact of implementing a functional block in hardware on the cycle time.

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 [10] 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. Alas, 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.

Thus, to generate truly optimized logic, human intervention as supported by our hardware/software co-evaluation approach is still required to make the best usage of logic capacity.

In the area of fuzzy processing, a number of approaches have been used to optimize fuzzy processing based on either custom hardware implementations or programmable solutions, using custom hardware processors or extensions to existing processors.

Custom fuzzy implementations are generally mapped directly to an ASIC process to implement a particular class of hardware problems [11]. This approach gives the most efficient solution if only a restricted class of fuzzy problems are to be implemented.

Programmable implementations of fuzzy processing are either additions to existing processors (such as found in the CPU12 from Motorola [12] or in the FLORA processor [13]), or custom fuzzy programmable processors. The FLORA processor extends a RISC instruction set with the `min` and `macc` instructions (for the minimum calculation and the multiply-and-accumulate operation, respectively) to improve fuzzy processing.

4 An Extendible Processor Core and Its Development Environment

We have developed an extendible processor core based on the MIPS-I RISC architecture in VHDL. This processor core gives us the possibility to study the effects of instruction set architectures on processor speed and implementation area using rapid prototyping.

For the processor to be useful for these purposes, we identified the following requirements:

high-level description The format of the processor description should be easy to understand and modify.

modular To add new instructions, only the relevant parts should have to be modified. A monolithic design would make experiments difficult.

extendible All data structures and interfaces should be designed such that new fields can be added with ease.

synthesizable The processor description should be synthesizable to derive actual hardware implementations.

The processor core has been designed with a distributed controller to facilitate instruction set extension and processor adaptation for specific application requirements. This distributed controller approach replaces a monolithic controller which would be difficult to adapt. The distributed controller is responsible for pipeline flow management and consists of communicating state machines found in each pipeline stage. Thus, changes in the architecture can be restricted to those modules where new functionality is provided.

The processor core is described in synthesizable VHDL. Thus, hardware implementations can be derived using logic synthesis. In our work, we use the Synopsys Design Compiler [14] as synthesis tool to generate ASIC implementations. We have synthesized the VHDL description of the processor core for the AMS 0.6 μ CMOS process [15]. Table 1 gives the size of each module of the synthesized design.

A more detailed description of the processor core, its implementation and validation can be found in [16].

5 Defining Fuzzy Extensions

5.1 Fuzzy Principles

Fuzzy computation consists of three steps: fuzzification, inference and defuzzification.

Module	sq. mil	Description
AT	949	AT pipeline stage (PC, TLB access)
IF	735	instruction fetch unit
ID	2453	instruction decode unit
EXE	3557	execution unit (ALU)
MD	4037	integer multiply/divide unit
MEM	4128	data memory access
WB	321	register file writeback
RF	N/A	register file
CP0	3674	coprocessor 0 (exception handling, TLB)
CCON	384	cache controller
Total	20286	processor core

Table 1: Complexity of modules as size in sq. mil of the RISC processor core based on the MIPS-I architecture.

During fuzzification, crisp input signals are mapped onto fuzzy variables. Each input value is assigned a degree of membership in each fuzzy set, also referred to as alpha value.

Inference implements the evaluation of fuzzy rules. Fuzzy rules from a rule data base are applied to the fuzzified inputs, determining a fuzzy control action. Fuzzy rules have `if - then` semantics:

`if (`$Input_0$` is A) and (`$Input_1$` is B) and ... then `$Output$` is C`

The intersection of rule premises is performed by selecting the minimal alpha value. The rule conclusion gives the membership degree θ in the output fuzzy set C. As several rules may be applicable to some combination of input values, more than one membership degree θ can possibly be computed for a single fuzzy set. These degrees are than consolidated into a single membership degree θ by selecting the maximum of all computed θ values for each output set.

During defuzzification, control actions are converted back to crisp signals. The θ values delimit the output fuzzy sets defining an area. The ordinate of gravitational center of this area determines a crisp output control signal.

5.2 Subword parallelism

Packing of multiple data streams in a single processor word is referred to as subword parallelism. This method has gained widespread acceptance lately to support operation on multiple related data items in a single cycle for applications such as media processing, video conferencing, or multimedia and communication applications [17].

Subword parallelism can be applied to fuzzy computation for fuzzification, inference and defuzzification, as 8

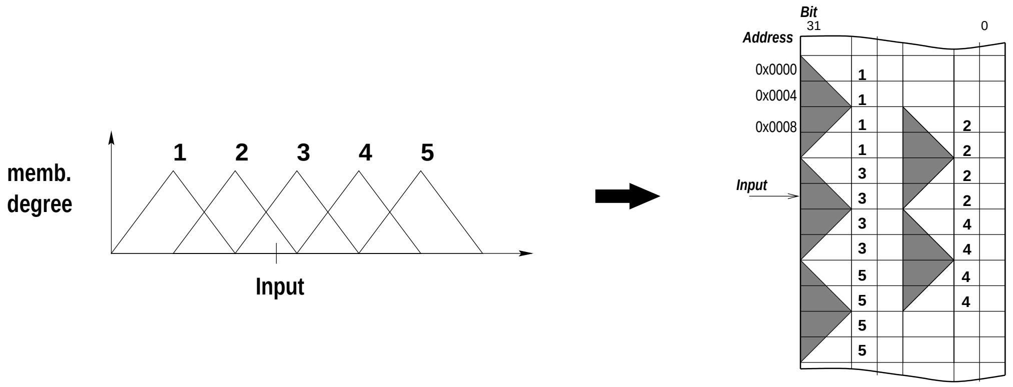


Figure 1: Subword data memory organization: each word contains membership information for two fuzzy sets. The fuzzy sets are specified using unique fuzzy set identifiers and the membership is encoded using 8 bit unsigned integer. The current implementation supports 2 overlapping fuzzy sets for each input.

bits offer sufficient precision for representing inputs, outputs and membership functions of fuzzy sets [18]. Subword parallelism can be used during fuzzification to compute multiple membership degrees in one step, during inference to perform rule evaluation in a single cycle, and during defuzzification to operate on multiple data sets for integration.

As alpha values require only 8 bits, more than one set can be packed in a single processor word. To identify fuzzy sets, some set identification has to be included, requiring an additional 4 bits for a maximum of 16 possible sets for a single input. Thus, in a 32 bit processor we can operate on two overlapping sets concurrently. This concept can be extended to five overlapping sets on a 64 bit processor.

By using data parallelism, parallel operation is performed on multiple data in determining alpha values for all fuzzy sets for an input, and performing defuzzification of two output sets concurrently.

For fuzzification, we define an optimized data layout in memory (see figure 1). For each memory access, the membership function for a crisp input is computed for two fuzzy sets. Both fuzzy sets and the associated membership degree are encoded in a single 32 bit word. The membership degree alpha for all other fuzzy sets defaults to 0.

This memory organization minimizes the number of memory accesses, requiring only one load per input and in this way speeding up the execution. In addition, memory requirements for this organization are minimal and independent of the number of fuzzy sets.

5.3 Fuzzy Instructions

To improve performance of RISC processors for fuzzy calculation, we have explored different ISA extensions for fuzzy workloads. For this purpose, we have extended the original MIPS-I instruction set architecture (ISA) with several instructions specialized for fuzzy computation. For each new instruction, we have analyzed its impact on hardware and software, as well as obtained performance. The final ISA extensions are determined by results of the hardware/software co-evaluation of these extensions.

To support fuzzy computations, we have considered the following instructions:

slw loads fuzzified values from the memory,
rulev evaluates fuzzy rules from the rule base,
macc multiply-and-accumulate operation for defuzzification,
hmul halfword multiplication for defuzzification, and
hadd result collection for defuzzification.

slw To optimize access to arrays, a register plus shifted register memory addressing mode (scaled load) can be used. This instruction is not specific to fuzzy calculation as it improves performance of array accesses as can be found in all general purpose programs and is included in a number of microprocessors, such as most CISC machines and several RISC processors (such as the Motorola m88k).

The instruction `slw` R_d, R_s, R_t takes two operands to specify a memory address – the first operand R_s specifies the base of the array and the second operand R_t the index in the array. To generate the actual memory address, the index is multiplied by the data size (4 bytes) and added to the base.

In fuzzy processing, the scaled load instruction can be used to implement the table lookup used for fuzzification of input values in a single processor cycle.

rulev Evaluation of fuzzy rules is performed by determining the minimum of all rule premises. During the inference step, only the rules where all premise alpha values are non-zero have to be evaluated.

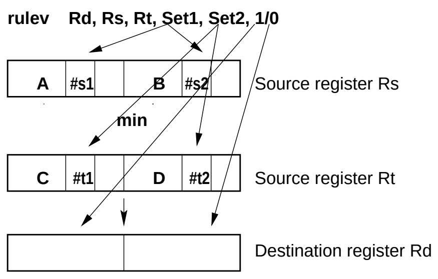


Figure 2: For the rule evaluation, fuzzy set identifiers are compared with set specifiers in the instruction, and the minimum is determined.

A rule evaluation instruction `rulev` evaluates a fuzzy rule in a single cycle. The instruction `rulev` R_d, R_s, R_t, $Set1$, $Set2$, `1/0` checks whether the alpha values from the source registers R_s and R_t are premises of the rule under evaluation and non-zero. If this is true, it determines the minimum alpha value and stores it in the register R_d. The values $Set1$ and $Set2$ are fuzzy set identifiers in the range 0 to 15. The last argument of the instruction can take the value 0 or 1 to specify whether the result is written in the left (if 1) or in the right (if 0) half of the destination register (see figure 2).

If the alpha values in the source registers are not premises of the rule under evaluation, the corresponding halfword of the destination register is set to zero.

macc Multiply-and-accumulate instruction is a popular choice for various instruction set extensions, as it is useful for solving several problems.

In our design, the instruction `macc` R_d, R_s, R_t, `1/0` multiplies the left or the right halfword (depending on the field specifier supported as the last argument of the instruction) of the source registers R_s and R_t and accumulates the result in the register R_d.

hadd We have introduced this instruction to perform the addition of two halfwords in parallel.

The instruction `hadd` R_d, R_s, R_t performs addition of the left and of the right halfwords of the source registers R_s and R_t in parallel and stores the results in the corresponding halfwords of the destination register R_d.

hmul The last instruction we have analyzed is the halfword multiplication.

The instruction `hmul` R_d, R_s, R_t multiplies the halfwords of source registers R_s and R_t in parallel and stores the result in the destination register R_d.

Problem	#I	#O	#R	#MF
Simple	2	1	7	5
Medium	3	2	14	5
Complex	7	3	80	5

Table 2: Complexity of fuzzy problems: fuzzy problems are classified by the number of inputs (#I), the number of outputs (#O), the number of fuzzy rules (#R) and the number of membership sets (#MF).

conf.	instruction set features
A	core
B	core, rulev
C	core, slw, rulev
D	core, slw, rulev, macc
E	core, slw, rulev, hmul, hadd
F	core, slw, rulev, hmul

Table 3: Processor configurations under evaluation.

6 Hardware/Software Instruction Co-Evaluation

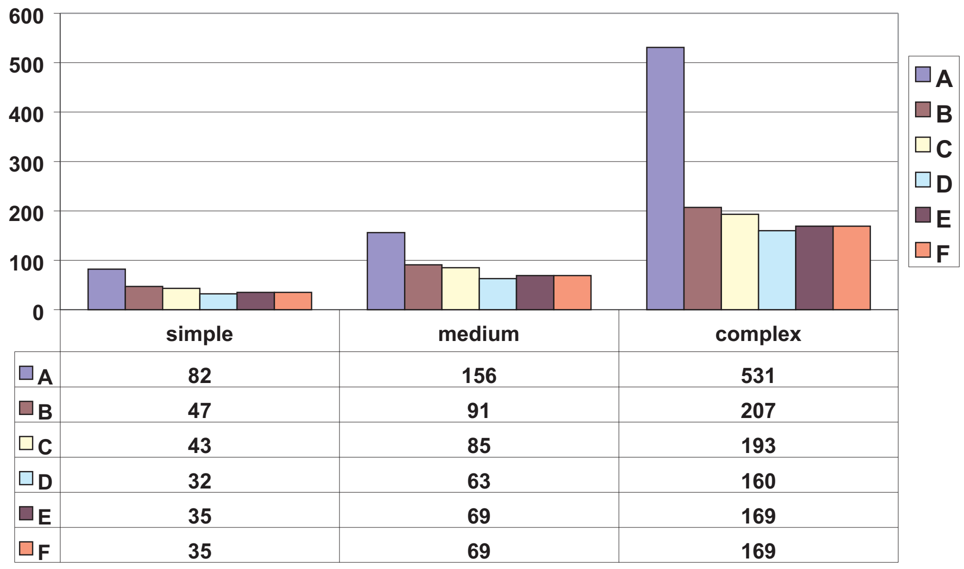
To give a balanced evaluation of the new instructions, we have performed evaluation of both hardware and software aspects.

We evaluate the impact of proposed instruction set features on different classes of fuzzy problems of varying complexity. Table 2 shows the classification of fuzzy problems (based on Costa et al. [13]) which will be used throughout this work. For each class of problem complexity, we have generated application programs using the new instructions and established the cycle count required for execution.

We have analyzed the performance improvement offered by several different processor configurations, ranging from the addition of a single instruction to support fuzzy rule evaluation to hardware support for all fuzzy processing steps. Table 3 gives an overview of the analyzed processor configurations and the instructions included in each of these configurations.

Configuration A implements the MIPS-I RISC instruction set architecture and serves as a reference for the comparison of the extended processor configurations. Configuration B implements only a single additional instruction supporting fuzzy rule evaluation, configuration C adds support for scaled loads for fuzzification and the remaining configurations offer different types of hardware support for defuzzification.

Software evaluation of proposed extensions is performed by computing the number of cycles needed to implement the functionality of the test programs when the instructions under evaluation are used. The cycle counts for



	simple	medium	complex
A	82	156	531
B	47	91	207
C	43	85	193
D	32	63	160
E	35	69	169
F	35	69	169

Figure 3: Cycle count for the evaluation of simple to complex fuzzy problems for different processor configurations.

design	F	I	D	/	total cycles
A	18	30	24	10	82
B	6	7	24	10	47
C	2	7	24	10	43
D	2	7	13	10	32
E	2	7	16	10	35
F	2	7	16	10	35

Table 4: Cycle count required for each processing step in fuzzy calculation for a simple fuzzy problem. F indicates the cycle count for fuzzification, I for inference, D for defuzzification and / for division.

a sample program are given in table 4.

The table lists cycle count required for performing the same fuzzy application on each of the analyzed configurations. The cycle count is given separately for each of the three fuzzy processing steps. In this table, the division has been extracted to simplify the analysis of proposed instruction set extensions.

The performance improvement of introducing specialized instructions becomes more pronounced with the growing complexity of the problem. Figure 3 gives an overview of the cycle count required for executing fuzzy problems of varying complexity for different processor configurations. The highest performance improvement is achieved by the introduction of the instruction **rulev**. This instruction improves performance of fuzzy calculation ranging from 74% to 157%, depending on problem complexity. The introduction of scaled load instruction reduces cycle count by an additional 7% to 9%, whereas instructions supporting defuzzification by 20% to 46%, depending on the architecture and on the design complexity.

Figure 4 shows that the execution profile of fuzzy problems differs significantly from simple to complex problems. With growing program complexity, the impact of fuzzification and defuzzification on the overall execution time decreases whereas the computing time for rule base evaluation becomes more significant. Thus, while simple problems spend much of their execution time in defuzzification, inference takes up 70% of execution time in complex problems.

As a result, complex problems benefit the most from the rule evaluation instruction (157%), with only minor improvements gained by other instructions (7% for the scaled load, 20% for defuzzification support). For simple problems, the biggest gain is still obtained by rule evaluation support (74%), but other extensions also offer significant improvements (9% for scaled load, 46% for defuzzification support) because fuzzification and defuzzification make up a larger part of the execution time.

Another important metric is code size, especially for embedded application. Using the extended ISA for fuzzy calculation we have fewer instructions for rule evaluation and we reduce the code size significantly (e.g., for a simple fuzzy problem the code size decreases from 82 to 40 instructions for configuration C).

However, the reduction of cycle count as a result of introducing the specialized instructions does not automatically imply shorter execution time. The implementation of

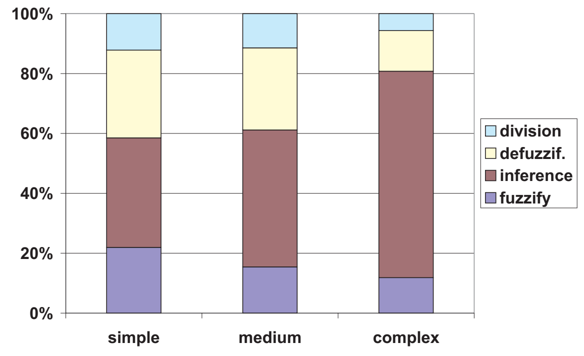


Figure 4: Execution time break down for fuzzy evaluation with varying complexity.

specialized instructions is often complex and may increase the cycle time of the processor and thus reduce the benefits of using the application specialized architecture. For this reason, it is necessary to perform hardware evaluation of the architectures under evaluation as well.

Data about the hardware implementation are derived by designing prototype implementations of the proposed instructions and using logic synthesis to generate a hardware implementation from the VHDL description. The resulting gate-level netlist can then be analyzed to obtain information about area and timing. As target process, we have used the AMS 0.6μ process.

The unmodified processor core (architecture A) achieves an operating frequency of 62 MHz, resulting in an inference speed of 2.5 μs to 8.5 μs depending on the problem complexity. The area cost and critical path of the proposed instructions are reported in table 5. By combining this information with the information about cycle count, overall performance of the different configuration can be obtained (table 6). This information can then be used to decide which instructions to include in the final processor design.

Because the proposed fuzzy instructions were all designed to minimize cycle time impact and resource usage in the first place, hardware evaluation reports only moderate implementation costs. The area increase to implement proposed instructions is low, especially when compared to overall chip size. The critical path is increased by up to 2.5ns for the most expensive instruction (`macc`), but none leading to an increase in cycle time.

7 Conclusion

In this paper, we have demonstrated how to apply hardware/software co-evaluation to instruction set definition. We have defined and evaluated fuzzy instruction set extensions. The instruction set extensions have been added to a RISC processor core based on the MIPS instruction set

instruction	**area increase (sq.mil)**	**critical path (ns)**
base	0.0	9.36
slw	30.6	9.68
rulev	288.3	10.50
macc	1674.4	11.76
hmul	1466.0	9.26
hadd	492.2	9.47

Table 5: Cost of implemented instructions: increase of chip area and cycle time.

	total cycles	**overall time (μs)**	**area (sq.mil)**	**area incr. (in %)**
A	156	2.50	23933	0%
B	91	1.46	24221	1.2%
C	85	1.36	24525	1.3%
D	63	1.01	25928	8.3%
E	69	1.10	25868	8.1%
F	69	1.10	25718	7.5%

Table 6: Results of hardware/software co-evaluation for a fuzzy problem of medium complexity.

architecture. The fuzzy processing instructions are based on the use of subword parallelism to fully exploit the processor's capabilities by processing multiple data streams in parallel.

The highest performance gain for fuzzy processing is brought by the rule evaluation instruction `rulev`, which alone accounts for a performance increase in excess of 150%. Fuzzification can be improved by using the scaled load instruction found in several commercially available processors. The defuzzification step in fuzzy processing can be improved significantly by subword instructions (`hadd`, `hmul`) such as found in a number of multimedia extensions (e.g., HP's MAX, Intel's MMX, or Sun's VIS instructions).

Based on the results of hardware/software co-evaluation of the proposed configurations, we have identified configuration C as the optimal architecture for fuzzy processing. This architecture implements fuzzification and inference in hardware whereas defuzzification is implemented in software. The architecture speeds fuzzy processing up to 175% at a hardware cost of only 1.3%.

Acknowledgment

The authors thank Dietmar Maurer for his help in designing the MIPS processor core.

References

[1] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture: reference for the R2000, R3000, R6000 and the new R4000*

instruction set computer architecture. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[2] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, NY, 1988. IEEE Standard 1076-1987.

[3] Giovanni de Micheli. Computer-aided hardware-software codesign. *IEEE Micro*, 14(4):10–16, August 1994.

[4] James R. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report 966, University of Wisconsin-Madison, Madison, WI, September 1990.

[5] Michael Gschwind. *Hardware/Software Co-Evaluation of Instruction Sets*. PhD thesis, Technische Universität Wien, Vienna, Austria, July 1996.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, second edition, 1996.

[7] Bruce K. Holmer. 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.

[8] Ing-Jer Huang and Alvin M. Despain. Synthesis of instruction sets for pipelined microprocessors. In *Proc. of the 31st Design Automation Conference (DAC '94)*, San Diego, CA, June 1994. ACM.

[9] Jun Sato, Alauddin Alomary, Yoshimichi Honma, Takeharu Nakata, Akichika Shiomi, Nobuyuki Hikichi, and Masaharu Imai. PEAS-I: A hardware/software codesign system for ASIP development. *IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences*, E77-A(3):483–491, March 1994.

[10] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[11] Valentina Salapura and Volker Hamann. Implementing fuzzy control systems using VHDL and Statecharts. In *Proc. of the European Design Automation Conference EURO-DAC '96 with EURO-VHDL '96*, pages 53–58, Geneva, Switzerland, September 1996. IEEE Computer Society Press.

[12] Motorola. *CPU12 Reference Manual*. Motorola, Inc., Phoenix, AZ, 1996.

[13] Alessandra Costa, Alessandro de Gloria, Paolo Faraboschi, Andrea Pagni, and Gianguido Rizzoto. Hardware solutions for fuzzy control. *Proceedings of the IEEE*, 83(3):422–434, March 1995.

[14] Synopsys. *Design Compiler Family Reference*. Synopsys, Inc., Mountain View, CA, November 1996. (Version 1997.01).

[15] AMS. *0.6-Micron Standard Cell Databook*. Austria Micro Systeme International AG, Unterpremstätten, Austria, March 1997.

[16] Michael Gschwind and Dietmar Maurer. 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*, pages 548–553, Geneva, Switzerland, September 1996. IEEE Computer Society Press.

[17] *IEEE Micro*, volume 16, number 4. IEEE Computer Society, August 1996.

[18] Raúl Rojas. *Theorie der neuronalen Netze: eine systematische Einführung*. Springer Verlag, Berlin, Germany, 1993.