Augmenting Modern Superscalar Architectures with Configurable Extended Instructions

Xianfeng Zhou and Margaret Martonosi

Dept. of Electrical Engineering
Princeton University
{xzhou, martonosi}@ee.princeton.edu

Abstract. The instruction sets of general-purpose microprocessors are designed to offer good performance across a wide range of programs. The size and complexity of the instruction sets, however, are limited by a need for generality and for streamlined implementation. The particular needs of one application are balanced against the needs of the full range of applications considered. For this reason, one can "design" a better instruction set when considering only a single application than when considering a general collection of applications. Configurable hardware gives us the opportunity to explore this option. This paper examines the potential for automatically identifying application-specific extended instructions and implementing them in programmable functional units based on configurable hardware. Adding fine-grained reconfigurable hardware to the datapath of an out-of-order issue superscalar processor allows 4-44% speedups on the MediaBench benchmarks [1]. As a key contribution of our work, we present a selective algorithm for choosing extended instructions to minimize reconfiguration costs within loops. Our selective algorithm constrains instruction choices so that significant speedups are achieved with as few as 4 moderately sized programmable functional units, typically containing less than 150 look-up tables each.

1 Introduction

General-purpose instruction sets are intended to implement basic processing functions while balancing the needs of many applications. Complex instructions that might accelerate one application are often unused by several other applications. Worse, their implementation difficulties may impact all programs by degrading clock rates or using up vital chip area.

Configurable hardware allows one to implement complex operations on an as-needed basis, one application at a time. In recent years, configurable computing based on Field-Programmable Gate Arrays (FPGAs) has been the focus of increasing research attention. The circuit being implemented can be changed simply by loading in a new set of configuration bits. Various architectures for FPGA-based computing have been proposed, ranging from co-processor boards accessed via the I/O bus, to relatively fine-grained structures accessed as an integral part of the CPU's data path. The approach we explore here is closest to the latter architecture. We envision programmable functional units (PFUs) with 150 CLBs or less which are built into the datapath of a modern superscalar processor, and which can access the register file and result bus just like other functional units in the machine.

Customized complex or extended instructions have several advantages over traditional instruction sets. First, customization allows one to match the flow of values within an extended instruction to the needs of the particular operation being performed. Second, one can customize the bitwidth of calculations to tightly match the needs of the particular application. Third, one can improve instruction-level parallelism (ILP) by amortizing the per-instruction cost of fetching, issuing, and committing over more work.

While these advantages are compelling, customized extended instructions cannot be applied universally. First, since the PFU is part of the datapath, increasing the number of inputs to a PFU also increases the number of register file ports needed by the processor. This increases machine complexity and may impact the cycle time. Second, reconfiguring a PFU for a particular extended instruction requires fetching configuration bits and sending them to the PFU. This reconfiguration latency warrants care in choosing to implement operations as extended PFU instructions.

With this in mind, we devised and modeled T1000, an out-of-order issue, superscalar processor with programmable functional units. Initial performance studies with a simple instruction selection algorithm shows 4-44% speedups for the MediaBench suite [1] when ignoring the reconfiguration penalties. To improve speedups under more realistic assumptions, we developed a selective approach for determining which extended instructions to implement and when to use them. The key difference of our work from previous work is to check many possibilities of converting an instruction sequence to valid extended instructions. The extended instructions chosen by our selective algorithm can typically fit in a PFU composed of <150 Xilinx look-up tables. Thus, PFUs can help improve performance without adding significantly to the area of the processor.

In the remainder of the paper, Section 2 shows our proposed architecture and its extended instruction-encoding format. Section 3 describes the evaluation methodology. Section 4 discusses a greedy instruction selection algorithm and gives initial performance data. When the number of PFUs is limited, however, greedy instruction selection results in too much reconfiguration overhead. For this reason, Section 5 introduces our selective algorithm which reduces the overhead caused by reconfiguring the PFUs too frequently. We discuss the PFU hardware cost in Section 6. Section 7 describes prior work and Section 8 offers conclusions.

2 T1000 Architecture[1]

The main contribution of our work is developing selective algorithms for harnessing fine-grained configurable hardware units. To evaluate this work, we place it in the following architectural context. The T1000 architecture we model is a reconfigurable computing architecture embedded into a superscalar, out-of-order issue CPU, as illustrated in Figure 1.

2.1 Background

PRISC, the first architecture with fine-grained configurable hardware units, was proposed by Razdan and Smith [2,4]. This architecture includes programmable functional units, or PFUs, attached directly to the datapath of a simple pipelined,

[1] T1000 is the name of the “liquid metal” morphing robot in the movie “Terminator 2” starring Arnold Schwartzenegger. We chose the name because, like configurable hardware, the T1000 can change its shape into arbitrary forms.

single-issue in-order processor. The PFU was dynamically configured to implement different extended instructions specialized to the applications.

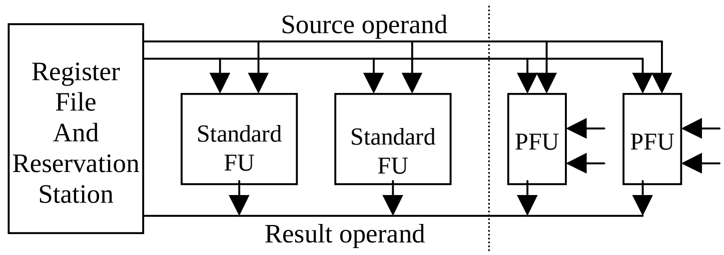

Fig. 1. T1000 architecture. To the left of the dotted line is a fixed RISC processor with out-of-order issue of four instructions per cycle

An extended instruction is created at compile time by converting an appropriate instruction sequence in the compiled code into a single PFU opcode. For example, a sequence of 3 data-dependent logic operations could easily be implemented more efficiently by a PFU based on lookup-tables than by a sequence of RISC instructions. If a PFU's specialized hardware evaluates the equivalent extended instruction in a single cycle, as compared to 3 cycles on traditional hardware, then there is a potential saving of 2 cycles each time the sequence is encountered.

2.2 T1000 Details

Like PRISC, T1000 relies on programmable functional units integrated into the datapath. A key difference from PRISC is that T1000 is assumed to be a 4-issue out-of-order machine. This is a more stringent test of the benefits of PFUs, since out-of-order issue helps tolerate the latencies of some data-dependent instruction sequences.

A T1000 extended instruction is encoded as a register-register operation with a specific opcode. We use a MIPS-like encoding format with an additional *Conf* field to control the loading of configuration bits. It defines the functionality of the PFU corresponding to that extended instruction. In the decode stage, this configuration signal is compared with the ID tag saved in each PFU. If the check shows that the correct extended instruction is currently configured in one of the PFUs, the extended instruction is dispatched normally. (This is akin to a cache hit.) Otherwise, the configuration bits must be loaded into one of the PFUs based on a LRU replacement policy before the extended instruction can be issued. A second key difference from prior work is our selective algorithm for choosing when to use extended instructions. This is described and evaluated in Section 5.

3 Methodology

3.1 Performance evaluation

To evaluate our selection algorithms and architecture, we developed a simulator based on the SimpleScalar tool set [5]. The simulator models an out-of-order issue, superscalar architecture coupled with PFUs. Out-of-order issue is managed using a Register Update Unit (RUU) [6]. The RUU scheme uses a reorder buffer to automatically rename registers and hold the results of pending instructions. The simulated processor can fetch, decode, issue, and commit four instructions per cycle and is simulated with perfect branch prediction. We simulate realistic instruction, data, and second-level unified caches, as well as instruction and data TLBs.

Each extended instruction corresponds to a set of conventional RISC instructions and is defined at compile time. The simulator takes as input SimpleScalar PISA object code files. A second input file specifies the instruction sequences that have been selected as extended instructions. Algorithms for selecting these sequences are given in Sections 4 and 5. Each extended instruction is fetched, issued and committed as a single instruction. We currently assume each extended instruction finishes execution in single cycle, and we choose sequences for which this assumption is valid, but this could easily be altered to allow for varying execution times for different extended instructions. With out-of-order issue hardware, varying execution times do not present a serious implementation obstacle.

In this study, we focus on the MediaBench benchmark suite [1]. The prevalence of narrow-width operations in multimedia processing applications makes them particularly suitable for a reconfigurable computing architecture. To collect data, we run each of the applications to completion.

3.2 Hardware Cost

The extended instructions we choose must fit within the resources of a PFU. We use the Configurable Logic Blocks (CLBs) of Xilinx XC4000-series devices as our target style of programmable hardware. To evaluate the performance and area needs of potential extended instructions, we implemented them in VHDL. The Xilinx Foundation tool then synthesizes and implements the design in an FPGA [7].

4 Potential Performance Payoff of Aggressive Instruction Selection

We evaluate the performance benefits of our selection algorithms in two steps. First, Section 4 explores the performance gain possible when we: (1) aggressively map as many extended instructions as possible, (2) use an unlimited number of PFUs, and (3) ignore any PFU reconfiguration overheads. While such assumptions are clearly not realistic, they help one see the best-case performance. This, in turn, guides the choices we describe in Section 5 to develop a more selective mechanism for choosing extended instructions that works well under more realistic assumptions.

Our greedy selection algorithm chooses all extended instructions that satisfy the following three criteria. First, the sequences are composed of fixed instructions marked as “candidates” by the profiling tool. The profiling tool is based on SimpleScalar’s sim_profile, and generates detailed profiles on operand bit-width and instruction execution time. For these experiments candidates are arithmetic and logic instructions with bitwidths of 18 bits or less, but this is a parameter that can be varied. Second, because of limitations on the register file read and write ports, instruction sequences must use at most two input registers and produce at most one output. Third, we look for “maximal” instruction sequences that take as long as possible to execute on the base RISC machine. This characteristic means that the sequences will have as many dependent instructions as possible.

When selecting candidate sequences, the greedy algorithm does not consider the number of PFUs available or the time spent reconfiguring PFUs. Since we evaluate it using an unlimited number of PFUs with zero reconfiguration costs, it represents a best-case performance estimate for this style of compiler and architecture.

4.1 Performance Results Using the Greedy Selection Algorithm

For MediaBench, the greedy algorithm identifies between 6 and 43 distinct extended instructions, and sequence lengths range from 2 to 8 instructions. The first and second bars per program in Figure 2 shows performance results (for unlimited PFUs and no reconfiguration cost) compared to the case of a superscalar processor with no PFUs with the characteristics given in Section 3. Speedups range from 4.5% for *g721-decode* to 44% for *gsm-decode*.

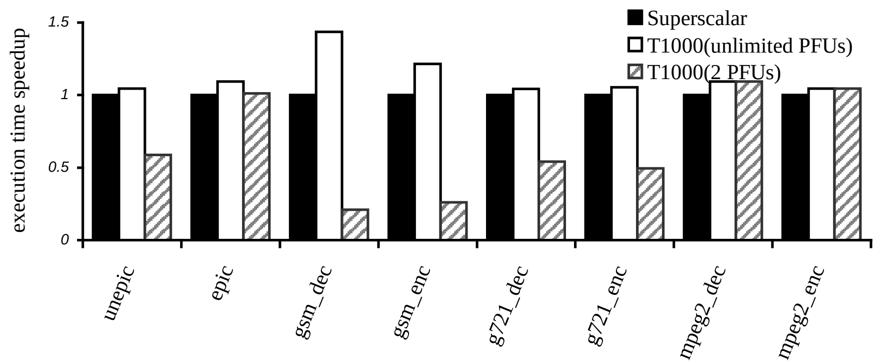

Fig. 2. Speedups using PFUs assuming a greedy selection algorithm. The first bar shows baseline speedup of a superscalar processor with no PFU, normalized to 1. The second bar for each application is the T1000 speedup with unlimited PFUs and zero reconfiguration cost. The third bar shows T1000 speedup with 2 PFUs, each with a 10-cycle reconfiguration penalty

Although the previous paragraph assumed that PFU configurations could be swapped in zero time, real chips have considerable configuration overhead. Thus, the third bar per application in Figure 3 illustrates the performance of the greedy algorithm on more realistic hardware. We consider a superscalar processor with only 2 PFUs, each with a 10-cycle reconfiguration latency. This reconfiguration time is fairly optimistic, but it is sufficient to show the negative impact reconfiguration penalties can have on performance. We use the same set of the extended instructions extracted by the greedy algorithm in the optimal experiment. The results here are quite discouraging. The reconfiguration penalty causes the performance with PFUs to be substantially worse than that of the original processor! Essentially, the PFU is thrashing. Each time an extended instruction is to execute, it causes the PFU to be reconfigured, and the greedy algorithm is too aggressive in selecting extended instructions. These disappointing results motivate the following section in which we present the main contribution of this work: a selective algorithm for choosing extended PFU instructions.

5 A Selective Algorithm for Choosing Extended Instructions

Section 4's results demonstrate that PFUs can improve performance greatly, but only if extended instructions are selected appropriately. In choosing extended instructions, we must not only consider their potential gains, but also minimize the incurred reconfiguration penalties.

5.1 Selective Algorithm Overview

Our selective algorithm works to maximize the benefits from extended instructions subject to the constraint that there are a finite number of PFUs and reconfiguring between different extended instructions takes time. The selective algorithm consists of the steps illustrated in Figure 5.

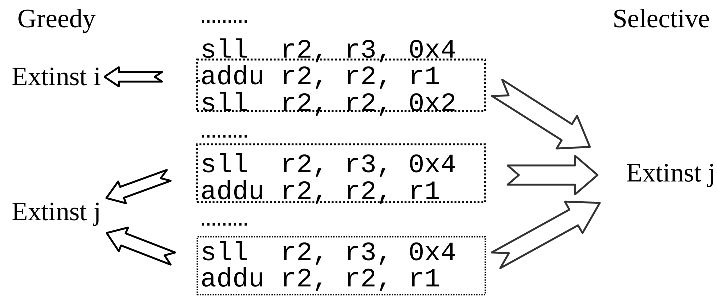

Fig. 3. Example of extracting common subsequences instead of maximal sequences

		i		j	
i		1		0	
j		1		2	

Fig. 4. Matrix for evaluating common subsequences in the selective algorithm

The algorithm has two key characteristics. First, we profile the program and determine which candidate sequences are responsible for more than 0.5% of the total application time. This step allows us to focus on the sequences that offer the highest payoff. It also limits the number of distinct instruction sequences considered, which reduces the likelihood of "thrashing" by too frequently reconfiguring PFUs. Second, the number of extended instructions selected within each loop never exceeds the number of PFUs. This step avoids the frequent reconfigurations that lead to thrashing when the greedy algorithm is used with limited PFUs.

A key difference between the selective algorithm here and the greedy algorithm in Section 4 concerns the treatment of instruction subsequences. Consider the case shown in Figure 3 with three maximal sequences in the same loop. The greedy algorithm would implement all three instruction sequences as extended instructions. Here, we determine when to choose a smaller common subsequence as opposed to instantiating all the possibilities.

Our approach begins by extracting all valid subsequences and adding them to the candidate extended instruction list. The list is organized as a k×k matrix, where k is the total number of distinct candidate instruction sequences in the loop. The [I,J] entry of the matrix corresponds to the number of appearances of candidate sequence I within all candidate sequences J throughout the loop. The sum of entries along the Ith row equals the total number of appearances of sequence I throughout this loop. The [I,I] entry corresponds to how often maximal sequences of I appear: cases where the Ith sequence appears alone and is not a subsequence of a larger instruction.

The matrix for the example in Figure 3 is shown in Figure 4. There is a 1 in the [I,I] entry of the matrix to denote the fact that the first candidate sequence from Figure 3 has one maximal appearance in the loop. Since the latter two sequences in Figure 3 perform the same operation, they share an identical PFU configuration. (This is true in both the greedy and selective algorithms.) They occupy row J in the matrix in Figure 4. A 2 in the [J,J] entry of the matrix indicates that there were two maximal appearances of this extended instruction in the loop. The key leverage of our algorithm is that the [J,I] entry of this matrix reflects the fact that sequence J also appears as a subsequence of I and thus is set to 1.

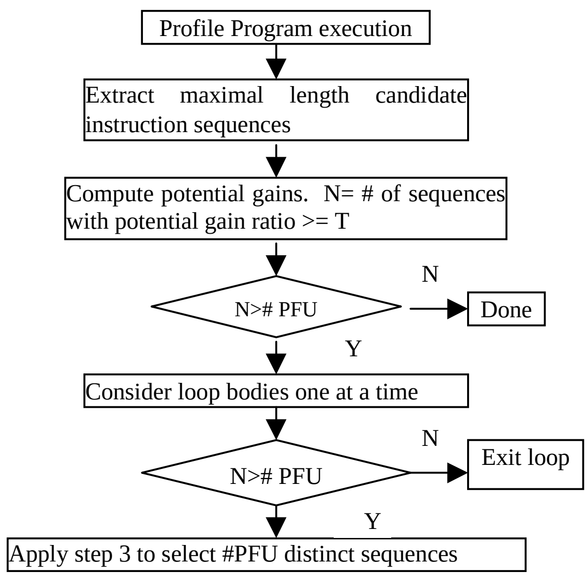

Fig. 5. Flow chart of selective algorithm

The final appropriate extended instructions are then selected from the list by comparing their potential gains. For example, sequence J appears a total of 3 times in the loop, each with a potential gain of 1 cycle. By contrast, sequence I appears only once, but with a potential savings of 2 cycles. If we are working with an architecture with only one PFU, then selecting the sequence with the highest total gain across the loop would lead us to choose sequence J.

5.2 Performance Improvements Using the Selective Algorithm

Figure 6 shows that the selective algorithm successfully chooses extended instructions that offer speedup by avoiding reconfiguration penalties as much as possible. Speedups for these benchmarks now range from 2-27%. Since our approach reduces dramatically the number of PFU reconfigurations, the reconfiguration penalties only account for a small fraction of total potential gains. In fact, our experiments show that we retain our excellent speedups even with reconfiguration times as high as 500 cycles.

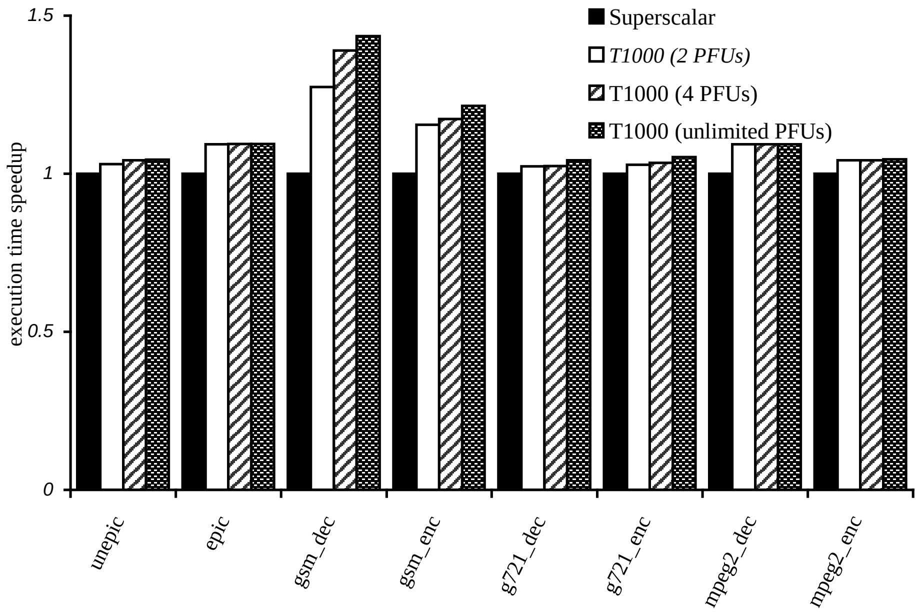

Fig. 6. Speedups achieved using the selective algorithm. For each benchmark, the second and third bars correspond to T1000 with 2 and 4 PFUs, respectively. The fourth bar models unlimited PFUs. A 10-cycle reconfiguration cost is assumed in all cases

Our selective algorithm also adjusts itself well to the number of PFUs available. Overall, we find that four PFUs are typically enough to achieve almost the same performance improvement as the optimistic speed-ups presented in Section 4. Figure 6 illustrates the results with 4 PFUs and compares them to the previous optimistic results achieved with unlimited number of PFUs.

6 Configurable Hardware Cost

The basic component of the PFU is a configurable logic block consisting of look-up tables (LUTs) and flip-flops. An N-input look-up table can implement any Boolean function of N inputs. The LUT propagation delay is independent of the function implemented. In this paper, we use standard CAD tools to map extended instructions to Xilinx devices in order to estimate the PFU hardware cost.

Figure 7 presents the area distribution of instructions chosen by our selective algorithm for the 8 benchmarks. The configurable hardware resources required by an extended instruction depend both on the type of operation and also on the operand widths. Quite a few of the extended instructions need very little hardware, largely due to profiling that indicates when they can be implemented with narrow-bitwidth inputs. On these examples, the most area-intensive extended instruction needs 105 LUTs.

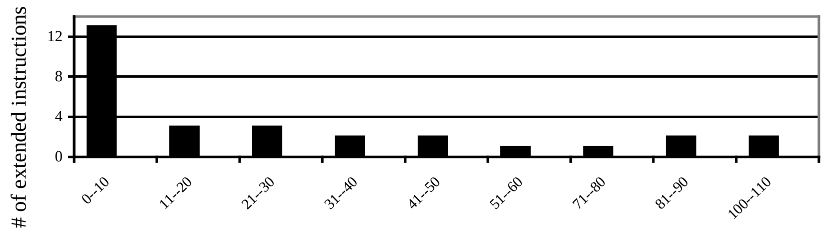

Fig. 7. Distribution of hardware requirements for the extended instructions extracted from 8 MediaBench benchmarks by our selective algorithm

7 Prior Work

There has been a large amount of work on the reconfigurable computing architectures with customizable instruction sets, and an exhaustive summary is difficult. Instead, we present some representative work categorized by the degree of coupling between the configurable hardware resources and the base processor.

Coarse-grained architectures include SPLASH1, SPLASH2 [8] and PAM [9]. In these, the configurable hardware resources are connected as a co-processor on the I/O bus of a standard microprocessor. While appropriate for coarse-grained problems, the disadvantage of these board-based systems is that they have high communication latencies and configurable hardware cost.

Medium-grained architectures include NAPA [10]. In NAPA, the Adaptive Logic Processor (ALP) can access the same memory space as the Fixed Instruction Processor (FIP), so the communication overhead between the ALP and the FIP is reduced compared with the coarse-grained architectures, but this approach still does not give the ALP full access to the register file.

Fine-grained architectures include the PRISC work [2,4]. PRISC was proposed to be a simple, pipelined, single-issue processor augmented with a single PFU. Because of the tight coupling between the PFU and the base CPU, PRISC requires only a small amount of configurable hardware resources and minimizes communication costs. Other representatives of this class include CoMPARE [11] and OneChip [12], etc. CoMPARE explores the impact of multiple PFUs and can execute RISC instructions and customized instructions concurrently. OneChip is an embedded system. It requires more functional modules to be implemented on PFU, which in turn introduces larger communication penalties. All of the above fine-grained architectures were evaluated on simple, in-order-issue, single-issue processors. The impact of PFUs on a superscalar processor's performance is different from that on a simple processor, and our work has quantified these differences.

8 Conclusions and Future Work

This work has explored the use of application-specific instructions in the context of modern superscalar architectures. In particular, we have proposed the T1000 architecture which adds programmable functional units (PFUs) into the datapath of a wide, out-of-order issue processor. These small configurable functional units based on FPGA-like technology have the potential to greatly improve performance. Our initial optimistic studies showed up to 44% performance improvements in some cases.

A key issue in using a small number of PFUs effectively is devising a selection algorithm that is both aggressive enough to uncover speedup opportunities, and yet also conservative enough to avoid cases where PFUs "thrash" as they frequently reconfigure back and forth to handle many selected configurable instructions.

With the goal of avoiding PFU thrashing, we developed and evaluated a selective algorithm for choosing instruction sequences for configurable implementation. Our choice is guided by the number of PFUs available and simple execution profiles of the program loops. This allows us to aggressively select configurable instructions that offer the largest performance savings with the smallest hardware needs. With this algorithm, we have shown performance improvements of up to 28% with 2 PFUs compared to simple superscalar processors without PFUs. Furthermore, our selective algorithm is so successful at avoiding PFU thrashing that these speedups are largely independent of the PFU's reconfiguration overhead. We view our work as a proof-of-concept demonstration that PFUs can offer worthwhile performance improvements in modern high-performance superscalar architectures.

References

1. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, MediaBench: A Tool for Evaluating Multimedia and Communications Systems. *Proc. Micro 30*, 1997
2. R. Razdan, and M.D. Smith: A High-Performance Microarchitecture with Hardware-Programmable Functional Units. *Proc. 27th Intl. Symp. On Micro*, pp. 172-180, Nov., 1994.
3. Xilinx Inc. *The Programmable Logic Data Book*, Xilinx 2100 Logic Dr. San Jose, CA 1998
4. R. Razdan, K. Brace, and M. Smith. PRISC Software Acceleration Techniques. *Proc.Int. Conf. on Computer Design.* Oct.1994.
5. D. Burger, T.M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. TR-1308, Univ. of Wisconsin-Madison CS Dept., July 1996
6. G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Trans. on Computers*, 39(3): 349-359, March 1990
7. Xilinx Inc. Foundation Series Quick Start Guide 1.5, Xilinx 2100 Logic Drive. San Jose, CA
8. J. Arnold et al. The Splash 2 Processor and Applications. *Proc. Int. Conf. on Computer Design*, Oct 1993
9. P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. *Systolic Array Processors*, J. McCanny et al. Eds., Prentice Hall, 1989
10. C. R. Rupp and M. Landguth et al. The NAPA Adaptive Processing Architecture. *Proceedings IEEE Symp. on FPGAs for Custom Computing Machines*. Napa Valley, CA, USA 15-17, April 1998
11. S. Sawitzki, A. Gratz and R.G. Spallek: Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Array, *Proc. of Field-Programmable Logic and Applications*, Tallinn, Estonia, August 1998
12. R. D. Wittig and P. Chow: OneChip: An FPGA Processor With Reconfigurable Logic, *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, CA, April 1996