Research Article

On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration

P. Guironnet de Massas, P. Amblard, and F. Pétrot

TIMA Laboratory, 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France

Received 31 October 2006; Revised 19 January 2007; Accepted 30 May 2007

Recommended by Wiesław Kuzmicz

This paper presents the necessary steps to modify the implementation of the SPARCV8 architecture to enhance it with multimedia-oriented instructions. The purpose is improving video compression performance without designing dedicated coprocessors. We investigate the complexity of modifying a standard processor instruction set and show that, although not trivial, this is feasible in a few weeks. We implemented 12 new instructions and use some of them to optimize the computation of a demanding step of the MPEG encoding. The result is a performance increase of 67% in the execution of a part of this algorithm, allowing us to expect a 30% speedup in the execution of an MPEG video compression. The area increase of the integer unit is about 18% and the clock frequency is not significantly modified in an LEON-2 implementing 6 among 12 of the new instructions.

Copyright © 2007 P. Guironnet de Massas et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

The consumer electronic market is pushing towards very high-performance and low-cost devices capable of coding or rendering pictures, video, voice, and music. The multimedia formats required by these appliances make use of very computationally hungry compression and coding algorithms, that must be achieved in real time either by the use of high-end processors, dedicated hardware, or processors that embed application domain specific instructions.

High-end processors are not viable in the consumer market, because of their size, cost, and power consumption. Dedicated hardware is certainly a possibility, because it provides the best MIPS per Watt ratio. However, the mask cost makes this solution suitable only for very wide markets and its hardwired nature tunes it for a single application. The last solution can be further divided into two cases: application specific processors, that is, DSP, or application domain specific instructions added in a low-cost general purpose processor core. This is the solution investigated in this paper. The emergence of MPSoCs based on low-cost processors on the same chip tends to make this approach interesting in practice.

All multimedia streaming applications share a common characteristic, they compute a big amount of independent small sized data. Providing single instructions multiple data

(SIMD) instructions is a good way for improving performance in case of generic purpose processors. This may be implemented in a coprocessor or, when it is possible, in the processor data-path itself.

In a general manner, we intend to show that optimizing a precise algorithm with some new dedicated instructions can be done by simply adding them to the execution flow of a standard general purpose processor. We aim to show that tuning a standard processor can be achieved at low cost in a short time. Therefore, our contribution is an implementation feasibility study of the addition of SIMD instructions within a classical RISC processor.

In this work, we introduce new instructions in the SPARC V8 instruction set architecture (ISA) to optimize video encoding algorithms for low-cost, possibly multiprocessor, embedded platforms. The hardware implementation is performed using the LEON-2 open source V8 compliant processor, “open source” being a requirement in our academic environment. The paper is organized as follows. Section 2 presents an overview of the different works in the field. Section 3 introduces the constraints on the architecture and design. Section 4 details the key points of adding new instructions. Section 5 presents the experimental setup to optimize the sum of absolute differences (SAD) algorithm, and Section 6 presents the results of our experimentation. Finally, we conclude.

2. RELATED WORKS

In the area of processors that embed application domain specific instructions, there are a lot of existing works. We can find two kinds of extended instruction sets architectures: the first one is the domain oriented (e.g., multimedia) extensions and the other is more application tailored extensions. We present these two categories hereafter.

2.1. Generic or domain specific instruction set extensions (ISE)

Almost each widespread high-performance architecture has now been extended with domain-oriented instructions. Some common examples are the ISA extensions MMX [1], sse, sse2, altivec, 3dnow, or VIS [2]. A performance comparison of such extensions for high-end processors is given in [3].

These extensions are suitable to optimize applications in the instruction set scope. However, to the best of our knowledge at the beginning of this work, there is no available RTL description of a processor with extended instruction set. Moreover, optimizing a specific application on a heterogeneous platform may require some highly tailored instructions that are not present in the available ISEs.

2.2. Application specific instruction sets architectures

There exist some commercial solutions to customize a generic processor with some added hardware allowing it to execute application specific instructions. This can be done using an instruction-set description language like the *Tensilica instruction extension (TIE)* [4, 5] that allows to extend a Tensilica processor [6] with application specific instructions. In studies done by the HP labs [7, 8], an application-directed configuration of nonprogrammable, VLIW-based processor accelerators was investigated.

The major drawback of such solutions is that the tailored processor remains an IP, hence some aspects cannot be tuned to fit specific requirements. Other works were made tailoring open source processors. In [9] a modified LEON core including new instructions and registers was defined. This was done to allow optimized communication and synchronization with a coarse-grained reconfigurable architecture called XPP. The integration of these modifications did not show major instruction execution flow issues and therefore it did not demonstrate the general tailoring potential of such processor. In [10], an approach to optimize a specific cryptography algorithm was demonstrated. However, the processor modifications were very simple and did not show the possibilities provided by the enhancement of such architectures with complex instructions.

Compared to the previous works, we present a solution that allows to optimize the runtime of a specific algorithm by adding new instructions to a general purpose processor (LEON-2). We also show that this is feasible in a short time even when the instructions are complex and require fine grained comprehension of the pipeline flow, bypass mechanisms, and multicycles execution. This solution is not ex-

clusive with coprocessor design or ad hoc IP's integration. In fact, in the ever growing MPSoCs field, there exists heterogeneous architecture that may combine application specific IPs with general purpose processors and several DSPs. For example, the Nomadik [11] platform includes ARM processors with IP- and DSP-based audio and video accelerators.

Even though the RT level synthesizable code of the processor is required to add new instructions, the proposed solution is not restricted to the LEON processor. In fact, there are several processors which have their full description code published under GPL/LGPL like licences. To name a few, we have the openSPARC T1 [12], the openrisc [13], the LEON SPARC V8/V9 [14]. The microblaze source code is available with licence fees [15] and the POWERPC 405 [16] processor code is licenced for free for educational purposes. These free descriptions have an ever growing place in the research and industry, specially the LEON processor [17–19].

3. CONSTRAINTS

3.1. Use of the SPARC V8

We choose to work on the SPARC V8 processor for two reasons. The first one is that the ISA is free of patents and thus implementations of the ISA do not need licences. The second reason is that there exists an implementation of the SPARC V8 architecture called LEON-2, which VHDL source code is available under the GPL/LGPL licences. Due to the open nature of this licence, we are able to base our work on a well known implementation.

The LEON-2 processor is an industry level, complete system on-chip since it includes caches, I/O controllers for Ethernet and PCI port, local embedded memory, and an AMBA AHB bus. The core contains a five-stage pipelined integer unit, described in Figure 1, separate instruction and data caches and has a communication port with a coprocessor and a floating point unit. We work exclusively on the integer unit, hence our work only focus on this component and its issues. The source code is fully synthesizable using commercial logic synthesis tools, allowing accurate performance and area estimations.

3.2. Other constraints

This experimental study was made in a strongly constrained framework. Before starting the experimentation by itself, we chose a number of restrictions. They are presented and explained hereafter.

- (i) Our new SPARC must be compatible with Version 8 specifications [20]. All the instructions of V8 remain implemented.
- (ii) We use the fact that in this V8 some binary instruction encodings are not valid. When executed, they generate an *illegal.instruction* trap. As a consequence, a correct program must not use these illegal instructions to perform system calls. System calls must be done via a `ti cc` instruction like `ta`. The new instructions added to the

FIGURE 1: Integer unit data path of LEON-2 core.

instruction set will only use these previously nonvalid operation codes.

- (iii) We do not suppress the `smac`, `umac` instructions present in the LEON-2 to remain compliant with specific programs which use them.
- (iv) We do not want to implement the new capabilities using a dedicated coprocessor. Our new instructions are put at work in the processor itself, without any communication overhead between the processor and a coprocessor.
- (v) We do not modify the flow of the other instructions. Obviously LEON is the result of a long (and good) work. No changes are necessary. The pipeline is not modified, and all the existing V8 instructions remain with the same number of execution cycles. However, we allow our new instructions to execute in several cycles if necessary.
- (vi) The modifications are to be made in the VHDL description and must be in the same writing style
- (vii) Our simulations of the new instructions and the performance evaluation will be done at the register transfer Level, on the synthesizable VHDL. Obviously, comparative studies of instruction sets efficiency should be

done with instruction set simulators at a higher level of abstraction.

- (viii) We do not intend to write compilers taking our instructions into account, but optimized techniques to do so exist [21].

These constraints imply that the target application must be optimized with only a few new instructions. For example, there are only 8 nonimplemented opcodes in the arithmetic and logic instructions family.

4. KEY POINTS OF THE IMPLEMENTATION PROCESS

As our work is not about how to optimize the instruction set itself but about implementing new instructions, we assume that the application was profiled and a subset of new instructions was defined to be implemented. The following key points concerns the implementation difficulties and characteristics of possible new instructions.

4.1. Map new instructions onto the available opcodes

In the V8 ISA, there are 6 instruction categories. Control transfer, load/store, and arithmetic/logic/shift instructions

		op3 [5:4]			
		0	1	2	3
op3 [3:0]	0	ADD	ADDcc	TADDcc	WRASR WRY
	1	AND	TSUBcc	ANDcc	WRPSR
	2	OR	ORcc	TADDccTV	WRWIM
	3	XOR	XORcc	TSUBccTV	WRTBR
	4	SUB	SUBcc	MULScc	FPop1
	5	ANDN	ANDNcc	SLL	FPop2
	6	ORN	ORNcc	SRL	CPop1
	7	XNOR	XNORcc	SRA	CPop2
	8	ADDX	ADDXcc	RDASR RDY STBAR	JMPL
	9			RDPSR	RETT
	A	UMUL	UMULcc	RDWIM	Ticc
	B	SMUL	SMULcc	RDTBR	FLUSH
	C	SUBX	SUBXcc		SAVE
	D				RESTORE
	E	UDIV	UDIVcc		SMAC
	F	SDIV	SDIVcc		UMAC

FIGURE 2: Table of arithmetic and logic opcodes of LEON ISA.

are example of categories. Firstly, we classify the new instructions in these subsets. Secondly, for a given subset, each new instruction must be mapped on an unimplemented opcode (see Figure 2). This step is mandatory to minimize the area and path overhead of the decoding stage.

4.2. Take care of pipeline structure and provided logic

The pipeline execution flow allows (or can be modified in order to) multicycles instructions. For example, in the V8 ISA, the instruction that load a double word takes 2 apparent execution cycles. It is easy to modify the execution flow in order to create any kind of multicycles instructions. Since the execution is generally sequential, implementing a 500 cycles instruction may not be a good idea. It can surely be done more efficiently by implementing a hardware IP or coprocessor. As said before, these two solutions are not exclusive, and taking the best of both worlds can allow designers to create very powerful architectures like in [9].

The pipeline execution flow implements data bypass from stage to stage. Hence, multicycles instructions can work incrementally on the result of a previous iteration. It is up to the designer imagination to take advantage of this characteristic. For example, we can imagine an instruction that performs indirect load on a traditional RISC architecture. A loaded word on the memory stage of the pipeline (Figure 1) can be bypassed to the execute stage to process a new address and indirectly load a data.

4.3. Instruction set characteristics

To avoid excessive overhead when adding new instructions, we must take care of the instruction set characteristics. For

example, typical RISC instructions use 3 operands (two arguments and a result). This implies that the scope of possible new instructions is limited. Implementing an instruction that process on a register set must be implemented as a multicycles instruction since typically in RISC architectures only two registers can be read at the same time (and only one can be written). Example of instructions that load or store a set of registers can be found in the POWERPC (`lmw`: load multiple word, and `stmw`: store multiple word) or ARM (`ldm` and `stm`), but not in the SPARC ISA.

4.4. New processing modules

Adding new processing modules allows us to design more complex instructions. However, we must take care of the critical path and logic overhead at implementation time. It may be a good idea to combine parallel execution in new modules and in the already available ones. The new modules must interact with other ones using existing temporal registers and control logic, allowing a minimal logic overhead and design simplicity.

In spite of the presented limitations and constraints, combining all the processing elements and pipeline characteristic allows us to define a large scope for the instructions that can be implemented in a short time. The instructions that cannot be implemented in the pipeline itself can be integrated in a hardware IP, coprocessor, or another modified processor. We can also imagine multiprocessor architectures built using only low-cost general purpose processors, each one being tuned for a specific class of algorithms or even a specific task.

FIGURE 3: Example of SIMD sum.

5. EXPERIMENTATIONS

5.1. Experimental setup

The experiments are performed using a processor, its caches and a local memory. Our goal is to enhance the processor performances in an MPEG-like video encoding algorithm. Consequently, we start with a summary on SIMD instructions and multimedia-oriented algorithm specificities. Then, we present a subset of new instructions and their implementation in order to enhance video encoding performances of the LEON-2 processor with a low area overhead. In this work, we focus on the optimization made to a given task of a video encoding algorithm which use only 4 of the 12 defined instructions. The other ones can be used in other steps of the encoding algorithm.

5.2. Application characteristics

5.2.1. SIMD instructions

Most data streams are composed by 8- or 16-bit coded values. In general purpose processors, the typical size of the registers is 32 or 64 bits. Instead of dealing with 32-bit vectors, the SIMD instructions deal with the 48-bit vectors independently. The idea is thus to use the 32-bit registers to store 4 vectors of 8-bit data and implement new instructions to perform computations on these 8-bit data vectors independently. Figure 3 shows an example SIMD instruction: the sum.

5.2.2. Characteristics of multimedia-oriented applications

In video and multimedia applications, we handle huge quantities of small-sized independent data. The use of SIMD instructions is well adapted to this characteristic, allowing huge improvement in computation speed.

In the motion estimation stage of a video encoding algorithm, the kernel often contains a sum of absolute differences (SAD). The main idea of this algorithm is to compare two frame blocks (8-bit data vectors) by accumulating their differences (Figure 4). This stage is very time consuming, hence we focus our work on this SAD algorithm in order to improve the LEON-2 processor video encoding performances. Since there are no data dependencies in this computation, a big improvement may be obtained by performing several absolute differences at the same time.

5.3. Our new instructions and their implementation

5.3.1. The instructions and their functional description

We have defined 12 new instructions. Almost all of them exist in the others extensions sets like MMX or VIS. These instructions can be classified into 4 categories:

- (i) 9 adder-based arithmetic instructions (1 cycle execution time): `diffabs`, `addp8`, `subp8`, `addp16`, `subp16`, `addp8Sat`, `subp8Sat`, `addp8SatS`, and `subp8SatS`;
- (ii) 1 packing instruction (1 cycle execution time): `pack`;
- (iii) 1 instruction for accumulation (3 cycles execution time): `addac`;
- (iv) 1 load/store instruction (3 cycles execution time): `ldwna`.

To give the functional description of the instructions, we use the common notation: reg_d is the destination register and reg_{s1}, reg_{s2} are the two source registers. We also use little-endian notation for the registers content. In Figure 5, there are execution examples of the instructions presented hereafter.

`addp8/subp8/addp8Sat/subp8Sat/addp8SatS/subp8SatS`: perform, respectively, the sum and difference byte per byte. The carry of a byte sum is not propagated to the next one. The instruction was implemented in both saturated and nonsaturated modes:

$$\begin{aligned} reg_d[31, 24] &\leftarrow reg_{s1}[31, 24] + reg_{s2}[31, 24], \\ reg_d[23, 16] &\leftarrow reg_{s1}[23, 16] + reg_{s2}[23, 16], \dots \end{aligned}$$

`Sat` and `SatS` stand for saturated and signed saturated operations.

`addp16/subp16`: same operation as above but between 16-bit data.

`diffabs`: this operation computes the absolute difference of the 4 bytes in two 32-bit registers. $reg_d[8(i+1) - 1, 8i] \leftarrow abs(reg_{s1}[8(i+1) - 1, 8i] - reg_{s2}[8(i+1) - 1, 8i])$ with $0 \leq i < 4$.

`pack`: this instruction allows to rearrange the 8-bit and 16-bit data contained in a general purpose register (32 bits) using a mask contained in another one. Defining $sel(i) = reg_{s1}[8(i+1) - 1, 8i]$ with $0 \leq i < 4$, the `pack` instruction performs

$$\begin{aligned} ®_d[8(i+1) - 1, 8i] \\ &= \begin{cases} reg_{s2}[8(sel(i) + 1) - 1, 8sel(i)] & \text{if } 0 \leq sel(i) < 4, \\ 0. & \end{cases} \end{aligned} \tag{1}$$

The second case of the rule allows us to copy only a part of a register in another one, and hence to easily translate matrixes, rearrange vectors, split, and reorganize data. The data are considered unsigned, hence translating a 16-bit data to a 32-bit register will not perform sign-bit propagation.

`ldwna`: this instruction loads a word with a nonaligned address from the memory. This instruction is very useful in frame computation, as it allows to load contiguous pixels and

FIGURE 4: SAD algorithm applied to two 4×4 frames macroblocks.

00 03 ff 01	←	ff fe 03 0f	addp8	01 05 fc f2
fe f9 00 00	←	ff fe 03 0f	subp8sat	01 05 fc f2
01 03 00 01	←	ff fe 03 0f	addp16	01 05 fc f2
ff ff ff ff	←	ff fe 03 0f	addp8sat	01 05 fc f2
70 80 7f 80	←	20 c0 20 c0	addp8satS	50 c0 60 b0
fe f9 f9 e3	←	ff fe 03 0f	diffabs	01 05 fc f2
fc 05 00 f2	←	01 02 08 00	pack	01 05 fc f2
00 00 fc fd	←	01 02 08 00	addac	01 05 fc f2

FIGURE 5: Execution examples of the implemented new instructions.

work on them even if they are not aligned in memory (a very common case).

addac: sum the four bytes of a register with the half word contained in another. The result is a halfword (16 bits):

$$\text{reg}_d[15, 0] \leftarrow \text{reg}_{s1}[15, 0] + \text{reg}_{s2}[31, 24] + \text{reg}_{s2}[23, 16] + \text{reg}_{s2}[15, 8] + \text{reg}_{s2}[7, 0].$$

This instruction is used to optimize byte sum accumulations.

5.3.2. The circuit implementation

We present hereafter the main common elements that can be seen in Figure 6.

- (i) *The component mm_adder* is a modified 32-bit adder which controls the carry propagation between each byte. Therefore, we have only one 32-bit adder in the processor because it is substituted to the original adder. It can be implemented with a modified Sklansky adder [22]. This component was inserted directly in the execution flow without major modifications,
- (ii) *The component mm_mixer* is used by the instruction `ldwna`. It has a 3-cycle execution flow. The execution of the instruction `ldwna` implies the load of two aligned words and the selection of the needed bytes owned by each loaded word. This action is made in two cycles overlapped with the load of the second word, hence the execution time of 3 cycles. With this com-

ponent, we have to deal with the multicycle pipeline controls and with all the data control problems like bypasses and interlocks. A possible optimization for this module is to store the previous memory address and the loaded bytes in order to achieve load bursts in a roughly 2-cycle per word rate,

- (iii) *The component mm_packer* is used by the instruction `pack` to demux and mux the desired bytes using the mask provided as an operand,
- (iv) *The component mm_acc* is used in the execution of the instruction `addac`. The execution of this instruction takes 3 cycles because we have to guarantee that this component will not increase the critical path of the execution stage. To compute the different sums, we use two 8-bit adders contained in this module in parallel with the adder of the IU (with control signals to make a 16-bit sum). Like *mm_mixer*, we have to deal with bypasses and multicycles controls.

Difficulties. With these categories of instructions, we had to explore different execution flows inside the integer unit. With the single cycle instructions, no major difficulties were encountered and the modifications of the IU were simple. However, with the multicycles instructions we had to deal with pipeline multicycles control, bypass and interlocks including data caches access. Since the LEON has a data cache and an instruction cache, the load or store instructions do not insert extra cycles unless there is a data dependency. Let us give two examples of execution flows.

Example of execution flow: `ldwna`

As said before, the `ldwna` instruction allows us to load a non-aligned word in 3 cycles. In Figure 7, we can see all the execution steps.

- Step 1: instruction fetch;
- Step 2: load registers contents;
- Step 3: compute the address of the first word;
- Step 4: compute the address of the second word in the execution stage, the memory stage receive the first word and sets the offset parameter of the mixer module;
- Step 5: bypass from writeback stage the first loaded word to the mixer module, operate the needed shift on it and

FIGURE 6: The modified 5 stage pipeline of LEON-2 with the new components in bold.

put the result in the y pipeline register (temporary mul/div result register not used at this time), the second word is loaded into the memory stage;

Step 6: bypass the y register and the second word to the mixer module and compute the result;

Step 7: write the result into the reg_d register.

Example of execution flow: addac

The addac sums up the bytes of a word. Its execution steps are as follows.

Step 1: instruction fetch;

Step 2: load registers contents;

Step 3: compute $\text{data1}[31, 24] + \text{data1}[23, 16]$ and $\text{data1}[15, 8] + \text{data1}[7, 0]$;

Step 4: data2[15,0] is forwarded in the decode/execute transition register and data1[15,0] + data2[15,0] is computed in the alu;

Step 5: compute $\text{result} = \text{data1} + \text{data2}$ with $\text{result}[31, 16] = 0$, at this step as shown in Figure 8 the accumulation is finished and the result will be stored into the reg_d register two steps later in the writeback stage, but the value is available as usual for forwarding.

6. RESULTS AND COMMENTS

Comparison of programs for the SAD algorithm computation

Now, we will present a simple implementation of an SAD algorithm using three of our new instructions (`ldwna`, `addac`, and `diffabs`). The main code snippets, both without and with our new instructions, are presented in Figure 9.

The optimized version is more efficient in several points.

- (1) We load the four bytes in two cycles with the `ldwna` instruction, in contrast with the four cycles needed when loading byte per byte. This assumes a cache hit for the first byte, otherwise the miss penalty are the same in both executions and the ratio of cache miss are the same.
- (2) We perform the accumulation in 3 cycles with the instruction `addac` versus the 4 cycles needed by the four `add` instructions.
- (3) In our case, for one byte, we compute the absolute difference in one cycle instead of three. In addition, since the operations are done in parallel on the 4 bytes, we need only 3 cycles instead of the original 12.
- (4) We have a shorter code; consequently, we improve instruction cache issues in a global way, along with memory usage.

FIGURE 7: Execution flow of the `ldwna` instruction.

We perform a simulation of the synthesizable RTL view of the LEON-2 enhanced with new instructions and run a SAD loop on a 4×4 frame block. We obtain an improvement of 67% in clock cycles requirement. The results are presented in Table 1.

Analysis of synthesis results

We have synthesized a modified LEON-2 used to optimize the SAD algorithm with these characteristics.

- (i) It includes 7 new instructions: `addp8`, `subp8`, `addp16`, `subp16`, `ldwna`, `diffabs`, and `addac`.
- (ii) It does not include the packaging instruction as it is not used.
- (iii) The `mm_adder` module is a second version without the instructions with saturation `addp8sat`/`subp8sat`. It is not based on a Sklansky adder, and is made of two simples 4×8 -bit carry save adders. As a consequence, the modules are smaller but with a higher propagation time. However, this is not a problem because it does not modify the critical path of the IU.
- (iv) The target technology is AMS C35 $0.35 \mu\text{m}$.
- (v) The processor was configured with 8 register windows, neither multiplier nor divider.
- (vi) The instruction and data caches are both 16 kb direct mapped.
- (vii) Area is given for a core without I/O pads.

After synthesis, the modified IU reports an area of $532\,000 \mu\text{m}^2$ to be compared with the $452\,000 \mu\text{m}^2$ of the original one. The complete processor without I/O pads reports $1\,125\,270 \mu\text{m}^2$ versus $1\,045\,390$ for the original one. The $80\,000 \mu\text{m}^2$ overhead is due to

`mm_adderV2`: $30\,000 \mu\text{m}^2$, $3\,000 \mu\text{m}^2$ per 8-bit adder, and $6\,000$ for muxes and internal logic,

`mm_mixer`: $17\,800 \mu\text{m}^2$. It includes a 32-bit register, and logic to make shifts (muxes).

`mm_acc`: $19\,000 \mu\text{m}^2$, it contains two 8-bit adders, a 16-bit register, and a few muxes.

The remaining $13\,200 \mu\text{m}^2$ are due to the added control logic inside de IU. It is important to remember that the increase of 18% of the IU represents less than 8% percent for the global processor without I/O pads. If we intend to include multiplier, divider, and other peripherals like Ethernet and PCI ports, the increase in area is marginal.

The maximum clock rates reported by the synthesis tools are 36.1 MHz for the original processor and 35.4 MHz for the modified one. The difference is marginal and the overall implementation can be optimized.

7. CONCLUSIONS

The scope of this work was to show that it is possible to obtain better performances for a precise algorithm or an

FIGURE 8: Execution flow of the addac instruction.

Standard SAD	Optimized SAD
@@ Code processing 1 byte	@@ Code processing 4 bytes
...	...
ldub [%o2], %i0	ldwna [%o5], %g3
ldub [%o3], %i1	ldwna [%o1], %g2
sub %i1, %i0, %i1	diffabs %g3, %g2, %o4
sra %i1, 31, %i5	addac %o4, %o0, %o0
xor %i5, %i1, %i1	...
add %o1, %i1, %o1	...
...	...
One absolute difference and the result accumulation takes 6 cycles assuming no cache misses, thus 24 cycles are needed to make the computation on 4 bytes.	The computation of 4 absolute differences and result accumulation takes 10 cycles assuming no cache misses, thus 10 cycles are needed to make the computation on 4 bytes.

FIGURE 9: The source codes with and without SIMD optimization.

TABLE 1: Cycle count for standard and optimized SAD algorithms.

	Standard SAD	Optimized SAD	Improvement
Execution cycles	1885	628	$1 - \left(\frac{628 \times 100}{1885} \right) \approx 67\%$

application implementation by adding very few new instructions to the instruction set of an existing architecture. This has some importance when the goal is to obtain better performances without choosing a more expensive technology or making a hardwired coprocessor that would perform orders of magnitude faster at a corresponding cost. We evaluated the feasibility of the improvement and the silicon area necessary for this approach.

An obvious limitation of our approach is the possibility to add only a small number of instructions. Our rule of using only the holes in the instruction set opcodes is however well suited for application specific instructions as opposed to domain specific extensions. The approach has been demonstrated on the opensource LEON-2 processor using the window search algorithm of the MPEG encoding algorithm. The results are very good, since for a marginal increase in area and clock cycle time, a 67% performance improvement has been measured.

However, the main contribution of this work is to show that instruction set extensions can be implemented with reasonable effort on a typical RISC processor.

REFERENCES

- [1] D. Bistry, C. Delong, M. Gutman, et al., *The Complete Guide to MMX Technology*, McGraw-Hill, New York, NY, USA, 1997.
- [2] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He, "VIS speeds new media processing," *IEEE Micro*, vol. 16, no. 4, pp. 10–20, 1996.
- [3] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*, pp. 124–135, Atlanta, Ga, USA, May 1999.
- [4] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 184–188, Las Vegas, Nev, USA, June 2001.
- [5] R. Gonzalez, "Configurable and extensible processors change system design," in *Proceedings of 11th Hot Chips Symposium*, Palo Alto, Calif, USA, August 1999.
- [6] "Tensilica: Configurable and Standard Processor Cores for SOC Design," <http://www.tensilica.com/>.
- [7] S. Aditya, B. Ramakrishna Rau, and V. Kathail, "Automatic architectural synthesis of VLIW and EPIC processors," in *Proceedings of the 12th International Symposium on System Synthesis*, pp. 107–113, San Jose, Calif, USA, November 1999.
- [8] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. Home-wood, "Lx: a technology platform for customizable VLIW embedded processing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 203–213, Vancouver, BC, Canada, June 2000.
- [9] J. Becker and A. Thomas, "Scalable processor instruction set extension," *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 136–148, 2005.
- [10] S. Tillich and J. Großschädl, "A simple architectural enhancement for fast and flexible elliptic curve cryptography over binary finite fields $GF(2^m)$," in *Proceedings of the 9th Asia-Pacific Conference on Advances in Computer Systems Architecture*, vol. 3189 of LNCS, pp. 282–295, Beijing, China, September 2004.
- [11] A. Artieri, "Nomadik: an MPSoC solution for advanced multimedia," in *Proceedings of the 5th International Forum on Application-Specific Multi-Processor SoC (MPSoC '05)*, Relais de Margaux, France, July 2005.
- [12] "openSPARC," <http://www.opensparc.net/>.
- [13] "opencores," <http://www.opencores.org/>.
- [14] "Gaisler Research," <http://www.gaisler.com/>.
- [15] "MicroBlaze Soft Processor Core," <http://www.xilinx.com/>.
- [16] "powerpc 405," <http://www.power.org/>.
- [17] S. Yang and I. Verbauwhede, "A realtime, memory efficient fingerprint verification system," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '04)*, vol. 5, pp. 189–192, Montreal, Que, Canada, May 2004.
- [18] T. Bautista and A. Nunez, "Flexible design of SPARC cores: a quantitative study," in *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES '99)*, pp. 43–47, Rome, Italy, May 1999.
- [19] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC V8 architecture," in *Proceedings of the International Conference on Dependable Systems and Networks (DNS '02)*, pp. 409–415, Washington, DC, USA, June 2002.
- [20] *The SPARC Architecture Manual*, version 8, <http://www.sparc.org/standards/V8.pdf>.
- [21] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pp. 145–156, Vancouver, BC, Canada, June 2000.
- [22] J. Sklansky, "Conditional-sum addition logic," *IRE Transactions on Electronic Computers*, vol. 9, no. 2, pp. 226–231, 1960.