

ISSN 1063-7192
Volume 19, Number 2-3
(June-September 2004)

PARALLEL ALGORITHMS and APPLICATIONS

Edited by
G. M. Megson, *The University of Reading, UK*

SEP 7 2004

ct
CURRENT
PERIODICAL
QA
76.642
.P369
V. 19:2/3
2004



Taylor & Francis
Taylor & Francis Group

Parallel Algorithms and Applications

Editor-in-Chief and Regional Editor, Europe: Professor Graham Megson

Department of Computer Science, University of Reading, PO Box 225, Whiteknights, Reading, Berkshire RG6 6AY, UK. Email: g.m.megson@reading.ac.uk

Regional Editor, North America: I. Stojmenovic (Ivan)

Computer Science Department, University of Ottawa, Ottawa, Ontario K1N 9B4, Canada. Email: ivan@site.uottawa.ca

Regional Editor, Eastern Europe: Professor Dr O. Brudaru (Octav)

"Gh. Asachi" Technical University of Iasi, Department of Production Systems Engineering, Copou 22, 6600 Iasi, Romania. Email: brudaru@tuiasi.ro

Regional Editor, Asia: L. M. Patnaik

Microprocessor Applications Lab, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India. Email: lalit@micro.iisc.ernet.in

Regional Editor, Australasia: F. Dehne (Frank)

School of Computing and Information Technology, Nathan Campus, Griffith University, Brisbane, Queensland 4111, Australia. Email: frank@dehne.net

Editorial Board:

S. G. Akl
School of Computing
Queen's University
99 University Avenue
Kingston
Ontario K7L 3N6, Canada
Email: akl@cs.queensu.ca

D. Das
Department of Computer
Science and Engineering
University of Texas at
Arlington
Arlington TX 76019-0015
USA
Email: das@uta.edu

G. Loizou
School of Computer Science
and Information Systems
Birkbeck College
Malet Street
London WC1E 7HX
UK
Email: george@dcs.bbk.ac.uk

M. Thuné
Department of Scientific
Computing
Uppsala University
PO Box 337
751 05 Uppsala
Sweden
Email: michael.thune@it.uu.se

V. Alexandrov
Department of Computer
Science
The University of Reading
PO Box 225
Whiteknights
Reading
Berkshire RG6 6AY, UK
Email:
v.n.alexandrov@reading.ac.uk

G. A. Gravvanis
Hellenic Open University,
6 Livathous Street
11362 Athens
Greece
Email: gravvanis@eap.gr

T. Plaks
Faculty of Business, Computing
and Information Management
London South Bank University
103 Borough Road
London SE1 0AA
Email: plakst@lsbu.ac.uk

M. Vajteršić
Institut für Scientific
Computing
Universität Salzburg
Jakob Haringer Strasse 2
5020 Salzburg
Austria
Email: marian@cosy.sbg.ac.at

M. Clint
Department of Computer
Science
Queen's University Belfast
University Road
Belfast BT7 1NN
Northern Ireland
Email: m.clint@qub.ac.uk

C. Lengauer
Fakultät für Mathematik und
Informatik
Universität Passau
Innstrasse 33
94032 Passau
Germany
Email:
lengauer@fmi.uni-passau.de

J. Teich
Lehrstuhl für Informatik 12
Hardware-Software-Co-Design
Friedrich-Alexander
Universität Erlangen-Nürnberg
Am Weichselgarten 3
91058 Erlangen
Germany
Email: juergen.teach@
informatik.uni-erlangen.de

S. A. Zenios
Department of Public and
Business Administration
School of Economics and
Management
University of Cyprus
CY-1678 Nicosia
Cyprus
Email: zenios@ucy.ac.cy

Aims and Scope

Parallel Algorithms and Applications aims to publish high quality scientific papers arising from original research and development from the international community in the areas of parallel algorithms and applications for parallel architectures.

Topic areas include:

- Algorithm synthesis, transformation techniques and CAD/CAM tools
- Emergent computing structures (e.g. genetic, neural, evolvable systems)
- Scheduling, simulation and performance evaluation
- Scientific, industrial and commercial applications
- Systems software and compilers
- Advanced architectures (e.g. vector, pipelined, parallel, distributed, clusters)
- Application specific techniques (e.g. FPGA, systolic, cellular)
- Algorithm design, analysis and usage

Papers in other areas will be considered subject to the demonstration of sufficient novelty and originality in the application of explicit or implicit parallelism in algorithm/architecture structures.

Parallel Algorithms and Applications**Volume 19, Issue No. 2–3****Contents****Special Issue:** Parallel and Distributed Algorithms**Guest Editors:** George A. Gravvanis and Hamid Arabnia

Editorial	77
A bit-serial floating-point unit for a massively parallel system on a chip..... <i>M. Schimpler, B. Schmidt and H.-W. Lang</i>	79
A locality-conscious load-balancer based on negotiations in dynamic unstructured mesh computations..... <i>A.S. Mohamed and V.S. Baydogan</i>	97
Deadlock-free dynamic reconfiguration over InfiniBand networks	127
<i>B. Zafar, T.M. Pinkston, A. Bermúdez and J. Duato</i>	
A portable software architecture for mesh-independent particle tracking algorithms..... <i>J.-R.C. Cheng, M.T. Jones and P.E. Plassmann</i>	145
A distributed normalized explicit preconditioned conjugate gradient method	163
<i>G.A. Gravvanis, K.M. Giannoutakis and N.M. Missirlis</i>	
HP java: programming support for high-performance grid-enabled applications	175
<i>H.-K. Lee, B. Carpenter, G. Fox and S.B. Lim</i>	

A BIT-SERIAL FLOATING-POINT UNIT FOR A MASSIVELY PARALLEL SYSTEM ON A CHIP

MANFRED SCHIMMLER^{a,*}, BERTIL SCHMIDT^{b,†} and HANS-WERNER LANG^{c,‡}

^a*Institut für Informatik und Praktische Mathematik, CAU Kiel, 24098 Kiel, Germany;* ^b*School of Computer Engineering, Nanyang Technological University, N4-02a-32 Nanyang Avenue, Singapore, Singapore 639798;* ^c*Fachhochschule Flensburg, 24943 Flensburg, Germany*

(Received 23 October 2003; In final form 8 April 2004)

This paper presents the design of a new bit-serial floating-point unit (FPU). It has been developed for the processors of the instruction systolic array (ISA) parallel computer model. In contrast to conventional bit-parallel FPUs the bit-serial approach requires a different data format. Our FPU uses an IEEE compliant internal floating-point format that allows a fast least significant bit (LSB)-first arithmetic and can be efficiently implemented in hardware.

Keywords: Bit-serial floating point units; Massively parallel processors; Systolic arrays; Floating-point unit

INTRODUCTION

Modern VLSI technology allows us to integrate massively parallel systems on a single chip. The area limitations for the processors of such systems require small but efficient computational units [11,14,20]. In particular, it may motivate the choice of a bit-serial data organization of the individual processors. The fine-grained processor arrays are used in a variety of applications where identical operations have to be performed on large data sets such as signal, image and video processing [16,20], cryptography [15], biological sequence analysis [17], scientific computing [19], visualization [11,18] and medical imaging [13]. Implementations of relevant numerical and signal processing algorithms often require floating arithmetic in order to achieve adequate accuracy. Thus, the integration of efficient floating-point arithmetic is essential to these processors.

Motivated by the design of an instruction systolic array (ISA) with 1024 processors on one chip [14], an FPU has been designed that meets these requirements. Its bit-serial structure allows for a fine-grained pipelined implementation on a minimal area of silicon. The design has been guided by experiences with a lower-integrated predecessor version of the processor array, which is available as an add-on board for a PC [6].

*Tel.: + 49-431-880-4480. Fax: + 49-431-880-4054. E-mail: masch@informatik.uni-kiel.de

†Corresponding author. Tel: + 65-67905194. Fax: + 65-67926559. E-mail: asbschmidt@ntu.edu.sg

‡Tel.: + 49-461-8051235. Fax: + 49-461-8051527. E-mail: lang@fh-flensburg.de

In the early days of massively parallel machines the individual processors had no integrated FPUs [3]. But when MFlops evolved as one of the most important measures for supercomputer performance, the designers added commercial bit-parallel floating-point processors as coprocessors to small PE-clusters of their massively parallel systems [21,22]. This approach was useful as long as the integrated circuit technology did not allow fitting both the PEs and the FPUs on the same die. But with the advances in semiconductor technology in the last decade, designers tended to increase the computational capabilities of the individual processors rather than designing parallel systems of floating-point processors.

More recent work has focused on FPU design for FPGAs. In Ref. [10], a serial multiplier is combined with a parallel adder in order to achieve high performance with a limited number of CLBs. A similar approach is taken in Ref. [9], in order to investigate the resource utilization for the FPU in modern FPGAs. In Ref. [8], a technique of parameterizing the FPU-design by the desired precision is discussed. Another adaptive approach has recently been taken in Ref. [7]. A generator tool is presented that designs an FPU for FPGA implementation dependent on the input of performance requirements. Even though the performance results are impressive, all these designs are currently not suitable for a massively parallel processor array, since they are not optimized with respect to area and power consumption.

The rest of this paper is organized as follows. The second section describes the ISA principle. An overview of the individual ISA processor architecture is given in the third section. The design methods for implementing efficient floating-point operations on bit-serial hardware are described in the fourth section. The fifth section presents the corresponding execution units. The performance of several floating-point operations is given in the sixth section. The seventh section concludes the paper.

PRINCIPLE OF THE ISA

The ISA [4,5] is a mesh-connected processor grid, where the processors are controlled by three streams of control information: instructions, row selectors and column selectors (see Fig. 1). The instructions are input in the upper left corner of the processor array, and from there they move step by step in horizontal and vertical direction through the array. This guarantees that within each diagonal of the array the same instruction is active during each clock cycle. In clock cycle $k + 1$ processor $(i + 1, j)$ and $(i, j + 1)$ execute the instruction that

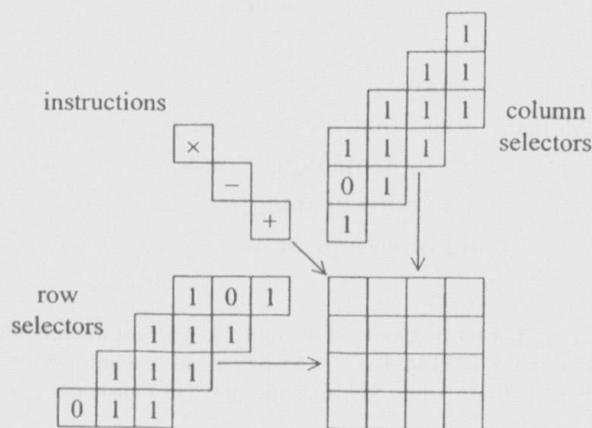


FIGURE 1 Control flow in an ISA.

has been executed by processor (i, j) in clock cycle k . The selectors also move systolically through the array: the row selectors move horizontally from left to right; the column selectors move vertically from top to bottom. Selectors mask the execution of the instructions within the processors, i.e. an instruction is executed if and only if both selector bits, currently in that processor, are equal to one. Otherwise, a no-operation is executed.

Every processor has read and write access to its own memory. Besides that, it has a designated *communication register (C-register)* that can also be read by the four neighbour processors. Within each clock phase reading access is always performed before writing access. Thus, two adjacent processors can exchange data within a single clock cycle in which both processors overwrite the contents of their own C-register with the contents of the C-register of its neighbour. This convention avoids read/write conflicts and also creates the possibility to perform parallel prefix operations efficiently, e.g. row sum.

PROCESSOR ARCHITECTURE

The processor architecture is part of a chip design containing an ISA of size 32×32 [14]. The main components of the processor architecture are a set of 64 data registers, the communication register, a unit for integer addition, logical operations and conditional instructions, a 16-bit multiplier, and a shifter/adder for floating-point arithmetic (see Fig. 2).

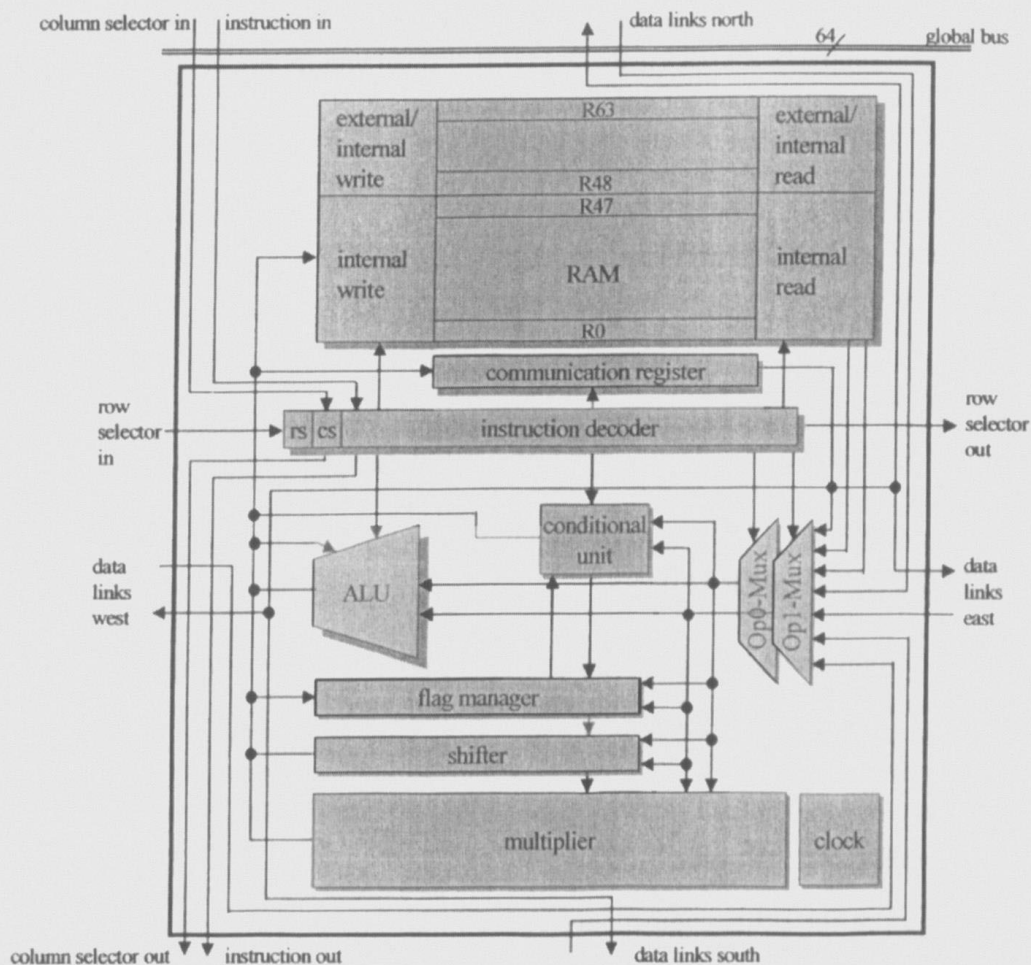


FIGURE 2 Block diagram of the processor architecture.

The data word length is 16 bits. Data is processed bit-serially, such that the execution of each instruction takes exactly 16 clock cycles. However, there are macro-instructions formed as series of elementary instructions in order to handle data types consisting of more than one machine, word, e.g. double precision floating-point numbers (see "Floating Point Operations" section). After receiving an instruction, the processor stores it in the instruction register, decodes the two operand addresses and the destination address, retrieves the operands from the register file, executes the instruction, writes the result back to the destination register, and passes the instruction to the next processor. Since all this is done bit-serially, it can be pipelined on bit level, such that a new instruction can be fetched and processed every 16 clock cycles. For instance, the first bit of the result is computed already after the second bits of the operands have entered the respective unit.

The instructions are 3-address instructions (two operand registers—op0 and op1 and destination register—dest) coded in two 32 bits. Several instructions are related to floating-point operations. However, these instructions in general do not make sense on their own, but only within macro-instructions, i.e. a sequence of instructions that realize a certain operation on floating-point numbers. The instruction

expadd op0, op1, dest;

computes the sum of the exponents of two floating-point numbers, as required for a multiplication. Similarly, there exist instructions EXPSUB for computing the difference of two exponents and EXPSEL for selecting the greater exponent as the result exponent in the case of an addition of two floating-point numbers. The instructions SHADD (shift and add), shsub (shift and subtract), nrm (normalize), LDMUL (load multiplier), and MULT (multiply) are used in macro-instructions for computing the mantissa of the result. The instruction

mode length;

is used for configuration of the floating-point control for handling single precision or double precision numbers. The instructions

mtw op0, dest;

expw op0, dest;

support the conversion between the IEEE floating-point format [1] and the internal format, forming the mantissa words and the exponent word of the internal format.

FLOATING POINT OPERATIONS

The processor supports and double precision floating-point arithmetic floating-point registers are implemented by groups of 16-bit data registers, and floating-point operations are implemented as instruction macros, i.e. as sequences of elementary instructions.

Internal Floating Point Format

The special cases for IEEE numbers are coded in all bits of the number representation. For example, in order to find out if a number equals infinity the complete mantissa and the complete exponent have to be compared with 0 and -1 , respectively.

Therefore, a bit-serial compare to infinity (required for every arithmetic operation) would need at least 32 internal clock cycles for single and 64 for double. To solve this problem a conversion into an internal format is performed. The internal representation of a double X consists of five 16-bit words $X[0], \dots, X[4]$. Here, $X[4]$ contains the exponent of the double, the sign, the kind of representation (absolute value/sign or 2-complement), and information about possible special cases (infinity, zero, NaN = not a number). $X[0]$ to $X[3]$ contain the significant (mantissa) of the double, extended to 64 bits.

$X[4]$ Exponent:	Sm	T1	T0	R	Se	Se	e9	e8	e7	e6	e5	e4	e3	e2	e1	e0
------------------	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----	----

Sm: sign of the significant, 0: positive, 1: negative

T1, T0: encoding of special cases: 00: normal number, 01: zero, 10: INIF (infinity), 11: NaN

R: representation of significant: 0: absolute value/sign representation, 1: 2-complement representation

Se: sign of exponent (two identical bits for overflow detection)

$e0, \dots, e9$: binary representation of the exponent E in 2-complement (little endian mode)

64 bit of the significant $M = (X[3], X[2], X[1], X[0])$, where 63 is the most significant bit (MSB):

$X[3]$ significant:	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
$X[2]$ significant:	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
$X[1]$ significant:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
$X[0]$ significant:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The value of such a double is:

$$0 \cdot (-1)^{\text{sm}} \quad \text{if } T0 = 1 \text{ and } T1 = 0;$$

$$\text{INF} \cdot (-1)^{\text{sm}} \quad \text{if } T0 = 0 \text{ and } T1 = 1;$$

$$\text{NaN} \quad \text{if } T0 = 1 \text{ and } T1 = 1;$$

$$(-1)^{\text{sm}} \cdot 0 \cdot M \cdot 2^E \quad \text{if } T0 = 0, T1 = 0 \text{ and } R = 0;$$

$$(-1)^{\text{sm}} \cdot 0 \cdot M' \cdot 2^E \quad \text{if } T0 = 0, T1 = 0 \text{ and } R = 1, \text{ where } M' \text{ is the 2-complement of } M;$$

Double numbers can be normalized or not normalized. A positive number and a number in absolute value/sign-representation is normalized iff bits 63, 62, 61 of the significant are 001, i.e. bits 63 and 62 hold the sign and then in bit 61 comes the MSB of the significant. As usual in absolute value/sign representation, the significant is positive and the sign is stored in the sm-bit of the exponent (see $X[4]$). A negative number in 2-complement is normalized iff bits 63, 62, 61 of the significant are 110. Again, bits 63 and 62 form the sign and it follows the most significant digit of the significant represented in 2-complement.

The four 16-bit words of the IEEE format are converted into 5 words of the internal format. The 52-bit mantissa is stored in four words, where in the first word an explicit 1 is introduced (this 1 is implicit in the IEEE representation). The first two 0's serve as two sign bits of the mantissa. Two sign bits are useful for detecting overflows: an overflow after an addition changes the second sign bit and a right shift for normalization is necessary. The exponent is stored in the last 12 bits of a separate word, also with two sign bits. The value 01 of the sign bits indicates *exponent overflow*, whereas 10 indicates *exponent underflow*. In these cases, the number is changed to INF or 0, respectively. The special values zero, INF and NaN are coded using the tag stored in bit 14 and 13 of the exponent word. In the case of a special value, the actual value of the mantissa is irrelevant. Bit 12 is used internally for indicating if the mantissa is in 2-complement representation after an addition or subtraction. The conversion is supported by special instructions of the processor.

Floating Point Addition

An addition of two floating-point numbers is performed in the usual way: the two exponents are compared, the mantissa of the operand with smaller exponent is shifted to the right appropriately, then the mantissas are added. If any operand is negative and not yet in 2-complement representation, its mantissa is complemented before. The bit-serial organization allows on-the-fly computation with one single half adder. If any operand is 0, its mantissa is set to 0. The result mantissa is in 2-complement representation. The result exponent is the larger exponent of the two operands. The tag of the result is computed depending on the tags of the two operands according to Table I.

The sign of the result is the sign of the result mantissa, except in the case that any operand is INF, then the sign is equal to the sign of this operand. If the result is 0 or NaN, the sign is irrelevant. Since the operands consist of more than one word (e.g. of four mantissa words in the case of double precision floating-point numbers), shifting of a mantissa by more than 15 bits is realized by displaced addressing of the operands (see Fig. 3).

In each case, in order to realize a shift distance of s , in the first step A_{-1} and B_{-1+k} are addressed, where $k = (s \text{ div } 16) + 1$. B_{-1+k} is loaded into the shifter and delayed by $d = 16(k - s)$ bits. In the next step A_0 and B_{0+k} are addressed. While B_{0+k} is loaded to the shifter, the d bits of B_{-1+k} appear at the output of the shifter and are added to A_0 . In the last step A_3 and B_{3+k} are addressed. However, B_{3+k} is an invalid operand, therefore it is cancelled, i.e. set to 0...0 or to 1...1 depending on the sign of the mantissa. If k becomes greater than 4, all B -operands are cancelled, i.e. set to 0...0. In the case that B has the greater exponent, then the difference of the exponents is negative and A has to be shifted. The situation is essentially the same as above with the roles of A and B interchanged.

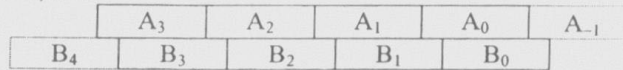
TABLE I Computation of result tags in the case of an addition

	X	0	INF	NaN
X	X^*	X	INF	NaN
0	X	0	INF	NaN
INF	INF	INF	$INF^\dagger$	NaN
NaN	NaN	NaN	NaN	NaN

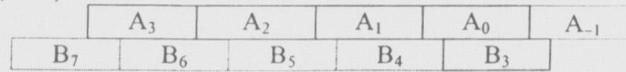
* Except the case that the result mantissa is 0, then the result tag is 0.

† Except the case that the signs of the two operands are different, then the result tag is NaN.

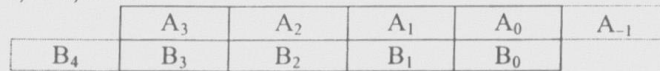
$s=5, k=1, d=11$:



$s=53, k=4, d=11$:



$s=0, k=1, d=16$:



$s \geq 64, k \geq 5$:

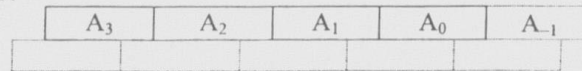


FIGURE 3 Addition of two 64-bit mantissas.

Normalization

After an addition, in general the result mantissa is not in normalized form, i.e. there are not exactly two leading 0's or 1's. Several cases are possible: there is one leading digit, two leading digits, (this is the normalized case), three through 63 leading digits, or 64 leading digits. In the latter case, the result is considered to be 0, if the leading digits are 0. In all other cases a left shift by $-1 \dots 62$ is necessary so that the mantissa starts with 001 or 110. The exponent has to be corrected by this shift distance A counter; the leading digits counter (LDC) is used to count the number of leading digits (-2) of the result produced during an addition/subtraction or multiplication. The LDC also records the first bit of the mantissa, indicating the sign of the mantissa, and detects if the mantissa is 0. Before normalization, the value of the LDC is transferred into the shift control register. But now this value has to be interpreted in a different way, since it is the distance for a left shift instead of a right shift as in the case of an addition. However, this can be done using the same hardware for normalization as for addition.

Floating Point Multiplication

A multiplication of two 64-bit mantissas is performed according to the scheme shown in Fig. 4, where e.g. $A_3 A_2 A_1 A_0 B_0$ denotes the 80-bit product of $A_3 A_2 A_1 A_0$ with B_0 . The result would be 128 bit long, however, only the most significant 64 bits can be used for the result mantissa F . Therefore the above scheme is truncated, in order to reduce the number of 16-bit multiplications. The scheme is shown in Fig. 5.

Addition of the intermediate results is performed simultaneously with the multiplications in the multiplier, which is actually a multiply-and-add unit. The error introduced by the truncation amounts to at most 3 bits. This seems to be tolerable, because the precision of the internal floating point format is 61 significant bits, whereas the original IEEE format has only

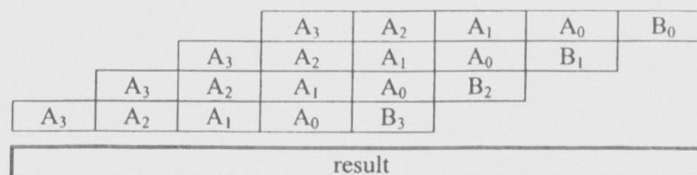


FIGURE 4 Multiplication scheme for 4-word operands.

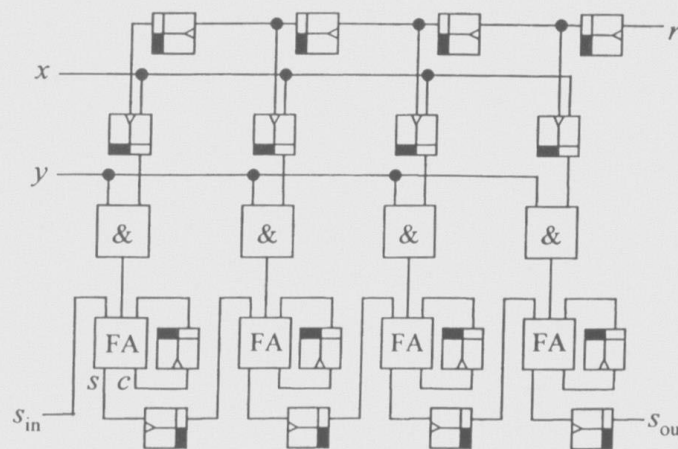


FIGURE 6 The original multiplier architecture for unsigned integers.

This multiplier design has been used in the parallel computer Systola 1024 [5]. One interesting property of this architecture is its capability to perform *multiply-add* operations. Each bit operation consists of the addition of a new bit product to the intermediate result and a shift operation. Therefore, the interconnection between adjacent full adders form a shift register (the sum register). Now we can use this shift register to add an additional operand. The only thing we have to do is to shift the third operand into the sum register of the multiplier with the LSB first. The first bits of the summand can be shifted in concurrently with the x operand. During the multiplication, in every clock cycle, we input one additional bit of the summand while one bit of y is processed.

The multiplier is capable of executing two different instructions, corresponding to the two phases for the multiplication. In the first phase, the operand x is loaded into the multiplicand register. We call this instruction LDMUL (load multiplicand). During the execution of a LDMUL-instruction, the y -input is forced to be 0. If a multiply-add operation is executed the least significant word of the operand s is shifted into the sum register during the LDMUL-instruction, too. The resulting bits produced by the first LDMUL-instruction are irrelevant.

The second phase is the multiplication with y . The instruction is called MULT (multiply). y is consumed bit by bit as described above. In a MULT-instruction, the x -input is not interpreted. The s -input is used to read the most significant word of the operand s . The output produces the least significant word of the result. After this MULT-instruction the most significant word of the result is still stored in the sum- and carry-registers of the multiplier. In order to output this rest of the result we need to execute another LDMUL-instruction. Therefore, the computation of the last example can be coded in three instructions:

LDMUL $X, S_{\text{low}}, \text{don't care};$

MULT $Y, S_{\text{high}}, \text{RES}_{\text{low}};$

LDMUL $0, 0, \text{RES}_{\text{high}};$

Numerous applications in cryptography deal with long operands, e.g. current RSA standards operate with lengths between 1024 and 4096 bits [12]. Such bit lengths cannot be

processed in one operation of a standard processor. One multiplication with large operands can be decomposed into a sequence of smaller multiplications and additions.

Obviously, an $N \times N$ -bit (N even) multiplication $x \cdot y$ can be constructed of four $N/2 \times N/2$ -bit multiplications as follows:

$$x \cdot y = x_{\text{high}} \cdot y_{\text{high}} \cdot 2^N + x_{\text{high}} \cdot y_{\text{low}} \cdot 2^{N/2} + x_{\text{low}} \cdot y_{\text{high}} \cdot 2^{N/2} + x_{\text{low}} \cdot y_{\text{low}}$$

where

$$x = x_{\text{high}} \cdot 2^{N/2} + x_{\text{low}} \quad \text{and} \quad y = y_{\text{high}} \cdot 2^{N/2} + y_{\text{low}}$$

In this case, we need four multiplications of half the size and three additions to get one result of full size. If we recursively use this technique, we can perform an $N \times N$ -bit multiplication by using an n -bit multiplier. It requires $N/n \cdot N/n$ small multiplications and $N^2/n^2 - 1$ additions.

The bit serial structure of the above multiplier allows for a simple and efficient execution of long multiplications: only the operand x is chopped into pieces $X_{N/n-1}, \dots, X_0$ where each piece has the length n of the multiplicand. After loading one of these pieces of x using a LDMUL-instruction the complete factor y is processed by a sequence of MULT-instructions. This takes $N/n + 1$ instruction cycles. The result has the length $N + n$. For the complete long multiplication we need to add one such partial product for each piece of x .

To speed up this procedure, we can take advantage of the multiply-add capability of our multiplier. We start with the least significant word X_0 of x . It is multiplied by the complete operand y leading to a result words $\text{RES}_{N/n} \text{RES}_{N/n-1} \dots \text{RES}_0$. RES_0 is the least significant word of the final result. Afterwards, we proceed with X_1 . Using multiply-add, we compute $X_1 \cdot y + \text{RES}_{N/n} \text{RES}_{N/n-1} \dots \text{RES}_1$. Again we come up with an intermediate result with $N + n$ bits, where the least significant word is part of the final result. In the same way, we continue until we come to the most significant word of x . The result of its multiplication with y delivers the rest of the final result. The following example shows the multiplication of two 2-word operands x and y together with the addition with a third 2-word operand s according to the scheme described above.

```

LDMUL  Xlow, Slow, 0;           //first stage: load xlow

MULT   Ylow, Shigh, RES0;       //compute lower result

MULT   Yhigh, 0, res1;          //compute 2nd partial result

LDMUL  Xhigh, res1, res2;       //first stage: compute 3rd partial result; second
                                     //stage: load xhigh

MULT   Ylow, res2, RES1;        //compute 2nd final result

MULT   Yhigh, 0, RES2;          //compute 3rd final result

LDMUL  0, 0, RES3;              //compute upper final result

```

Note that there are slots left to add 4-word operand; however, an overflow might occur. The programmer would have to take appropriate precautions.

This example can be extended to longer operands as described above. In general, the multiplication of two m -word operands requires $m(m + 1) + 1$ instruction cycles.

Extended Multiplier for Negative Operands

The multiplier of the previous section is capable of multiplying two unsigned integers. With only some minor modifications in the instruction set and in the hardware, it is possible to extend it to a signed multiplier where one operand may be negative in 2-complement representation. We make use of a well known property of 2-complement numbers [2].

PROPOSITION Let x and y be n -bit numbers in 2-complement representation, i.e. $x, y \in [-2^{n-1}, 2^{n-1} - 1]$ and let x' and y' be the values of x and y interpreted as unsigned integers, i.e. $x' = x$ if $x \geq 0$ and $x' = 2^n + x$ if $x < 0$ respectively $y = y'$ if $y \geq 0$ and $y' = 2^n + y$ if $y < 0$. Let $z = x \cdot y$ and $z' = x' \cdot y'$. Then, it holds $z = z' - \text{sign}(y) \cdot 2^n \cdot x' - \text{sign}(x) \cdot 2^n \cdot y' + \text{sign}(y) \cdot \text{sign}(x) \cdot 2^{2n}$, where $\text{sign}(x)$ and $\text{sign}(y)$ denotes the sign-bit of x and y , respectively, i.e. $\text{sign}(x) = 0$ if $x \geq 0$ and $\text{sign}(x) = 1$ if $x < 0$.

Proof We consider the four cases determined by the signs of x and y .

- (i) $x \geq 0, y \geq 0 : z = x \cdot y = x' \cdot y' = z'$.
- (ii) $x \geq 0, y < 0 : z = x' \cdot (y' - 2^n) = x' \cdot y' - 2^n \cdot x' = z' - 2^n \cdot x'$.
- (iii) $x < 0, y \geq 0 : \text{this case is analogue to (ii)}.$
- (iv) $x < 0, y < 0 : z = (x' - 2^n) \cdot (y' - 2^n) = x' \cdot y' - 2^n \cdot x' - 2^n \cdot y' + 2^{2n} = z' - 2^n \cdot x' - 2^n \cdot y' + 2^{2n}.$

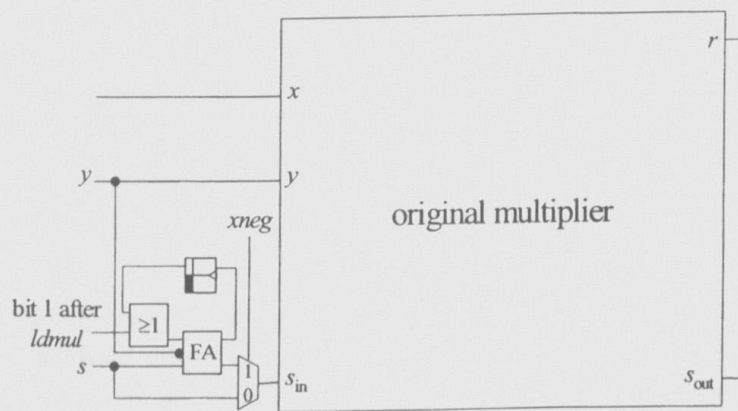
In case (iv) the value 2^{2n} can be omitted, because interpreted as a $2n$ -bit number it is equal to 0. □

In the above proposition z' corresponds to the result of the multiplication of x and y with the unsigned multiplier of "Principle of the ISA" section. The terms $-\text{sign}(y)2^n \cdot x'$ and $-\text{sign}(x) \cdot 2^n \cdot y'$ denote the correction steps associated with negative operands to compute z from z' . In order to extend the original multiplier to deal with 2-complement numbers these correction terms have to be integrated into the new multiplier architecture.

Let us assume that x is negative. According to the proposition $2^n \cdot y'$ has to be subtracted from the result z' . This is equivalent to adding the 2-complement of y to the most significant word of z' (which has the weight 2^n). In the bit-serial case, the 2-complement can be computed on-line with the multiplication by using a multiplexer and a half adder. For the implementation of the multiplication with negative x , we introduce the instruction MULTSX. MULTSX is basically identical to MULT. Additionally, it sets $xneg$ to the sign (stored in the MSB) of x . It is used to select either 0 or $-y$ as additional summand to s . The addition is performed on-line with the input of y and bit-serially delivers $s - y$ at s_{in} . The resulting architecture is shown in Fig. 7.

Floating Point Adder

The floating point adder consists of a shifter, an adder, two units for displaced addressing of the operands, two complement units and a logic for cancellation of operands. The shifter is a configurable shift register of variable length between 0 and 15 bits (see Fig. 8).

FIGURE 7 Multiplier for nonnegative y and arbitrary x .

For a floating-point addition, the result of the exponent comparison is stored in a special register, the shift control register (SCR). The highest bit SCR[11] indicates if the exponent difference is negative and determines the shift operand. Bits SCR[0–3] determine the delay of the shifter. If this is a non-negative number s , it is complemented by the XOR-gates and the last 1-bit delay is activated, in order to achieve the required delay of $16 - s$. Bits SCR[4–5] are used for displaced addressing of the shift operand, as explained in the “Floating Point Operations” section. If any of the bits SCR[6–10] is different from SCR[11], the shift distance would be greater than 64, so the shift operand maybe cancelled. A bit-serial adder adds the shifted operand and the other operand. If any operand is negative and not yet in 2-complement representation, it is complemented before the addition.

PERFORMANCE

For single precision floating-point numbers, an addition takes 14 instruction cycles and a multiplication takes 7 instruction cycles. The multiplication consists of the addition of the exponents and the multiplication and rounding of the significants. The internal format allows us to detect all exceptions (0, infinity, or NaN) by interpretation of the tag-bits during

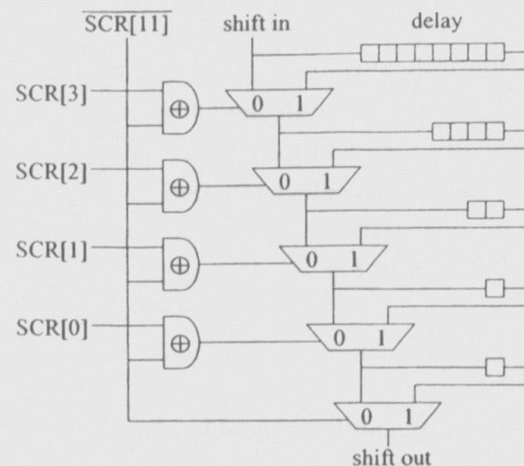


FIGURE 8 16-bit shifter.

TABLE III Multiplication of two single precision numbers with the presented FPU

<i>Instructions</i>				<i>Comments</i>
EXPADD	B[2],	A[2],	P[2];	{add exponents}
LDMUL	0,	A[0],	0;	{load A[10]}
MULZU	B[1],	0,	× [0];	{load A[1]}
LDMUL	× [0],	A[1],	× [1];	
MULZU	B[0],	A[1],	0;	
MULZ	B[1],	0,	P[0];	
LDMUL	0,	0,	P[1];	

The two operands are stored in registers A[0], A[1], A[2] and B[0], B[1], B[2]. The result is written into the registers P[0], P[1], P[2].

the addition of the exponents in the first instruction. A truncated multiplication of 32-bit operands requires six multiplication operations: the first loads the least significant half of the first operand into the multiplier. The second multiplies it with the most significant half of the second operand. The third loads the most significant half of the first operand and generates the most significant 16 bits of the intermediate result. In the remaining three instructions, the intermediate result is added to the product of the most significant word of the first operand and the whole second operand. Table III shows detailed instructions for the multiplication.

The addition is more complicated, since the difference of the exponents provides the shift distance of one operand. It is computed in the first instruction again together with the tags for possible exceptions. Controlled by the shift control register (SCR) the indirect addressing fetches the right fractions of the significands to be added. The leading digit counter (LDC) counts the number of leading 0's in order to detect the final shifting distance for normalization after the addition. Finally, the result is shifted left by this distance, and the exponent of the result is decreased accordingly. Table IV shows details of the addition.

TABLE IV Addition of two single precision numbers with the proposed FPU

<i>Instructions</i>				<i>Comments</i>
EXPSUB	A[2],	B[2],	T[2];	{compare exponents}
SETNF	T[2],	SCR;		{store difference of exponents in SCR register}
NOOP				{wait until SCR is available}
SHADDU	0,	0,	0;	{initialize shifter with 0's and start adder}
SHADDU	A[3],	B[3],	0;	{one adders will be modified, the appropriate operand will be shifted and added}
SHADDZ	A[0],	B[0],	T[0];	{start LDC and add}
SHADD	A[1],	B[1],	T[1];	{add}
EXPSE10	A[2],	B[2],	T[2];	{select the greater exponent}
EXPSUB	A[2],	LDC,	0;	{wait for LDC and adapt exponent: }
EXPSUB	A[2],	LDC,	B[2];	{create exponent of result}
OR	0,	LDC,	SCR;	{initialize SCR}
NRMU	A[3],	0;		{start normalize}
NRM	A[0]	B[0];		{normalize}
NRM	A[1],	B[1];		{normalize}

The two operands are stored in registers A[0], A[1], A[2] and B[0], B[1], B[2]. The result is written into the registers B[0], B[1], B[2].

The performance of one processor is

$$\begin{aligned}
 & 2 \text{ operations}/21 \text{ instruction cycles} \times 1 \text{ instruction cycle}/16 \text{ clock cycles} \\
 & \times f \text{ clock cycles/second} \\
 & = f/168 \text{ operations/second}
 \end{aligned}$$

with $f = 200$ MHz, we have a performance of one processor of 200/168 million floating-point operations/second or 1.2 MFlops; this amounts to 1.2 GFlops for one chip with 1024 processors or to 20 GFlops for a board with 16 chips. For double precision floating-point numbers, an addition takes 18 instruction cycles, a multiplication takes 16 instruction cycles. The performance of one processor is

$$\begin{aligned}
 & 2 \text{ operations}/34 \text{ instruction cycles} \times 1 \text{ instruction cycle}/16 \text{ clock cycles} \\
 & \times f \text{ clock cycles/second} \\
 & = f/272 \text{ operations/second}
 \end{aligned}$$

with $f = 200$ MHz, we have a performance of one processor of 200/272 million floating-point operations/second or 0.735 MFlops; this amounts to 735 MFlops for one chip with 1024 processors or to 12 GFlops for a board with 16 chips.

Also more complex floating-point operations can be computed with the presented FPU, e.g. division, square root. Table V shows the instruction count for several floating-point operations with the presented FPU.

Since instruction count for individual floating operations does not consider format conversion time and communication time within the processor array to fetch the operands, it is often a weak performance measure that does not reflect the behaviour of a complete application. Therefore, we will use the instruction count for two typical floating-point applications on the ISA in our performance evaluation (see Table VI). The two applications are 2-dimensional matrix multiplication and 1-dimensional fast Fourier transform (FFT).

Matrix Multiplication

The problem is to compute the product of two $n \times n$ matrices $F = A \cdot B$, defined as $F_{ij} = \sum_{k=1, \dots, n} A_{ik} \cdot B_{kj}$. We first assume that $n = 32$ (thus, fitting into the processor array). Suppose that matrices A and B have been loaded into the array. Then, the first column of A , which is in the first column of the array, is broadcast along the rows and stored intermediately in some single or double register H . Similarly, the first row of B is broadcast along the columns. Now each processor (i, j) has elements A_{i1} and B_{1j} available and can compute

TABLE V Instruction counts for several floating-point operations in single/double precision

Operation	Single	Double
Format conversion	6	8
Addition/Subtraction	14	18
Multiplication	7	16
Division	52	101
Square root	92	138

TABLE VI Floating-point performance of the 2-dimensional matrix multiplication and 1-dimensional FFT on the ISA processor array of size 32×32

Application	Single (Efficiency)	Double (Efficiency)
Matrix Multiplication (64×64)	0.853 GFlops (71%)	0.522 GFlops (71%)
FFT (2048-points)	0.613 GFlops (51%)	0.427 GFlops (58%)

Efficiency indicates the achieved performance with respect to the theoretical peak performance, i.e. 1.2 GFlops (single) and 0.735 GFlops (double).

$F_{ij} := F_{ij} + A_{il} \cdot B_{lj}$. The next step is to ringshift A by one column to the left in order to get the second column of A , elements A_{i2} , into the first column of the array, from where they can be broadcast along the rows. Similarly, B is ringshifted along the columns. After n such steps, registers F contain the product of A and B . Multiplication of larger matrices can be easily partitioned using 32×32 matrix multiplication as a building block.

Each iteration step consists of two floating-point operations per processor (addition and multiplication) and the ringshift/broadcast communication. The two floating-point operations need 21 instructions for single precision and 34 instructions for double precision. Ringshift/broadcast requires 3 instructions per register on the ISA. Communication costs can be halved by performing a 64×64 matrix multiplication and assigning 2×2 values to each processor. This leads to 30 instructions per iteration step for single precision and 49 instructions per iteration step for double precision, which corresponds to a performance of 0.853 GFlops (single) and 0.522 GFlops (double) receptively. The time required for format conversion can be neglected, since it is only done at the beginning and at the end of the procedure.

FFT

Figure 9 shows the signal flow graph of the FFT of a sequence of n points, here for $n = 8$. It consists of $\log(n)$ stages. In each stage, a complex multiplication with some primitive root of unity, a complex addition, and a complex subtraction is performed at each of the n points. The output of stage k is permuted by interchanging the two halves of subsequences of length

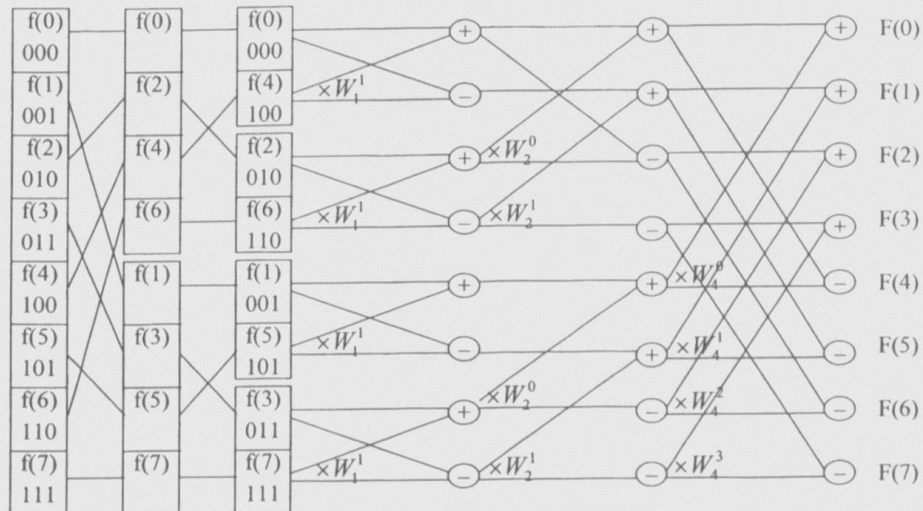


FIGURE 9 Signal flow graph of an 8-point FFT.

2^k before being input to the next stage. Initially, the sequence of the n points is permuted according to the so-called bit reversal permutation.

The implementation of the ID-FFT of size $n = 2048$ on a 32×32 ISA assigns two values to each processor. The computation in each stage consists of four (real) multiplications and six (real) additions in each processor. The implementation of the permutation between every two stages involves routing of data, which is performed either within the rows of the ISA (in the case of interchanging the two halves of subsequences of length ≤ 64) or by interchanging complete rows of the ISA (in the case of interchanging the two halves of longer subsequences). The interchanging of the C -register contents of two halves of m processors is implemented by $m/2$ ringshift operations.

The ten floating-point operations per stage require 112 instructions for single precision and 172 instructions for double precision. The necessary communication per iteration step can be performed on the average within 92 instruction (single) and 153 instruction (double), respectively. This leads to a performance of 0.613 GFlops (single) and 0.427 GFlops (double) for the processor array of size 32×32 .

CONCLUSIONS AND FUTURE WORK

In this paper, the design of a FPU has been presented. Its bit-serial nature allows for an area efficient and energy efficient implementation. Because bit-serial operations with the IEEE floating-point number representation are difficult to implement, we use an internal floating-point format that allows fast LSB-first arithmetic. Our FPU is optimally suited for a fine-grained pipelined architecture. That is why it has been used in the processing elements of the KPROC [6], a parallel system with 1024 processors on a single chip.

Most architectures require fast floating-point computation for any price. In case of a massively parallel system on a single chip, there are other criteria that might be even more important than latency: area and power consumption. In this case, it turns out that the bit serial approach provides significant advantages in comparison to the bit parallel one. Adder and multiplier can be implemented in a more economic way, exceptions can be detected earlier, and rounding and normalization are relatively simple in comparison to the parallel case. The disadvantage of higher number of clock cycles for one operation is compensated by an increased number of processing elements that fit on one chip.

Due to the advances in FPGA technology, it will soon be possible to integrate an array of several hundred simple processors on one single FPGA. An architecture like that could be immediately equipped with floating-point capabilities using the FPU presented in this paper.

References

- [1] (1985) *IEEE standard for binary floating point arithmetic*, ANSI IEEE 754-1985, New York.
- [2] Hennessy, J.L. and Patterson, D.A. (1996) *Computer Architecture: A Quantitative Approach* (Morgan Kaufman Publishers).
- [3] Hillis, D. (1986) *The Connection Machine* (MIT Press, Cambridge, MA).
- [4] Kunde, M., *et al.* (1988) "The instruction systolic array and its relation to other models of parallel computers", *Parallel Computing* **7**, 25–39.
- [5] Lang, H.W. (1986) "The instruction systolic array, a parallel architecture for VLSI", *Integration, the VLSI Journal* **4**, 65–74.

- [6] Lang, H.W., Maaß, R. and Schimmler, M. (1994) "The instruction systolic array—implementation of a low-cost parallel architecture as add-on board for personal computers", *Proceedings of HPCN'94*, LNCS 797 (Springer), pp 487–488.
- [7] Liang, J., Tessier, R. and Mencer, O. (2003) "Floating point unit generation and evaluation for FPGAs", *Proceedings of FCCM'03* (IEEE Computer Society).
- [8] Lienhart, G., Kugel, A. and Männer, R. (2002) "Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulation", *Proceedings of FCCM'02* (IEEE Computer Society).
- [9] Ligon, W.B., McMillan, S., Monn, G., Schoonover, K., Stivers, F. and Underwood, K.D. (1998) "A re-evaluation of the practicality of floating-point operations on FPGAs", *Proceedings of FCCM'98* (IEEE Computer Society).
- [10] Louca, L., Cook, T.A. and Johnson, W.H. (1996) "Implementation of IEEE single precision floating point addition and multiplication on FPGAs", *Proceedings of FCCM'96* (IEEE Computer Society), pp 107–116.
- [11] Meißner, M., Grimm, S., Straßer, W., Packer, J. and Latimer, D. (2001) "Parallel volume rendering on a single chip SIMD architecture", *Proceedings of the IEEE Symposium on Parallel and Large Data Visualization and Graphics* (IEEE Computer Society), pp 107–113.
- [12] Menezes, A.J., van Oorschot, P.C. and Vanstone, S.A. (1997) *Handbook of Applied Cryptography* (CRC Press).
- [13] Schmidt, B., Schröder, H. and Schimmler, M. (2001) "Tomographic image reconstruction on the instruction systolic array", *Computing and Informatics* **20**, 27–42.
- [14] Schmidt, B. and Schimmler, M. (2000) "KPROC—an instruction systolic architecture for parallel prefix applications", *Parallel and Distributed Computing Practices* **3**(2), 163–178.
- [15] Schmidt, B., Schimmler, M. and Schröder, H. (1998) "Long operand arithmetic on instruction systolic computer architectures and its application to RSA cryptography", *Proceedings of Euro-Par'98*, LNCS 1470 (Springer), pp 916–922.
- [16] Schmidt, B. and Schimmler, M. (1999) "A parallel accelerator architecture for multimedia video compression", *Proceedings of Euro-Par'99*, LNCS 1685 (Springer), pp 950–960.
- [17] Schmidt, B., Schröder, H. and Schimmler, M. (2002) "Massively parallel solutions for molecular sequence analysis", *Proceedings of IPDPS'02* (IEEE Computer Society).
- [18] Sim, L.C. and Schröder, H. (2003) "MIMD–SIMD hybrid system: towards a new low-cost parallel system", *Parallel Computing* **29**(1), 21–36.
- [19] Sim, L.C., Schröder, H., Chian, L.C. and Leedham, G. (2003) "Fast solution of large $N \times N$ matrix equations in an MIMD–SIMD hybrid system", *Parallel Computing* **29**(11–12), 1669–1684.
- [20] Taveniku, M., Ahlander, A., Jonsson, M. and Svensson, B. (1998) "The VEGA moderately parallel MIMD, moderately parallel SIMD, architectures for high performance array signal processing", *Proceedings of IPPS/SPDP'98* (IEEE Computer Society), pp 226–232.
- [21] Thiel, T. (1994) "The design of the connection machine", *Design Issues* **10**(1).
- [22] Valafar, F. and Ersoy, O.K. (1995) A parallel implementation of backpropagation neural network on masspar MP-1, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, Athens, GA, November, 158–166.