The European Logarithmic Microprocessor

J. Nicholas Coleman, Chris I. Softley, *Member*, *IEEE*, Jiri Kadlec, Rudolf Matousek, Milan Tichy, *Member*, *IEEE*, Zdenek Pohl, Antonin Hermanek, *Member*, *IEEE*, and Nico F. Benschop

Abstract—In 2000 we described a proposal for a logarithmic arithmetic unit, which we suggested would offer a faster, more accurate alternative to floating-point procedures. Would it in fact do so, and could it feasibly be integrated into a microprocessor so that the intended benefits might be realized? Here, we describe the European Logarithmic Microprocessor, a device designed around that unit, and compare its performance with that of a commercial superscalar pipelined floating-point processor. We conclude that the experiment has been successful, and that for 32-bit work, logarithmic arithmetic may now be the technique of choice.

Index Terms—High-speed arithmetic, emerging technologies, instruction-set design, SIMD processors, design studies, logarithmic number system.

1 Introduction

A new microprocessor has been developed. Reliant for the execution of 32-bit real operations on a logarithmic arithmetic unit, such operations are performed substantially faster and also somewhat more accurately than with the floating-point (FLP) system. An operand is represented as its base-2 logarithm, itself a fixed-point value and so capable of multiplication, division, and square root in minimal time, and in the case of multiplication and division with no rounding error. Using an original approximation technique, addition and subtraction are carried out with speed and accuracy similar to that of FLP arithmetic. We described these techniques in [1], [2], to which reference should be made as the background to this work.

This device has been manufactured, integrated into a development system, and comprehensively evaluated. We envisage that applications for it will be in embedded systems of some numerical complexity, such as the more advanced digital filters and graphics systems. Inferences from [1] were that a significant reduction in cycle count may be expected in scalar codes, with about 6 dB better accuracy. However, a great many other factors will impinge in practice. Will the expected results be obtained in reality, and indeed will the new techniques work at all?

We begin, for completeness, with a short restatement of the techniques on which the device is based. It will be evident that a logarithmic ALU has little in common with an FLP one, and it is also true that conventional microprocessor architecture is adapted to deploy an FLP unit to best advantage. This raises a dilemma: does one simply take an FLP microprocessor and substitute a logarithmic ALU, forgoing some significant advantages that the logarithmic number system (LNS) may offer, or should one invest the substantial effort of reworking the microprocessor design accordingly? Among the few authors, including ourselves, who have considered this issue, the consensus is for the latter, and we next discuss the architectural issues relevant to the integration of the new ALU in the most favorable way.

With regard to accuracy, we had presented in [1] a series of simulations of the proposed ALU, in which we predicted its maximum and average error, and illustrated its typical performance on arithmetic kernels. We now revisit this experiment, this time running on the microprocessor itself, and show an almost identical set of results to that predicted.

Regarding speed, we had made crude predictions based on the cycle counts of the basic operations, but had not considered the many additional factors coming into play when the arithmetic unit is integrated into a processor. As we now have such a device, we are in a position to evaluate these practicalities, and for this purpose we chose an industry-standard superscalar pipelined FLP processor for comparison. Although the two devices had different arithmetic systems and supporting architectures, they were otherwise, so far as possible, similar. We present a simple analysis of the two devices to show that the predictions in [1] regarding scalar and peak vector performance are likely to hold in practice.

Theoretical processing rates, however, are an unreliable indicator of the behavior of a real device. We finish, therefore, with two case-study applications, both on a substantial scale and representative of real-world problems. Each application is programmed in optimized assembly language and run on the two processors. After measuring the speed and accuracy of each implementation, we analyze the dynamics of the architectural complex in order to isolate the contribution made by the different arithmetic systems to the results.

We show that the logarithmic arithmetic system used on the ELM has been instrumental in delivering an overall

- *J.N. Coleman is with the School of Electrical, Electronic and Computer Engineering, The University, Newcastle upon Tyne, NE1 7RU, UK. E-mail: j.n.coleman@ncl.ac.uk.*
- *C.I. Softley is with Photonfocus AG, Bahnhofplatz 10, CH-8853 Lachen, Switzerland. E-mail: softley@photonfocus.com.*
- *J. Kadlec, M. Tichy, Z. Pohl, and A. Hermanek are with the Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic, Pod Vodárenskou věží 4, PO Box 18, 182 08 Praha 8, Czech Republic. E-mail: {kadlec, tichy, xpohl, hermanek}@utia.caz.cz.*
- *R. Matousek is with Acision, Villa J, Spielberg Office Centre, Holandská 5, 639 00 Brno, Czech Republic. E-mail: rudolf.matousek@acision.com.*
- *N.F. Benschop was with Philips Research, Eindhoven, The Netherlands. He is now retired and resides at Drossaardstraat 71, 5663 GJ Geldrop, The Netherlands. E-mail: n.benschop@chello.nl.*

Manuscript received 13 Feb. 2007; accepted 13 June 2007; published online 2 Aug. 2007.
Recommended for acceptance by B. Veeravalli.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0058-0207.
Digital Object Identifier no. 10.1109/TC.2007.70791.

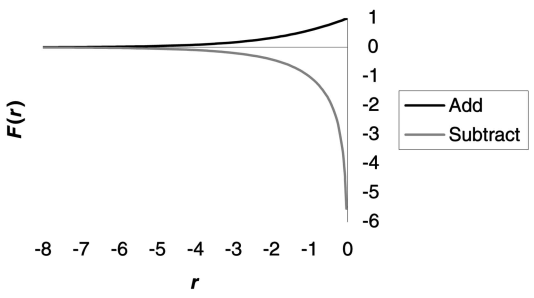


Fig. 1. LNS addition and subtraction functions.

increase in speed and improvement in accuracy. The higher speed has permitted a trade-off with reduced architectural complexity. The other relevant issues are silicon area and power dissipation. Although preliminary results suggest a broadly comparable area and a significant reduction in power, space does not permit an adequate treatment of these factors here, and in any case our results are not yet complete. We will therefore deal with these issues in a sequel to this paper. On the basis of the evidence currently presented, however, we conclude that, for 32-bit work, logarithmic arithmetic may now be the technique of choice.

2 LNS Arithmetic

In a logarithmic number system, a number x is represented as the fixed-point value $i = \log_2 x$, with a special arrangement to indicate zero x and an additional bit to show its sign. For $i = \log_2 x$ and $j = \log_2 y$, and assuming without loss of generality that, in dyadic operations, $j \leq i$, LNS arithmetic involves the following computations:

$$\begin{aligned}
&\log_2(x+y) = i + \log_2(1+2^{j-i}),\\
&\log_2(x-y) = i + \log_2(1-2^{j-i}),\\
&\log_2(x \times y) = i + j,\\
&\log_2(x \div y) = i - j,\\
&\log_2(\sqrt{x}) = i \div 2.
\end{aligned}$$

Although multiplication, division, and square root are straightforward, and implemented with simple modifications to the fixed-point unit, addition and subtraction require the evaluation of a nonlinear function, $F = \log_2(1 \pm 2^r)$, as illustrated in Fig. 1 for $r = j - i$.

Early proposals for short-word-length systems, for example, [3] at 20 bits, suggested implementation with a lookup table containing all possible values of F. Beyond about 20 bits, the exponentially increasing storage requirements render this approach impractical and the function is better stored at intervals through r, with intervening values obtained by some interpolation procedure. This introduces inaccuracy and increases the delay time. Subsequent work therefore focused on finding an acceptable compromise between the two. The first interpolators had order-of-magnitude differences between multiplication and addition times. A first-order Taylor method [4] was fabricated in 3μ CMOS [5]. At 28-bits plus sign and accuracy within FLP error bounds, multiplications completed with 55 ns latency and additions with 1.4 μs. A higher order polynomial method at 31-bits plus sign [6] also offered better accuracy than FLP. A lower accuracy variant of it, which did not, was fabricated in 1.2μ CMOS [7]. Multiplication latency here was 13 ns and addition 158 ns. Both designs exposed a difficulty in subtractions when $r \rightarrow 0$, where the rapidly changing derivative demands successively narrower intervals and a vast increase in storage. To circumvent this problem, we described a cotransform in [2]. Applied to r and i when $-0.5 < r < 0$, it returns $r2$ and $i2$, where $r2 < -1$, the new values $r2$ and $i2$ being passed to the interpolator. A further development of the interpolator itself was then described in [2], [1]. Its errors were also within FLP limits, and with a significantly shorter critical speed path than hitherto, it raised the prospect of 32-bit LNS addition better than FLP in terms of both speed and accuracy. This design, together with the cotransform unit, forms the basis of the implementation described herein and is summarized in Section 2.1. Finally, ongoing work focuses not only on faster interpolation but also, for example, [13], on longer word lengths.

2.1 Addition and Subtraction Algorithms

For each function, the range of $r = j - i$ is partitioned into segments at increasing powers of 2. Each segment is divided into intervals, at each of which the function F and its derivative D are stored. By partitioning r, a linear Taylor interpolation produces an estimate of the function at intermediate points. The error in the estimation increases from zero when the required value lies on a stored point, to E when it falls immediately to the side of the next stored point. It was observed that the shape of this error curve is very similar in all intervals in both curves. It is therefore possible to calculate the error in any particular case by storing, alongside F and D for each interval, its value of E. A separate table stores the normalized shape of the common error curve, from 0 when the required value lies on an interpolation point, to 1. This table is known as the P (proportion) function. The error is calculated by multiplying E by P and is then added to the result of the interpolation. A full explanation is given in [1], the only modification to which was that the truncation of the interpolator multipliers was determined empirically, by successively trimming low-order bits from the multiplier in the hardware description model and simulating with random values while keeping within FLP error bounds.

The range shifter is inserted immediately after the selection of i and calculation of r. As described above, it is deployed selectively whenever r falls close to zero in the subtraction operation, transforming i and r into new values, where r is now in the linear region. Space does not permit an explanation of its operation here; it is fully explained in [2].

The entire unit is depicted in Fig. 2. Its accuracy was verified by exhaustive simulation of the hardware description model before fabrication, as described in Section 4.

2.2 Data Format, Range, and Precision

IEEE standard FLP representation uses a sign, an 8-bit biased exponent, and a 23-bit significand. The latter has an implied binary point immediately to its left and a hidden "1" to the left of the point. Extreme exponent values (0 and 255) are used for representing special cases; thus this format holds values in the range $\pm(1.0 \times 2^{-126})$ to $(1.111\ldots \times 2^{+127})$, $\approx \pm 1.2\text{E}-38$ to $3.4\text{E}+38$.

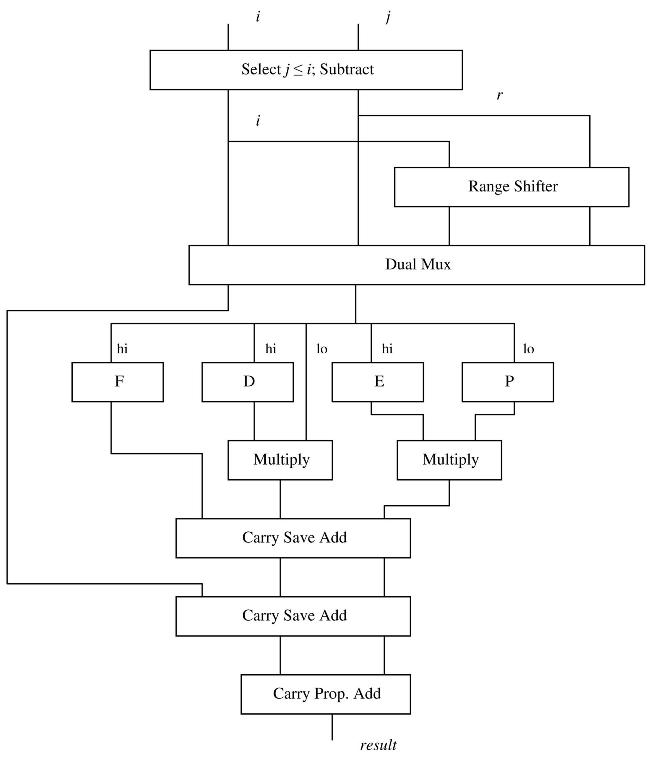


Fig. 2. Addition and subtraction hardware.

S	8-b Exponent	23-b Mantissa

In the equivalent LNS representation used throughout this work, the integer and fractional parts form a coherent two's complement fixed-point value in the range ≈ -128 to $+128$. The real numbers represented are in the range $\pm 2^{-128}$ to 2^{+128}, $\approx \pm 2.9\mathrm{E}-39$ to $3.4\mathrm{E}+38$. The smallest positive value, 40000000_{16}, is used for representing zero, while $\mathrm{C0000000}_{16}$ represents NaN.

S	8-b Integer	23-b Fraction

Underflows are taken to zero. Overflows, invalid operands, and any operation with NaN as an input return NaN.

3 Processor Philosophy and Architecture

As far as we were aware, there were no LNS-based microprocessor devices yet in existence. The motivation for this work was to develop one. However, a few authors had noted, as we did, that LNS ALUs have markedly different characteristics from their FLP equivalents and are therefore likely to require some reevaluation of the surrounding microprocessor design to deploy them to best advantage. Paliouras et al. [8] used the work in [6] as the basis for the addition unit in a proposed VLIW device, optimized for filtering and comprised of two independent ALUs, each with a 4-stage pipelined adder and a single-cycle multiplier. Simulations suggested an operating frequency of 12 MHz in 0.7μ CMOS. In an outline proposal for the ELM [9], we noted that the small size of the LNS multiplier-cum-integer unit encouraged its replication, leading to a short vector design with four single-cycle multipliers and two multicycle flowthrough adders. Arnold [10] instead proposed a VLIW device with only one single-cycle multiplier and one 4-stage pipelined adder, suggesting that large-scale replication of functional units could lead to difficulties with the complexity of the multiplexing paths leading to and from the registers. He then considered a number of issues related to suitable instruction set architectures (ISAs), but did not present a finished design or simulation results.

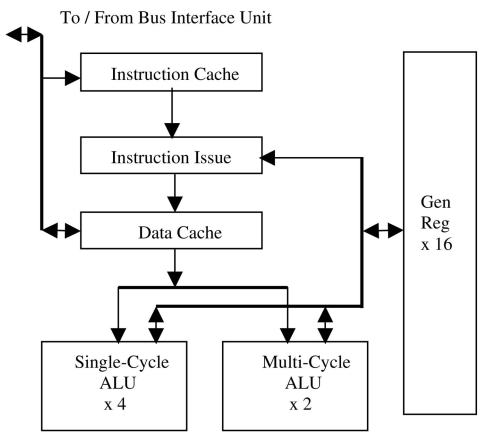


Fig. 3. ELM pipeline.

The following discussion is based on the characteristics of the LNS arithmetic unit at our disposal, and we show how these have influenced the design decisions relating to its integration into the ELM pipeline. Depicted in Fig. 3, this is based around 16 general-purpose registers.

Simulations of the hardware model had revealed that the basic addition/subtraction unit would incur about three times the delay of a fixed-point addition and that the range shifter, when invoked, would delay it by a further cycle. The first decision concerned the desirability of pipelining the arithmetic units. A number of factors seemed to weigh against:

- Whereas an FLP addition or multiplication circuit is complex enough that it is usually economic to emphasize its throughput by pipelining it, such an argument does not apply to fixed-point units and therefore not to the LNS multiplier. This leaves the LNS addition/subtraction unit as the only block of circuitry with a delay greater than a cycle —usually three and occasionally four—and it does not seem worthwhile to devise a pipelining scheme for this operation alone.
- One of the principal advantages of the LNS is the reduced latency of the multiplication operation. This implies that the device will be of particular utility in latency-sensitive applications, in which case it would seem consistent also to minimize the latency of the addition operation, but pipelining would inevitably increase it.
- The addition/subtraction unit has a variable delay time, which would complicate a pipeline control algorithm.

Accordingly, we have arranged a fully interlocked pipeline in which a single-cycle ALU executes all operations except logarithmic addition and subtraction. These are handled by the multicycle ALU, which executes with a three-cycle flowthrough time. The instruction issue unit continues to launch two more instructions while waiting for the addition to complete, provided that these are destined for the single-cycle ALU and do not otherwise conflict with the addition or subtraction in progress. This raises the possibility that both ALUs may complete at the same time, and the register file update paths are designed with sufficient bandwidth to accommodate this.

The second decision concerned the ISA. Mainstream computer architecture favors a register-register approach which decouples the data cache from the pipeline via a load/store unit in order to permit its independent concurrent operation. This is often combined with superscalar instruction processing in order that the loads can be issued in parallel with the arithmetic operations. Review of the likely application areas for this device, and the practicalities of designing it, suggested that this might not be the best approach and that a dual RR/RM architecture with a cache incorporated into the pipeline might be considered instead. Pertinent factors were:

- Time did not permit any consideration of superscalar design, so the device is limited to the issue of one instruction per cycle. Explicit load instructions would use up an issue slot.
- Complex numeric codes contain a high proportion of loads and stores as loaded data often have a short working life. Explicit loading would unavoidably increase the number of cycles per executed real instruction.
- A decoupled cache would inevitably have a longer access latency than a tightly integrated one, which would be inconsistent with the low latency emphasized elsewhere.

The drawback would be that an integrated cache could not operate independently of the pipeline. Potentially this could reduce performance in two ways:

- "Intelligent" features such as early data forwarding or predictive loading would be impossible. It would have to be accepted that, if required data were not in the cache, then the system would stall while the entire line was loaded. However, this could be minimized by making the cache as large as possible, with a relatively small line size.
- Disparities between the speed of the cache and that of the pipeline could not be accommodated so easily. However, since the LNS adder/subtractor uses several ROM devices itself, the pipeline speed will always be governed by memory speeds, so large disparities are unlikely to arise.

Consequently, the ISA allows for RR and RM addressing. Superficially similar to that of the IBM S/360, operands located in memory are addressed by a base register plus a 16-bit unsigned displacement. All instruction codes are 32 bits, and in total there are 35 distinct operations.

The final design question was more clear-cut. So far, we have emphasized low latency, sometimes at the expense of throughput. Functional unit replication is commonly employed to boost throughput on FLP devices. Unlike functional unit pipelining, this is a very appropriate strategy on the ELM, where the single-cycle unit is so small that it can easily be replicated four times. A vector capability is thereby provided, permitting the execution of four integer, logical, or LNS multiplication, division, or square root operations in one clock cycle. The vast majority of the silicon area used for the arithmetic circuitry can thus be reserved for the more substantial multicycle ALU. Two such units are provided, allowing two additions or subtractions to proceed in three cycles.

Accordingly, all instructions are available in either scalar or vector form, the former operating on only one memory location or register, and the latter on a set of four consecutive locations or registers (or two in the case of addition and subtraction). A hybrid mode is available to apply a scalar to all elements of a vector. The orthogonal ISA permits almost all meaningful combinations of scalarity and addressing mode.

Scalar integer and logical operations set a 2-bit condition code, according to the result being < 0, $= 0$, or > 0. Branch instructions may be conditional on any combination of the three. Because of the ambiguity, when a logarithmic and an integer instruction complete at the same time, logarithmic instructions do not set the condition code. Conditional tests on logarithmic values are instead made by applying an appropriate logical operation. For example, a register would be reloaded from itself to determine the state of the sign bit, which has the same significance in logarithmic values as in integers. A useful by-product of the data representation is that a one-place logical left shift brings the logarithmic part of a value into the same significance as that of an integer, transforming a sign comparison around zero into a magnitude comparison around unity. This might be used, for example, following a magnitude comparison between arbitrary values, which in the LNS would be best accomplished by division.

To maintain the required memory bandwidth, the instruction and data caches supply, respectively, one and four 32-bit words to the processor per cycle. Each cache is of 8-Kbyte capacity and is two-way set associative (until a late stage in the design they were to have been 16 Kbytes, which was within the timing tolerance but slightly over budget). A 64-bit asynchronous external bus interface is provided, with a minimum transaction time of three cycles per read and four per write, although these times can be increased with wait states if desired. Input and output devices may be mapped into the memory address space, and transfer 32 bits per operation. I/O devices may be accessed by polling, but also have recourse to a single-level nonmaskable interrupt.

Following verification on an FPGA, devices were fabricated in 0.18μ CMOS and packaged in 181-pin grid arrays for easy handling during development work. Sign-off simulations immediately before fabrication had indicated an operating frequency of 166 MHz. Fabricated devices ran on the tester at 150 MHz. The current devices have been installed on development boards and are functional at 125 MHz, although a batch of higher quality boards is ready for manufacture and it is expected that these will permit some increase in operating speed.

The existing development systems consist of a 7×8 in board containing a 125-MHz ELM1A, 4 Mbytes of 10-ns static RAM, basic peripherals, and serial interface to a host PC. The bus speed, both for read and write accesses, is set to four cycles. Software support is currently comprised of a relocating assembler, a linkage editor, and a mathematical

TABLE 1
Errors in ELM Addition and Subtraction Operations

	$\lvert e\rvert_{\text{max rel log}}$	$\lvert e\rvert_{\text{av rel log}}$	$e_{\text{max rel arith}}$	$e_{\text{min rel arith}}$	$e_{\text{av rel arith}}$	$\lvert e\rvert_{\text{av rel arith}}$
Add	0.61639037	0.24831750	0.37871610	–0.42724924	–0.010716149	0.17212057
Subtract	0.68221501	0.24811711	0.42695100	–0.47287539	–0.0051578344	0.17198167

link library. An implementation of one of the public versions of ANSI-C is in an advanced state of preparation.

4 Accuracy

In IEEE 754 arithmetic using the round-to-nearest option, a result is guaranteed to lie on the closest available quantization point. This is sometimes called "exact" rounding, and is difficult to achieve in the LNS. The logarithm itself might be rounded to the nearest point, but because of the nonlinear relationship between i and $x = 2^i$, this does not guarantee that x will also be exactly rounded when i lies near the midpoint of an interval. On the other hand, the LNS has the felicitous property of a significantly lower relative error than the maximum of that of FLP. Even with inexactly rounded i, this more than compensates for the inexact rounding of x, and consequently an LNS implementation will tend to have smaller worst-case relative errors than those of FLP. A detailed working out of this principle is presented by Arnold and Walter [11], who refer to this property of the LNS as "better than FLP" accuracy. The object of the experiments described herein is to determine whether this has happened in practice.

We evaluated the ELM addition and subtraction units in two ways. An exhaustive analysis determined the definitive error characteristics, and stochastic experiments illustrated their typical operational behavior. The units used for the measurement of accuracy are as defined in [1].

4.1 Definitive Analysis

ELM assembly language programs were developed to sweep through all representable values of the log. domain difference of the two operands to the logarithmic addition or subtraction, as far as a difference of 24×2^{23}. This corresponds to sweeping through the entire range of r up to the point at which the value of the relevant function is set to hardwired zero. In other words, the function approximation hardware in the logarithmic addition and subtraction unit has been tested for all representable values of its operand r.

These tests were carried out during the design verification stage prior to layout and fabrication. An FPGA prototype implementation of the ELM, logically identical to the ELM itself in all respects except cache size, was employed to speed up the test, and the test was split into 768 sections so that the data generated for each section would be small enough to fit into the memory available on the FPGA development board. A Perl script automated the generation and retrieval of a complete set of results for all 768 sections for both addition and subtraction.

Analysis of the results was automated by a Perl script, which read in the result saved from the FPGA run and evaluated the corresponding exact result using IEEE double-precision FLP. It then calculated the error in the ELM implementation and produced the maximum, minimum, mean, and mean-size figures for each section and for the entire data set, in terms of both log. domain LSBs and equivalent FLP LSBs. The script also binned the errors to form error histograms both for every section and for the entire data set.

The error figures for the entire range of r for LNS addition and subtraction on the ELM are given in Table 1.

The error distributions are shown in Fig. 4. Since the largest errors found for any value of r for both addition and subtraction were under 0.5 equivalent FLP bits, we conclude that the ELM implementation of LNS satisfies the required bounds for "better than FLP" arithmetic.

The final hardware implementation of the algorithms differs slightly from that already published [1, Table 1] and these results show its error performance to be marginally better. The only previous work to show the error distributions for LNS was Lewis' interleaved memory implementation of 32-bit LNS with a weak error model for subtraction of nearly equal quantities [6]. The distributions for normal operation of that unit agree in shape with the distributions presented here, except, of course, for the region in which the weak error model comes into play. No discussion of these distributions is given, however, and this is a possible area for further work.

4.2 Typical Behavior

In [1], we described an experiment in which we compared the accuracy of 32-bit FLP operations and kernels with that of their simulated LNS equivalents. Reference should be made to this work for a description of the method, and to [1, Fig. 8] for the results. The average processing error of the simulated LNS implementation had been compared with that of FLP, and the experiments were repeated several

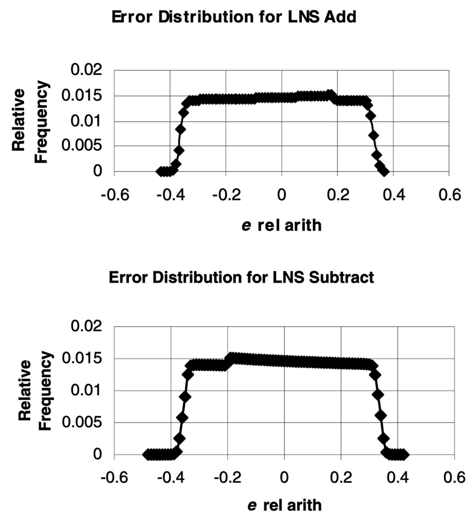


Fig. 4. Error distribution in ELM LNS addition and subtraction.

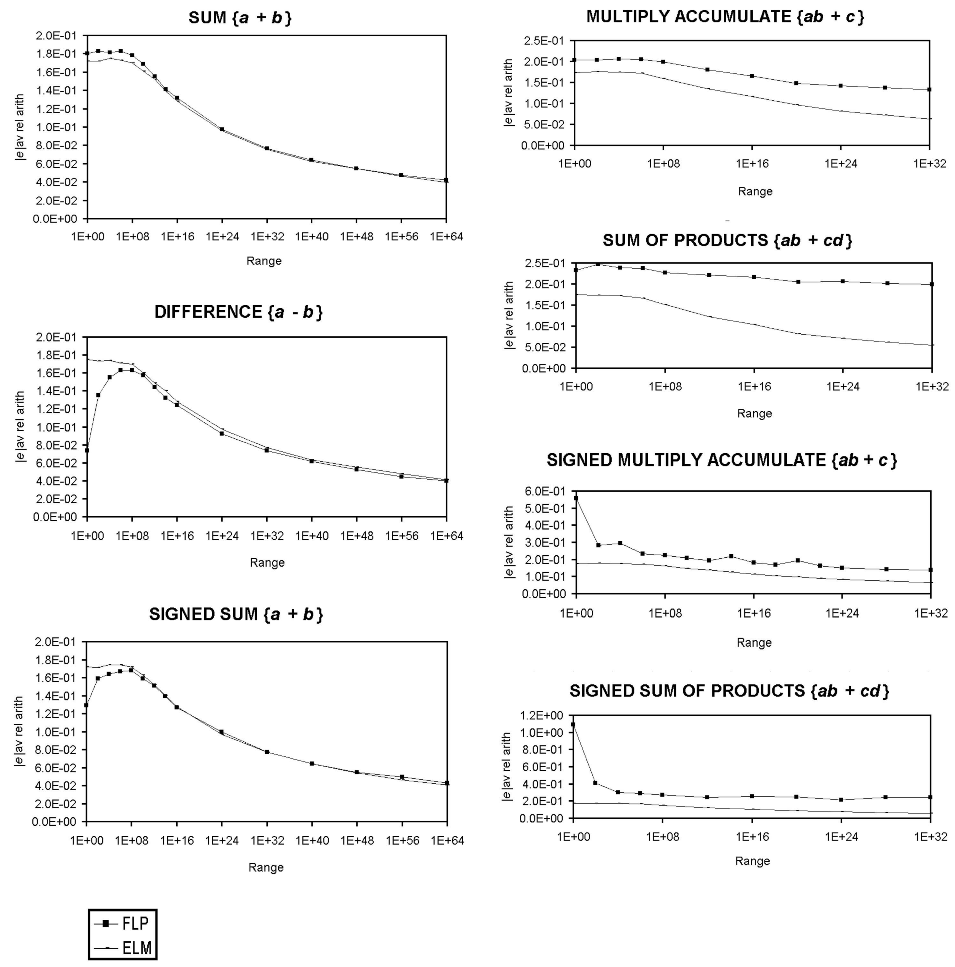


Fig. 5. Error in ELM execution of arithmetic kernels.

times for each kernel, with gradually increasing dynamic range.

We now repeated these experiments, this time running the LNS program on the ELM evaluation board rather than in simulation. The procedure was otherwise identical, although we had upgraded the Pascal compiler in the interim and were therefore reliant on a different randomization intrinsic and thus on different input data. We show these new results in Fig. 5. The results obtained from the ELM are virtually identical to the simulated LNS predictions. The FLP results are also similar, except that extremes, some of them anomalous, in the two signed multiplicative kernels seem to have varied. These presumably originate in some interference pattern between FLP methods and imperfections in the randomizers.

Overall, the LNS errors are generally less than those of FLP and are smoother throughout the dynamic range. They are less sensitive to the actual data values, although there might be some question as to whether this would be the case had the randomizer been implemented in LNS arithmetic.

5 Speed

The ELM is fabricated in 0.18μ technology, and all of the measurements reported in this work are taken from the existing 125-MHz systems. In order to discuss the speed, it is helpful to have an FLP standard for comparison. For this purpose, we looked for an industry-standard FLP DSP device, as similar as possible to the ELM in every respect apart from its arithmetic system. We chose the Texas Instruments TMS320C6711. Apart from DSP applications, this device is also used in graphics. It runs at 150 MHz, the fastest speed grade available in 0.18μ technology. Although a faster speed grade, the C6711D, has recently become available, it is fabricated at 0.13μ with copper interconnect and is therefore not directly comparable with the current implementation of the ELM. A variant, the C6701, is currently manufactured at 0.18μ and runs at 167 MHz, but is also not comparable because it has a simplified on-chip memory, having dispensed with the cache in favor of a RAM.

The TMS is a superscalar device with a VLIW arrangement and an RR instruction set. It has two independent arithmetic units, A and B, each with 16 registers $0..15$, parallel pipelined FLP add and multiply units, and integer and reciprocal-approximate units. Up to eight independent instructions may be issued per cycle (the symbol "||" in the assembly listings shown later designates that an instruction is issued in the same packet as the previous one). The add and multiply units have a latency of four cycles, the integer unit one. Divide and square root are implemented in software by using the Newton-Raphson method. Optimized assembly-language routines are provided by the manufacturer for this purpose and require about 30 and 40 cycles,

TABLE 2
Hardware Resources of TMS and ELM Devices

	TMS	ELM
Fabrication technology	0.18μ, 5-layer metal	0.18μ, 6-layer metal
Clock speed	150 MHz	125 MHz
Real add / subtract units	2	2
Real multiply units	2	4
Real divide / sq-root units	2 (reciprocal / recip. sq-root approx.)	
Integer / logical units	2	
Registers	32	16
Level-1 data cache size	4 Kbytes	8 Kbytes
Level-1 data cache mapping	2-way set associative	2-way set associative
Level-1 data cache policy	Read-allocate	Write-allocate, writeback
Level-1 data cache organisation	32 bytes x 2 ways x 64 lines, (dual ported)	32 bytes x 2 ways x 128 lines
Level-1 data cache read bandwidth	2.4 Gbytes / sec	2 Gbytes / sec
Level-1 data cache write bandwidth	1.2 Gbytes / sec	1 Gbyte / sec
Level-1 instruction cache size	4 Kbytes	8 Kbytes
Level-1 instr. cache mapping	Direct	2-way set associative
Level-1 instr. cache organisation	64 bytes x 64 lines	32 bytes x 2 ways x 128 lines
Level-2 cache size	64 Kbytes	-
Level-2 cache mapping	4-way set associative	-
Level-2 cache policy	Write-allocate, writeback	-
Level-2 cache organisation	128 bytes x 4 ways x 128 lines	-
External bus width	32 bits	64 bits
External bus protocol	Synchronous	Asynchronous
External bus bandwidth	340 Mbytes / sec	250 Mbytes / sec

respectively. Loads and stores enter the memory pipeline, which has a four-cycle latency. There is a 4-Kbyte level-1 (L1) data cache, 4-Kbyte L1 instruction cache, and a 64-Kbyte unified level-2 (L2) cache. The device has a 32-bit external bus with a configurable protocol. On the evaluation board used for these experiments, it used synchronous dynamic RAMs (SDRAMs), and was running at its maximum bandwidth of about 340 Mbytes/sec.

The ELM is a scalar device with an RM instruction set and 16 general registers. The device has a 64-bit asynchronous bus with a bandwidth on the current boards of 250 Mbytes/sec. The resources available to each device are detailed in Table 2.

Evidently the devices are comparable in terms of fabrication technology and clock speed. Each device has six principal ALUs. The TMS is comprised of two adders, two multipliers, and two integer units (the adders and multipliers can also be used for fixed-point work), while the ELM has two adders and four combined multiplier/integer units. The devices are also broadly comparable in terms of memory bandwidth, but do have different cache arrangements. The TMS has a unified L2 cache, which is not present on the ELM. There are also differences in the L1 caches, which to some extent are consequent on the different arithmetic techniques. The straightforward implementation of vector operations on the ELM, putting four identical functional units in parallel, requires a simple data cache organization in which four consecutive words are read out simultaneously. The TMS, with pipelined functional units launching operations in different cycles, is more likely to encounter bank conflicts and is therefore more likely to benefit from dual porting to the cache.

The TMS has an optimizing C compiler. It is not straightforward to program in assembly language because it does not have an interlocked pipeline, that is, there is no hardware mechanism for checking that one instruction has completed before a dependent one begins. The programmer must therefore construct a dependency chart from which the parallel operations are scheduled manually, calculating the state of the pipeline at each clock cycle and if necessary inserting no-ops to account for instruction latency. In vector operations, there might be several iterations of the loop in different stages of the pipeline simultaneously. The pipeline is brought into the correct state by a prologue before the start of the iterating kernel, and the terminating iterations are wound down by an epilogue at the end. To simplify programming, the manufacturer provides a "linear assembler," which determines the dependencies, schedule, and register allocation automatically. The preferred method of programming is then to use the compiler for the outer scalar sections of code, which can easily be scheduled efficiently by the compiler, and to write the inner loops in linear assembly language.

The ELM has an interlocked pipeline and also an RM ISA. The two features together make for a relatively straightforward assembly language interface, which is currently the only method of programming the device.

Latencies and vector throughput on the two devices are summarized in Table 3. Those for the discrete operations are shown first, followed by the values for multiply-accumulate (MAC) and sum-of-products (SOP) kernels. It is assumed in the two kernels that input data are taken from memory and results are returned there. The maximum vector execution rate on the TMS is two additions and two multiplications per

TABLE 3
Latency and Throughput of TMS and ELM Devices

Operation	TMS				ELM			
	Latency (cycles)	Latency (ns @ 150MHz)	Scalar Throughput (per cycle)	Vector Throughput (per cycle)	Latency (cycles)	Latency (ns @ 125MHz)	Scalar Throughput (per cycle)	Vector Throughput (per cycle)
Add	4	27	0.25	2	3	24	0.33	0.67
Subtract	4	27	0.25	2	3 (4)	24 (32)	0.33 (0.25)	0.67
Multiply	4	27	0.25	2	1	8	1	4
Divide	≈30	≈200	≈0.033		1	8	1	4
Square-root	≈40	≈267	≈0.025		1	8	1	4
Load	4	27	0.25	4	1	8	1	4
Store	4	27	0.25	2	2	16	0.5	2
MAC	16		0.0625	0.67	7		0.14	0.5
SOP	16		0.0625	0.67	7		0.14	0.5

A
```
LV      0,A0    * MAC, X = AB + C
MLV     0,B0
ALV     0,C0
ALV     2,C2
LV      4,A4
MLV     4,B4
STV     0,X0    * STORE 0, CYCLE 9
ALV     4,C4
ALV     6,C6
LV      0,A8
MLV     0,B8
STV     4,X4    * STORE 4, CYCLE 17
ALV     0,C8
ALV     2,C10
LV      4,A12
MLV     4,B12
STV     0,X8    * STORE 8, CYCLE 25
```

B
```
LV      0,A0    * SOP, X = AB + CD
MLV     0,B0
LV      4,C0
MLV     4,D0
ALRV    0,4
LV      8,A4
MLV     8,B4
ALRV    2,6
LV      4,C4
MLV     4,D4
STV     0,X0    * STORE 0, CYCLE 11
ALRV    8,4
LV      0,A8
MLV     0,B8
ALRV    10,6
LV      4,C8
MLV     4,D8
STV     8,X4    * STORE 4, CYCLE 19
ALRV    0,4
LV      8,A12
MLV     8,B12
ALRV    2,6
LV      4,C12
MLV     4,D12
STV     0,X8    * STORE 8, CYCLE 27
```

Fig. 6. ELM assembly language implementations of vector kernels. (A) MAC and (B) SOP.

cycle. The cache-processor bandwidth permits four loads or two stores per cycle. Its peak performance is therefore 0.67 MAC or SOP operations per cycle. These figures will only be achieved under best-case conditions with relatively long vectors, where the pipeline can be kept saturated with sequential elements. In purely scalar sequences, the loads, multiplications, addition, and store would execute in 16 cycles, a throughput of 0.0625 MAC or SOP operations per cycle.

On the ELM, it is possible to launch two logarithmic additions in one cycle and then to process four loads and four multiplications in the remaining two cycles before the addition completes. The four multiplications perform implicit loads, so all eight words are loaded in the two cycles. The maximum execution rate on this device is thus two additions, four multiplications, and eight loads in three cycles. The best schedule, again working from and to memory, is illustrated in Fig. 6, which also serves as an example of programming. The MAC calculates $x[n] = a[n]b[n] + c[n]$. It starts with the instruction "Load vector," which loads $a[0]..a[3]$ into $\mathrm{R0..R3}$. "Multiply logarithm vector" multiplies each register by the corresponding element of b. Each instruction takes one clock cycle. "Add logarithm vector" then adds $c[0]..c[1]$ into $\mathrm{R0..R1}$ in three cycles. Another ALV accumulates $c[2]..c[3]$ into $\mathrm{R2..R3}$. This also requires three cycles, but during the last two, the products of the next four elements of a and b are prepared in $\mathrm{R4..R7}$. With the addition now complete, the result is returned to $x[0]..x[3]$ via "Store vector," which takes two cycles. It is evident in Fig. 6A that the sequence can repeat every eight cycles, giving a peak vector rate of 0.5 MAC per cycle. Fig. 6B illustrates the SOP kernel. Here, the ab product is generated in $\mathrm{R0..R3}$ and cd in $\mathrm{R4..R7}$, the two products being added by "Add logarithm register vector." The second addition adds $\mathrm{R6..R7}$ to $\mathrm{R2..R3}$. Immediately after this starts, the former registers are finished with and $\mathrm{R4..R7}$ are reused for the preparation of the next product. Again, the sequence maintains 0.5 SOP per cycle. Neither kernel makes any use of $\mathrm{R12..R15}$, which are available for other purposes, for example, holding the base address of the vectors. The scalar MAC or SOP would execute in seven cycles, yielding a throughput of 0.14 per cycle.

Measuring by time, and recognizing the slightly slower clock speed of the current implementation of the ELM, additions and subtractions are still marginally faster than on the TMS, except when the range shifter is deployed. Allowing that this might occur in half of the subtractions, and that additions and subtractions might occur equally, the average speed of the ELM on these operations improves on that of the TMS by a few percent. Multiplications are

3.4 times the speed, and divisions and square roots complete in a small fraction of the time. It might be argued that additions and subtractions on the TMS are subject to the delay times of three more pipeline registers than on the ELM, so a strict calculation of flowthrough times would improve the TMS values. Against this, however, the entire design philosophy of the TMS device is based on pipelining. Its behavior would otherwise be radically different, and we must evaluate the device as it is.

Evidently the TMS is at an advantage in vector sequences. However, it will only attain this level of performance when it is possible to maintain a continuous pipeline flow. This will be difficult in code with data dependencies or with short vectors for which the start-up times dominate. In longer vector sequences, the TMS may exceed the performance of the ELM, particularly where fewer memory accesses are required, such as in running MAC sequences or weighted vector sums. However, in very long vector processing, other factors may come to dominate. In particular, it may become difficult to maintain the required memory bandwidth when the cache capacity is exceeded. At this point, the performance of the two devices is likely to converge as both become dominated by their external memory bandwidths, which are similar.

The ELM has a lower latency than the TMS in almost all cases, and hence a significantly higher scalar throughput. The figures suggest that, as predicted in [1], the ELM can be expected to offer around twice the performance of the TMS in scalar code and in less regular or short vector sequences. In code involving a significant proportion of division or square root operations, the ELM may exceed this.

6 Case Studies

Processor architecture, instruction scheduling, compiler efficiency, caching, memory access patterns, and other related factors all affect final performance. Two large-scale case studies were undertaken to determine whether the theoretical predictions hold in practice.

Material was drawn from two application areas: DSP and graphics. The first had a substantial proportion of vector code, the second far less, thus playing to the respective strengths of both processors. In both applications, all but a few percent of the processing workload consisted of relatively small kernels, so compiler effects were eliminated by writing these in assembly language. On the TMS we used the linear assembler, which was given every facility to optimize for speed, including speculative processing beyond array bounds and freedom from aliasing constraints. In most cases this produced an obviously near-optimal schedule, but in the few instances where this was not the case, the results were improved by assigning registers manually. On the ELM, we wrote standard assembly language and then optimized it manually by moving single-cycle instructions wherever possible to lie in the shadow of a preceding multicycle one. The outer mainlines were written in compiled C on the TMS, with options set to emphasize speed, and linked to the kernels. On the ELM, the mainlines were written in assembler, but were not subsequently optimized.

How efficient was the resulting TMS assembly programming? By considering the number of operations, the critical speed path between them, and the available functional units, we calculated the minimum possible latency for the real arithmetic processing in each of the scalar loop kernels. We show this as L_A. It is not realistically possible to avoid conditional branching latency in scalar codes, and taking this into account, a revised figure L_{AB} is also shown. This represents the minimum number of clock cycles in which the loop could possibly execute, assuming an unlimited supply of registers, perfect scheduling, loads, stores, and all housekeeping operations lying within the latency of arithmetic operations, and no overheads accruing from any other source such as from calling conventions. The latency achieved by the C compiler with the original high-level code is then given as L_C. Finally the latency actually achieved by the scheduled assembly program is shown as L. The ratios L_C/L_{AB} and L/L_{AB} therefore represent the quality of the result; the closer these values are to unity, the better. To avoid distortion, the latency of called arithmetic functions is shown separately, for example, as (+ div). For vector loops, the assembler will devise a schedule in which arithmetic operations and branches are both pipelined. The best possible schedule in this case, designated L_{ABV}, is determined simply by dividing whichever operation occurs most frequently by the number of functional units available to service it. For example, a loop containing six additions would require a minimum of three cycles per iteration with two pipelined addition units. As before, L/L_{ABV} grades the result.

For each application, we first describe the algorithm, mentioning any factors likely to influence execution speed. Excerpts of code are included where they serve to explain any significant differences in performance. Examples of input and output data are given. With regard to output, we have only shown that produced by the ELM, but where there are significant differences between it and that of the TMS, we have performed an analysis of accuracy against an 80-bit standard by using the same procedure as in Section 4. We measure the time taken by each processor, and by working through the assembly listings, counting the number of clock cycles for each kernel and multiplying by the number of passes through it, we analyze the number of clock cycles spent in each. The difference between the expected total clock cycles and the observed runtime is accounted for by stalling. This is predominantly because of cache activity, but also includes delays arising on the TMS from bank collisions and on the ELM from the subtractor range shifter (and to a negligible extent, it also includes the mainline processing). Finally, we discuss the results and show how the different arithmetic implementations have accounted for any differences in performance.

6.1 A Vector Application: Recursive Least Squares QR Filtering

This type of rapidly converging adaptive filter would be used at higher orders, such as $\mathrm{N} = 64$, in precision control systems, such as that of a DVD player. At lower orders, it could be used in applications such as voice compression for Internet transmission. A preliminary version of this experiment and the algorithm used was described in [12].

The kernel is programmed as shown in Fig. 7A. The i loop processes the boundary cell to the left of each row and consists of a chain of serial dependencies, including a square root and a division. The inner j loop processes the remainder of the row and may be vectorized. Programming of the latter is illustrated in Figs. 7B and 7C. The calculated efficiency of the TMS version is shown in Table 4.

A

```
for (i = 0; i<= n; i++) {
  temp = sqrt (u [i] [i] * u [i] [i]
    + r [i][i] * r [i] [i]);
  den = 1 / temp;
  sfi = u [i] [i] * den;
  cfi = r [i] [i] * den;
  r [i] [i] = temp;
  z [i] = z [i] * cfi + d [i] * sfi;
  d [i + 1] = -z [i] * sfi + d [i] * cfi;
  p [i + 1] = p [i] * cfi;
  if (i != n) {
    for (j = i + 1; j <= n; j++) {
      r [i] [j] = r [i] [j] * cfi
        + u [i] [j] * sfi;
      u [i + 1] [j] = -r [i] [j] * sfi
        + u [i] [j] * cfi;
    }
  }
}
```

B

```
L2:     ; PIPED LOOP PROLOG
           LDW      .D2T2    *B5++,B7
           LDW      .D1T1    *A4++,A3
           MPYSP    .M2      B7,B4,B7
           MPYSP    .M1      A0,A6,A0
||         LDW      .D2T2    *B5++,B8
           LDW      .D1T1    *A4++,A0
||         MPYSP    .M2X     A0,B4,B7
           MPYSP    .M2      B7,B4,B8
           MPYSP    .M1      A3,A6,A3
||         LDW      .D2T2    *B5++,B7
           MPYSP    .M2X     A3,B4,B9
||         LDW      .D1T1    *A4++,A0
||         ADDSP    .L1X     B7,A0,A0
           MPYSP    .M2      B8,B4,B7
||         B        .S1      jloop
           SUB      .L2X     B12,A13,B0
||         SUB      .L1X     B12,A13,A1
||         MPYSP    .M1      A0,A6,A3
||         LDW      .D2T2    *B5++,B7
           MVK      .S1      0x3,A2
||         MV       .D2      B7,B8
||         MPYSP    .M2X     A0,B4,B
||         LDW      .D1T1    *A4++,A0
||         ADDSP    .L1X     B8,A3,A0
jloop:     ; PIPED LOOP KERNEL
   [ A2]   SUB      .L1      A2,1,A2
|| [ B0]   B        .S1      jloop
|| [ A1]   STW      .D1T1    A0,*A5++
||         MPYSP    .M1      A0,A6,A
||         MPYSP    .M2      B7,B4,B
           SUBSP    .L2X     B1,A3,B8
||         MV       .S2      B8,B1
||         MPYSP    .M1      A0,A6,A3
||         LDW      .D2T2    *B5++,B7
   [ A1]   SUB      .S1      A1,1,A1
|| [!A2]   STW      .D2T2    B8,*B6++
|| [ B0]   ADDK     .S2      0xffffffff,
||         MV       .L2      B9,B8
||         ADDSP    .L1X     B7,A3,A0
||         MPYSP    .M2X     A0,B4,B9
||         LDW      .D1T1    *A4++,A0
L4:     ; PIPED LOOP EPILOG
end:
           ADDK     .S1      0x1,A13
L6:
           CMPGT    .L1X     A13,B12,A1
   [!A1]   B        .S1      iloop
           MV       .D2      B12,B6
           ADDK     .S2      0x1,B6
           SHL      .S2      B6,0x2,B6
           ADDK     .S2      0x4,B6
||         ADDK     .S1      0x4,A11
           ADD      .L1X     B6,A10,A10
||         ADD      .L2      B6,B10,B10
||         ADDK     .S1      0x4,A12
||         ADDK     .S2      0x4,B11
|| [ A1]   LDW      .D2T2    *+SP(24),B3
           ; BRANCH OCCURS
```

C

```
        LV      0,R(14)
        MLSV    0,CFI
        LV      4,U(14)
        MLSV    4,SFI
JLOOP   ALRV    0,4
        LV      8,U(14)
        MLSV    8,CFI
        ALRV    2,6
        STV     0,R(14)
        MLSV    0,MSFI
        ALRV    8,0
        LV      4,U04(14)
        MLSV    4,SFI
        ALRV    10,2
        LV      0,R04(14)
        MLSV    0,CFI
        STV     8,U10(14)
        LA      14,16(14)
        LDA     12,1(12)
        BCL     NZ,JLOOP
```

Fig. 7. QR RLS algorithm. (A) C language version of i and j loops, (B) TMS assembly language, and (C) ELM assembly language implementation of the j loop.

TABLE 4
Efficiency of TMS Assembly Implementations of RLS Filter Loops

Loop	L_A	L_{AB}	L_{ABV}	L_C	L	$L_C / L_{AB(V)}$	$L / L_{AB(V)}$
i	29 (+ div, sqrt)	41 (+ div, sqrt)		74 (+ div, sqrt)	56 (+ div, sqrt)	1.80	1.37
j (not inc. prologue & epilogue)			2	23	3	11.5	1.5

The j loop, as implemented on the ELM, is shown in Fig. 7C. It may be largely understood by way of the explanation in Section 5, new points in this example being the addressing offset from R14 on line 1, "Multiply logarithm scalar vector" on line 2, which multiplies all four registers $\mathrm{R0} \ldots \mathrm{R3}$ by the constant CFI from memory, and "Load address" and "Load decremented address" at the end, which respectively increment R14 by 16 and decrement R12 by 1. The latter is followed by a conditional branch back to the loop.

The experiment was repeated for filter order $\mathrm{N} = 8, 16, 32,$ 64, and 128. An input file spanned 2,000 time steps. Random noise, increasing in amplitude, was passed to the reference input, and $\frac{1}{2}\mathrm{N}$ time steps later was coupled into the signal. Inputs were quantized to 24-bit fixed-point values (23 bits plus sign) and thus took advantage of the entire 23-bit precision of the arithmetic. The same input data were presented to the TMS and the ELM, in each case being loaded into the memory of the respective development system before execution. In real life, the devices would be reading the data from an A-D converter, but this would take a similar length of time as that to access memory.

Both systems were timed (the TMS versions were run with and without the L2 cache), and the outputs were compared with that of an 80-bit Pentium to evaluate the SNR. An example of input and output is shown in Fig. 8. The SNR values for the results are presented in Fig. 9. The average TMS value is 108.4 dB and that of the ELM 117.5, so the ELM has delivered an average improvement of 9.1 dB or 1.5 bits. Since the 23-bit quantization has imposed a limit of

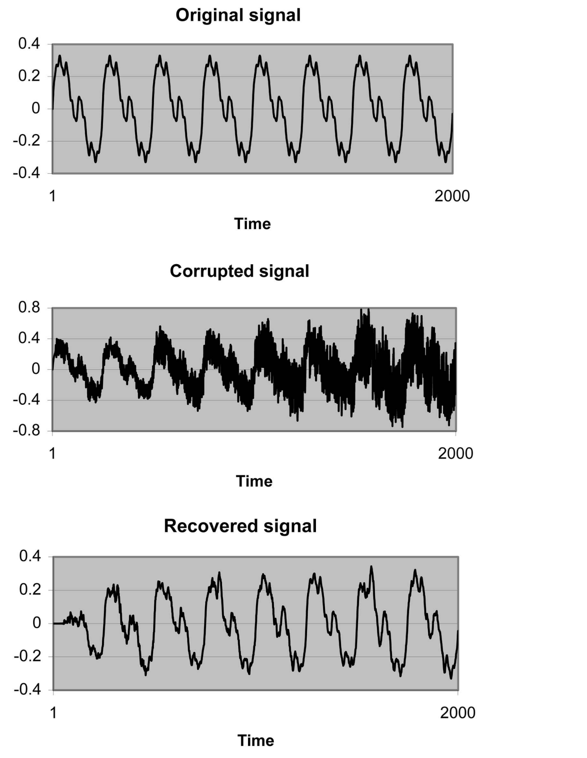


Fig. 8. Example of input and output data (output from ELM1A) for $\mathrm{N} = 32$ filter.

138.5 dB, we can conclude that the TMS has added 30.1 dB of processing noise, whereas the ELM has added 21.0 dB. The generally smoother progression of the SNR values throughout the range of the ELM results is also noteworthy, and appears to be consistent both with the results in Fig. 5 and with the intuitive expectation that SNR will rise and fall around some optimum with N, although we make no attempt to explain it.

The runtimes are presented in Fig. 10, which shows an $\mathrm{O}(\mathrm{N}^2)$ increase consequent on the 2D nature of the arrays. A knee is present in each curve, where the total amount of data exceeds the capacity of the caches. For the TMS with only a 4-Kbyte L1 cache, this is at $\mathrm{N} = 32$. For the ELM with its 8-Kbyte cache, it is at $\mathrm{N} = 64$, and for the TMS with a 64-Kbyte L2 cache, at $\mathrm{N} = 128$. Evidently there is little to choose between the ELM with one cache and the TMS with two, except for one specific instance at $\mathrm{N} = 64$, where the

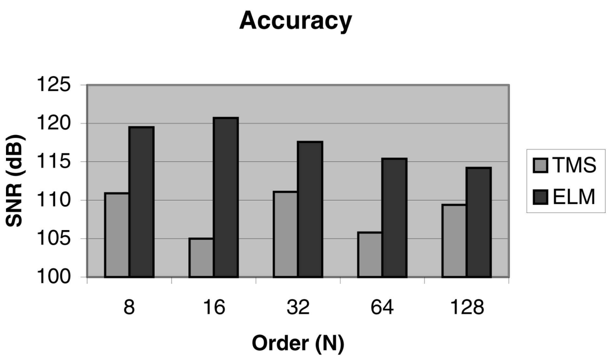


Fig. 9. Accuracy of TMS and ELM implementations of filter.

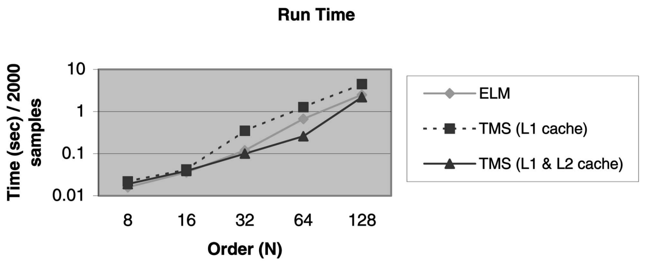


Fig. 10. Runtimes of TMS and ELM implementations of filter.

data are too large to fit into the cache on the ELM but just small enough to be cached at L2 on the TMS.

Figs. 7 and 11 together explain this result. With increasing N, the j loop increases in prominence as the i loop wanes. The j loop runs on the TMS in three cycles per iteration against an average of 5.25 on the ELM, the better performance on the TMS arising from the pipelining of the functional units. The i loop completes in 126 cycles on the TMS but 53 on the ELM, almost all of this difference being attributable to the division and square root operations. Furthermore, the ELM stalls for significantly longer than the TMS, obviously due to its lack of an L2 cache, except at $\mathrm{N} = 128$ where this occurs on both processors. The improved performance of the i loop is, however, so significant that, except at $\mathrm{N} = 64$, the ELM actually completes in slightly fewer cycles overall than the TMS does.

The longer cycle time but lower cycle count on the ELM result in the processors delivering a similar level of performance throughout. However, the ELM has done so despite what are evidently the major handicaps of having no L2 cache or vector pipeline. The dramatic simplifications inherent in the LNS division and square root operations have translated into an ability to dispense with a large measure of silicon complexity. The ELM has also achieved this result without running at its full design speed.

6.2 A Scalar Application: Recursive Ray Tracer

A demanding area of computational graphics is the ray tracer. An image is represented as a mesh of triangles into which a matrix of rays is projected from the viewer's eye. The first triangle on the path of each ray is visible to the viewer, and once identified its surface qualities are considered. If reflective, transparent, or both, then further rays are fired in the direction of reflection or refraction. A binary tree of recursive activations is built, and the color at each intersection point added to the total for that ray. To obviate testing each ray with each triangle, the latter are

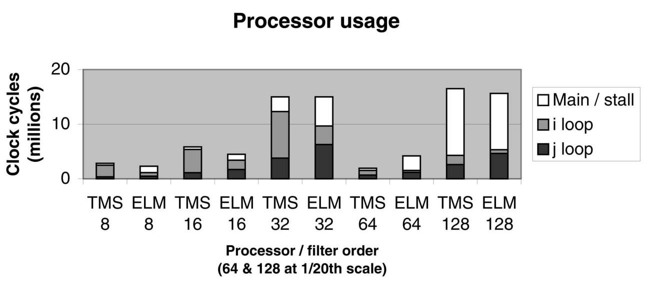


Fig. 11. Analysis of clock cycle usage in TMS and ELM implementations of filter.

TABLE 5
Efficiency of TMS Assembly Implementations of Ray Tracer Loops

Loop	L_A	L_{AB}	L_{ABV}	L_C	L	$L_C / L_{AB(V)}$	$L / L_{AB(V)}$
Ray-ellipsoid	32	44		82	51	1.86	1.16
Ray-plane	19 (+ div)	37 (+ div)		65 (+ div, fabsf)	45 (+ div)	1.76	1.22

grouped into a hierarchy of bounding axis-aligned ellipsoids. Only after a ray intersects an ellipsoid is it tested against the list of triangles within. There are two ray-ellipsoid intersection algorithms. The "algebraic" method treats the ellipsoid as a quadric, substitutes the ray parameters, and tests the discriminant. Programmatically, this involves a large number of multiterm SOPs. The "geometric" method rescales the ray into the ellipsoid's own space, effectively converting this into a sphere, and then makes a ray-sphere intersection test. This involves far fewer operations, but several are divisions or square roots. Intersection with the triangles is tested in two stages. The ray-plane algorithm rejects any rays that do not intersect the plane in which the triangle lies, and involves a division. Finally, Badouel's algorithm tests whether the intersection point lies within the triangle. This code is sparse and irregular but can be, and was, arranged so as to avoid significant use of division.

It would be very difficult to vectorize this application. The data consists of either scalars or three-element vectors. The control paths vary, even between successive iterations of the inner loops. Mapping a 3D model into a 1D memory results in fragmentation, so as the program will typically access randomly scattered blocks of between 8 and 32 words each, memory accesses are also difficult to vectorize.

A model was rendered at 256×256 pixels and comprised of 796 triangles within 86 ellipsoids in three levels, a total model size of 115 Kbytes. The TMS assembly language (Table 5) was highly optimal and a significant improvement on the compiled output.

There was no discernable difference between the outputs from the TMS and the ELM, which were similar to that shown in Fig. 13, except that this has random jitter on the refracted ray paths to simulate slightly opaque water. This increased the runtime unpredictably and was disabled during the timings to prevent discrepancies arising in the randomizers.

Runtimes were measured, excluding calls to initial runtime support on the TMS, for example, for storage allocation and cache initialization. The times were 14.9 s on the 150-MHz TMS using the algebraic algorithm and with both caches operational, and 10.9 s on the 125-MHz ELM with the geometric method. Restricted to the use of its L1 caches only, the TMS runtime increased to 241 s, probably because of thrashing in the program cache. Comparing the 125-MHz ELM with one cache to the 150-MHz TMS with two, the ratio of runtimes is thus 73 percent and that of clock cycles 61 percent. An analysis is presented in Fig. 14.

Unlike the digital filter, this application is intrinsically suited to the capabilities of the ELM, which runs with a lower cycle count than the TMS in every part of the algorithm. These gains have been achieved in different ways.

In the ray-triangle routine, the ratio of ELM/TMS clock cycles is 67 percent. The most heavily used part of this code is illustrated in Fig. 12, where the ratio for a pass through the complete sequence is 35/53. Broadly, the lower cycle count is attributable to the reduced latency of the multiplication and load operations, implicit loading, and the reduced branching latency.

In all its passes through the ray-plane loop, the TMS device consumed 999m clock cycles and the ELM 411m, a ratio of 41 percent. The processing included 11.9m divisions. Allowing 30 cycles for each on the TMS thus accounts for 357m cycles. Discounting the effect of these divisions, the ELM/TMS ratio would be 411/642 or 64 percent. The reduction in cycle count here stems from very similar factors to those just described for the ray-triangle routine.

In the ray-ellipsoid loop, the ratio of ELM/TMS cycles is 79 percent. As seen in Table 5, the TMS linear assembler has generated a particularly efficient schedule in this case, due to the fact that the code consists of a large number of independent SOPs which have allowed the processor to run at close to its peak vector rate. It is unlikely that the ELM would have improved on this directly. Instead, the algorithm was rewritten to use a smaller number of operations, despite several being divisions or square roots. Although this approach was counterproductive on the TMS, it exploited one of the main strengths of the LNS.

The data structure used in this algorithm was comprised of 115 Kbytes. This was far greater than the capacity of either L1 data cache, and since about 12 out of 64 Kbytes in the TMS L2 cache was occupied by frequently used program code, it was over twice the available size of the latter. Data accesses had little spatial locality, with random access across the entire structure. The small scattered accesses fit well into a fine-grained cache such as that on the ELM, which is eight words wide. In the TMS L2 cache with a 32-word line, more time is likely to be spent loading unnecessary data at the boundaries. Despite the availability of an L2 cache, the TMS device stalled for 21 percent of its overall runtime against 27 percent on the ELM.

In total, the ELM required only 61 percent of the clock cycles of the TMS, and did so while also dispensing with an L2 cache. It appears that, although some of this reduction is attributable to the low latency of division and square root, much also accrues from that of multiplication and from the streamlining of the load and control operations made in support of it.

7 Conclusion

As an alternative to FLP, we had proposed the use of the LNS for the representation and processing of real numbers. Although the advantages for multiplication, division, and square root were obvious, the question had been whether

A

```
if (td->u1 == 0) {
  /* uncommon case omitted */
}
else {
  beta = ((v0 * td->u1) - (u0 * td->v1))
    * td->recip1;
  if (beta < 0 || beta > 1) {
    goto quit;
  }
  alpha = (u0 - beta * td->u2)
    * td->recip2;
}
if (alpha < 0 || alpha + beta > 1) {
  goto quit;
}
```

B

```
           LDW     .D2T1    *+B13(8),A5
||         AND     .S1      A6,A5,A6
||         AND     .L1      A7,A5,A7
           LDW     .D1T2    *+A15[A6],B5
||         LDW     .D2T1    *+B13(4),A0
           LDW     .D2T1    *+B13(20),A3
           LDW     .D1T2    *+A15[A7],B4
           LDW     .D2T1    *B13,A4
||         ZERO    .S2      B0
           CMPEQSP .S2X     A5,B0,B0
   [!B0]   B       .S1      l1
||         SUBSP   .L1X     B5,A0,A0
           LDW     .D2T2    *+B13(12),B12
           STW     .D2T1    A3,*+SP(20)
           SUBSP   .L1X     B4,A4,A4
           STW     .D2T1    A0,*+SP(24)
           LDW     .D2T2    *+B13(16),B4
|| [ B0]   MVKL    .S2      _divf1,B5
           ; BRANCH OCCURS
; uncommon case omitted
l1:
           MPYSP   .M1      A5,A0,A12
           MPYSP   .M1X     B12,A4,A3
           LDW     .D2T1    *+B13(28),A0
           NOP              2
           SUBSP   .L1      A12,A3,A12
           MVKL    .S1      0x3f800000,A3
           MVKH    .S1      0x3f800000,A3
           NOP              1
           MPYSP   .M1      A12,A0,A0
           ZERO    .D1      A12
           NOP              2
           MV      .S2X     A0,B5
           CMPGTSP .S1      A0,A3,A1
||         CMPLTSP .S2X     B5,A12,B0
           OR      .S2X     B0,A1,B0
   [ B0]   B       .S1      L6
   [ B0]   ADDK    .S2      0xffffffff,DP
           STW     .D2T1    A0,*+SP(28)
   [ B0]   ADDK    .S1      0x10,A13
   [ B0]   MV      .D2      DP,B0
           NOP              1
           ; BRANCH OCCURS
           MPYSP   .M1X     B4,A0,A0
           LDW     .D2T2    *+B13(60),B5
           NOP              2
           SUBSP   .L1      A4,A0,A5
           NOP              3
           MPYSP   .M2X     A5,B5,B6
           NOP              3
           CMPLTSP .S2X     B6,A12,B0
L3:
   [ B0]   B       .S1      L6
   [!B0]   LDW     .D2T1    *+SP(28),A0
|| [ B0]   ADDK    .S2      0xffffffff,DP
   [ B0]   MV      .D2      DP,B0
|| [ B0]   ADDK    .S1      0x10,A13
           NOP              3
           ; BRANCH OCCURS
```

C

```
        L       0,U1(12)
        S       0,ZERO
        BC      NZ,TI1
* uncommon case omitted
TI1     LR      1,5
        ML      1,U1(12)
        LR      2,4
        ML      2,V1(12)
        SLR     1,2
        ML      1,RECIP1(12)
        LR      0,1
        BC      LT,TI9
        SLL     0,1
        BC      GT,TI9
        L       2,U2(12)
        MLR     2,1
        LR      0,4
        SLR     0,2
        DL      0,U1(12)
TI5     LR      3,0
        BC      LT,TI9
        ALR     3,1
        LR      2,3
        SLL     2,1
        BC      GT,TI9
```

Fig. 12. Excerpt from ray tracer: ray-triangle intersection. (A) Original HLL, (B) Equivalent in optimized TMS assembler, and (C) In optimized ELM assembler.

addition and subtraction could be carried out at least as fast and accurately as in FLP, not only in simulation but by working silicon. Anything less would be unlikely to gain acceptance. Once this were established, however, the LNS would become a viable alternative and the question would arise as to possible techniques for integrating it into a practical computer. How would the architecture of a new device need to be adapted to accommodate an LNS arithmetic unit? Would the theoretical gains actually translate into measurable improvements in performance?

We developed a new microprocessor, the ELM, based on a 32-bit logarithmic arithmetic unit. We found that additive operators would in fact be marginally more accurate than

Fig. 13. *Nymphaea alba*: 256×256 pixel ray traced image (with 512×512 adaptive supersampling) rendered in 12.0 s on 125-MHz ELM1A (available in color in the Computer Society Digital Library at http://

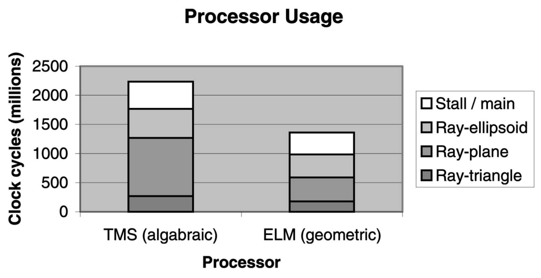


Fig. 14. Analysis of clock cycle usage in TMS and ELM implementations of ray tracer.

those of FLP, and as multiplicative operators return exact results, a typical kernel making equal use of each would incur roughly half the total error of FLP arithmetic. We had previously verified this expectation by using a simulation of the proposed unit. Running the same experiments again, this time on the fabricated microprocessor, we obtained an almost identical set of results. Clearly we may conclude that the LNS can indeed be implemented in silicon, with better accuracy than that of FLP arithmetic.

We evaluated its speed by comparison with a commercial FLP processor of similar fabrication technology. Each processor had six principal ALUs: two adders, two multipliers, and two integer units on the FLP processor, versus two adders and four combined multiplier/integer units on the LNS device. The FLP processor was the result of its designers' best efforts to produce a first-class arithmetic unit, in real silicon, integrated into a microprocessor in the most favorable way, and the LNS processor was the result of our own best efforts to do the same. We were, of course, comparing our first-off silicon and circuit board with the last of several revisions of the FLP device, but were nonetheless able to run the ELM at $\frac{5}{6}$ of the clock speed of the latter. Additive times were marginally better on the ELM, multiplications were 3.4 times the speed, and divisions and square roots were many times faster than the FLP software implementation. If, as we expect, a reworked PCB enables us to run the ELM a little closer to its design speed, which was similar to that of the FLP device, then we may simply compare clock cycles. In this case, the ELM would offer improvements to addition and multiplication of 1.3 and 4 times, respectively. In any event, LNS addition and subtraction can be implemented not only with "better than FLP" accuracy, but also with better speed. The criteria set out above have been satisfied, and we conclude that the LNS is a clear improvement on 32-bit FLP arithmetic.

In order to harness this new power, we had considered how the design of a microprocessor might be adapted to achieve the best synergy with desirable features of the arithmetic. The LNS is intrinsically of low latency, and this philosophy pervaded the design of the ELM. Functional unit pipelining was dispensed with, and an RM instruction set adopted to eliminate explicit loading. However, as the multiplier is effectively cost free, throughput was enhanced by functional unit replication, with a resulting short-vector capability.

Substantial case studies pinpointed how and where the use of LNS arithmetic had been of value. These studies also served to experiment with possible application areas for the ELM and to demonstrate its overall capability. Measuring by time, that is, despite the fact that the ELM was running at a slower clock speed, we found in one case that the inherent simplification of the division and square root operations almost entirely compensated for the lack of an L2 cache. In the other, we not only dispensed with the L2 cache but also obtained a reduction to 73 percent of the execution time. This arose through 1) the reduced latency of all real operations and of architectural decisions such as a tightly coupled L1 cache made in support of it, and 2) rewriting the algorithm to use fewer operations, despite several being divisions and square roots. In terms of clock cycles, this reduction was to 61 percent.

The ELM delivered an improvement in accuracy equivalent to 1.5 bits; in the application concerned, a reduction in processing noise to about $\frac{2}{3}$ of that of the FLP device.

In summary, the LNS delivered faster execution than FLP arithmetic, more accurate results, and did so with a significant reduction in architectural complexity. Power dissipation and silicon area are yet to be considered (although preliminary results are encouraging), but on the basis of the present results, the LNS may now be the preferred technique for 32-bit work. This is particularly so in emerging algorithms that are naturally cast in terms of divisions, but where a great deal of programmer time is currently spent in reworking to alternative formulations. A point that we made in [1] may be repeated here: the design of an LNS unit requires significant attention to only one block of circuitry. If the adder/subtractor could be reduced in latency to less than that of a 32×32 bit multiplier, then assuming an equal proportion of additions and multiplications, the LNS would be faster than *fixed-point* arithmetic. That is the next step.

Acknowledgments

The authors gratefully acknowledge the support of the European Strategic Programme for Research and development in Information Technology (ESPRIT) Long-Term Research Programme, which funded this work under Grants 23544 and 33544. Partners in the project were the University of Newcastle upon Tyne, University College Dublin, Academy of Sciences of the Czech Republic, Prague, Philips Research, Eindhoven, and Massana, Dublin. We also thank Philips Semiconductors, Nijmegen, where the ELM devices were fabricated. The initial design of the evaluation board was done by MEng student Steve Fishwick.

References

[1] J.N. Coleman, E.I. Chester, C. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 702-715, July 2000, erratum, vol. 49, no. 10, p. 1152, Oct. 2000.

[2] J.N. Coleman and E.I. Chester, "A 32-Bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point," *Proc. 14th IEEE Symp. Computer Arithmetic,* 1999.

[3] F.J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20-Bit Logarithmic Number System Processor," *IEEE Trans. Computers,* vol. 37, pp. 190-200, 1988.

[4] D.M. Lewis, "An Architecture for Addition and Subtraction of Long Wordlength Numbers in the Logarithmic Number System," *IEEE Trans. Computers,* vol. 39, pp. 1325-1336, 1990.

[5] D. Yu and D.M. Lewis, "A 30-b Integrated Logarithmic Number System Processor," *IEEE J. Solid-State Circuits.,* vol. 26, pp. 1433-1440, 1991.

[6] D.M. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit," *IEEE Trans. Computers,* vol. 43, pp. 974-982, 1994.

[7] D.M. Lewis, "114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications," *IEEE J. Solid-State Circuits.,* vol. 30, pp. 1547-1553, 1995.

[8] V. Paliouras, J. Karagiannis, G. Aggouras, and T. Stouraitis, "A Very-Long Instruction Word Digital Signal Processor Based on the Logarithmic Number System," *Proc. Fifth IEEE Int'l Conf. Electronics, Circuits and Systems,* 1998.

[9] J.N. Coleman, C.I. Softley, J. Kadlec, R. Matousek, M. Licko, Z. Pohl, and A. Hermanek, "The European Logarithmic Microprocessor—A QR RLS Application," *Proc. 35th IEEE Asilomar Conf. Signals, Systems, and Computers,* 2001.

[10] M.G. Arnold, "A VLIW Architecture for Logarithmic Arithmetic," *Proc. Euromicro Symp. Digital System Design,* 2003.

[11] M.G. Arnold and C. Walter, "Unrestricted Faithful Rounding Is Good Enough for Some LNS Applications," *Proc. 15th IEEE Symp. Computer Arithmetic,* 2001.

[12] J.N. Coleman, C.I. Softley, J. Kadlec, R. Matousek, M. Licko, Z. Pohl, and A. Hermanek, "Performance of the European Logarithmic Microprocessor," *Proc. SPIE Ann. Meeting,* 2003.

[13] C.H. Chen, R.-L. Chen, and C.-H. Yang, "Pipelined Computation of Very Large Word-Length LNS Addition/Subtraction with Polynomial Hardware Cost," *IEEE Trans. Computers,* vol. 49, pp. 716-726, 2000.

J. Nicholas Coleman was initially educated in music, receiving the BA degree from York University. His first employment was in commercial applications and systems programming. Becoming increasingly interested in engineering, he joined Plessey Telecommunications Research, Poole, Dorset. Working initially as a software engineer and then as a hardware engineer, he was responsible for the design of a custom processor for real-time test of the System-X telephone exchange. At Brunel University, Uxbridge, Middlesex, he then began work on a PhD, for which he designed and built a special-purpose dataflow computer. After receiving the PhD degree, he joined Newcastle University, where he is a lecturer in computer engineering and the coordinator of the European Strategic Programme for Research and development in Information Technology (ESPRIT) Project on which this work is based. He has recently founded a spin-off company, Northern Digital, which, with research and development funding from the United Kingdom government, aims at commercializing the outcome of this project. His research interests include high-speed processor design and processor-intensive applications such as graphics.

Chris I. Softley received the BEng (hons) degree in microelectronics and software engineering from the University of Newcastle upon Tyne in 1999. His dissertation focused on arithmetic cells for a logarithmic number system (LNS) CPU. He was then employed by the university as a research assistant, seconded to the Philips Research Laboratories in Eindhoven, The Netherlands. Here, he worked primarily on the design of the European Logarithmic Microprocessor, under the European Strategic Programme for Research and development in Information Technology High-Speed Logarithmic Arithmetic Unit (ESPRIT HSLA) Project. Upon completing this work in 2002-2003, he moved to Switzerland and joined Photonfocus AG, becoming the head of sensor development in 2005. He currently develops high-performance CMOS image sensors for industrial applications which have a combined linear-logarithmic response for high dynamic range. He is also with the Hochschule für Technik, Rapperswil, where he supervises research on low-power low-noise readout circuits for fast CMOS imagers. His research interests include image sensors, energy-efficient analog circuits, computer arithmetic, number systems, and processor architecture, particularly as applied to vision and visualization systems. He is a member of the IEEE and the IEEE Computer Society.

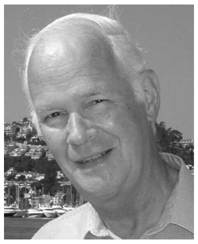
Jiri Kadlec received the MSc degree from the Czech Technical University in Prague, in 1982 and the PhD degree from the Academy of Sciences of the Czech Republic in 1987. Since 1990, he has been with the Institute of Information Theory and Automation at the Academy of Sciences of the Czech Republic, where he is currently the head of the Department of Signal Processing. He was a researcher at Ruhr University, Bochum, Germany, in 1989, the University of Athens, Greece, in 1990, Queen's University of Belfast, United Kingdom, from 1992 to 1995, and the Katholieke Universiteit Leuven, Belgium, from 1995 to 1996. His research interests include recursive system identification algorithms suitable for FPGA, rapid prototyping of advanced signal processing algorithms, and scalable floating-point arithmetic for FPGA SoC designs.

Rudolf Matousek received the MSc degree in automation in transportation sciences from the Czech Technical University in Prague, in 2000. His research interests include computer arithmetic, FPGA implementation of DSP algorithms, and design methodology and tooling for the dynamic reconfiguration of FPGA devices.

Milan Tichy received the MSc and PhD degrees from the Czech Technical University in Prague, in 1999 and 2006, respectively. In 2003, he was awarded the two-year Marie Curie Fellowship, which he spent in the Department of Computer Science at Trinity College, Dublin, Ireland, from 2004 to 2006. He is currently a senior researcher in the Department of Signal Processing at the Institute of Information Theory and Automation of the Academy of Sciences of the Czech Republic. His research interests include reconfigurable systems, parallel algorithms and architectures, parallel adaptive algorithms, VLSI implementations, and embedded systems. He is a member of the IEEE and the IEEE Computer Society.

Zdenek Pohl received the bachelor's degree in electrical engineering from the Czech Technical University in Prague. He is currently working toward the PhD degree at the Czech Technical University. He is also a researcher in the Institute of Information Theory and Automation at the Academy of Sciences of the Czech Republic. His research interests include FPGA implementations of signal processing algorithms, speech coding, and rapid prototyping.

Antonin Hermanek received the MSEE degree from the Czech Technical University in Prague, in 1998 and the PhD degree from the Université Paris-Sud, Orsay, in 2005. He is a researcher in the Institute of Information Theory and Automation at the Academy of Sciences of the Czech Republic. His research interests include blind equalization, multiple-input, multiple-output (MIMO) and orthogonal frequency-division multiplexing (OFDM) communication systems, FPGA implementation of DSP algorithms, array processing, and rapid prototyping for signal processing. He has published more than 30 papers in these areas. He is a member of the IEEE.



Nico F. Benschop received the MSc degree in electronics from Delft University, The Netherlands, in 1966 and the PhD degree from Waterloo University, Ontario, Canada in 1971. From 1970 to 2002, he was with Philips Research Laboratories, Eindhoven, The Netherlands, researching design methods for digital VLSI in logic, arithmetic, and state machines. After retirement, he finished a book, *Associative Digital Network Theory*, combining these three main aspects under one heading of the finite associative algebra of function composition (semigroups) to improve structural insight into complex computer circuits and algorithms.