A New VLSI Vector Arithmetic Coprocessor for the PC

Christoph Baumhof
Institut für Angewandte Mathematik
Universität Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

A new vector arithmetic coprocessor MIM XPA3233 with integrated PCI bus interface has been developed in CMOS VLSI technology. The chip performs dot products of vectors with components of the IEEE DOUBLE *data format to full accuracy or with only one final rounding. Details on the realisation of the multiplication, accumulation and carry resolution processes are discussed. Performance data and some details about the actual VLSI realisation are presented. Software support for the coprocessor is available in the programming languages PASCAL-XSC and C-XSC or from a special C subroutine library. Programming examples are shown using PASCAL-XSC and C.*[†]

1 Introduction

In high-performance numerical and scientific computation the summation of products is a frequent task. The dot product is a basic operation in higher mathematical spaces such as vectors and matrices. In addition, an exact dot product operation $s = a \cdot b = \sum_{i=1}^{n} a_i * b_i$ is an invaluable tool for verified solutions of numerical problems by means of enclosure methods. A uniform mathematical foundation of computer arithmetic and verifying algorithms has been developed by Kulisch [13].

Considering single floating-point operations, the result defined by the IEEE arithmetic standards [1, 2] is the best possible result. However, the composition of a dot product operation from elementary floating-point operations permits accumulation of rounding errors and catastrophic cancellation of leading digits. Cancellation of leading digits is inherent in iterative refinement and defect correction methods.

Using interval arithmetic and an accurate dot product which are available in scientific programming languages like PASCAL-XSC [8] or C-XSC [9], enclosure algorithms have been developed for a variety of numerical problems [6, 12]. IEEE arithmetic supports the basic operations needed for interval arithmetic. This paper describes the design of a scalar product unit (SPU) providing an accurate dot product operation on a VLSI chip for use in standard microprocessor systems. The basic operation principle has been presented in [10].

In PASCAL-XSC and C-XSC, dot product expressions are evaluated using a software library. The SPU has been projected to replace some of these software routines by fast hardware operations. One goal of the VLSI implementation was to achieve the same computational performance of the dot product computation as the i486 microprocessor using its IEEE floating-point instructions would have delivered. To fulfill the requirements of the standard 32 bit PCI bus [15], the chip operates at a clock rate of 33 MHz. This results in a peak performance of 6 MFLOPS. Furthermore, the design was guided by the requirement of a small gate count to make possible a cost-efficient semicustom VLSI implementation.

2 Dot product computation using a Long Accumulator

Many algorithms for the computation of an accurate dot product have been proposed in the past, but the easiest and most flexible algorithm for a hardware implementation is the Long Accumulator (LA). The LA is basically a fixed point register long enough that the entire dot product computation can be executed without error. To achieve this, the LA length must be $L = k + 2e_{\max} + 2|e_{\min}| + 2l$ digits of the base b of the input floating-point format with mantissa length l and exponent range $e_{\min}$ to $e_{\max}$. k additional digits are provided to catch intermediate overflows. After the accumulation, the exact dot product value in the LA is rounded once to the destination floating-point format using one of the four rounding modes described in the IEEE standard.

In the SPU implementation, the IEEE DOUBLE

[†]The research and development project presented in this paper is supported by the Volkswagen-Stiftung, Hannover (Germany).

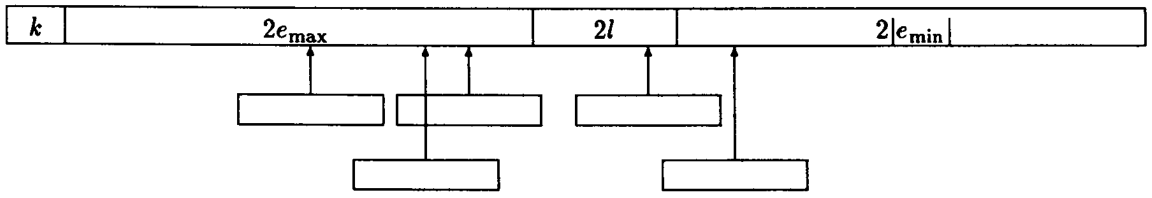

Figure 1: Long Accumulator with double length products

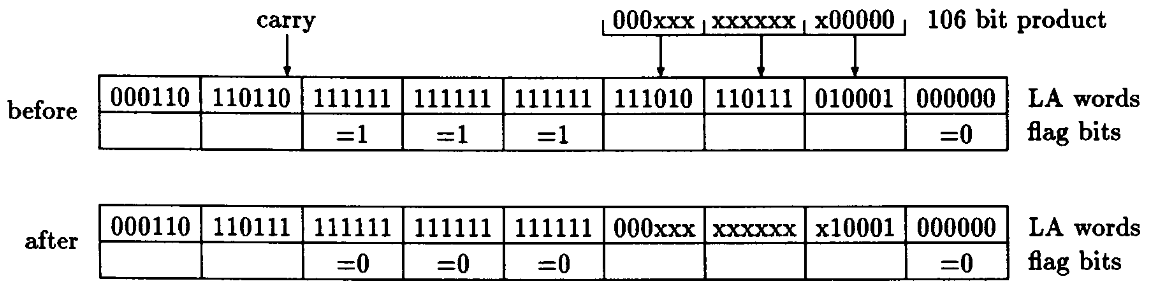

Figure 2: Accumulation and carry resolution

data format is used where $l = 53$ bits, $e_{min} = -1022$ and $e_{max} = 1023$. Choosing $k = 92$ bits, the LA length becomes $L = 4288$ bits. This accumulator is represented in hardware by a 67×64 bit dual ported RAM.

For a dot product computation, the products $a_i * b_i$ are evaluated to double length, i.e. with 106 mantissa bits, aligned to the LA according to their exponents and then added into the LA (fig. 1).

A 106 bit product mantissa touches at most three of the 64 bit LA words. The product addition is thus done in three steps using a 64-out-of-128 bit shifter to extract the matching product part and a 64 bit carry-select adder. The carry handling is done with a block carry-lookahead scheme as proposed in [11, 14]. Each LA word carries two flag bits indicating "all bits are 0" and "all bits are 1" for this LA word. So the address where a carry will be resolved can be generated from the flags when the start address of the additions is known (fig. 2). This is performed in parallel to the three additions.

The flag bits have to be updated on each write access to a LA word to reflect the current state, and at each accumulation if a carry is generated and has to be resolved. In this case, for the LA words that switch from "=1" to "=0" and vice versa, only the flag bits are updated to the new value. So on read accesses to the LA, the flags must be checked first before the RAM contents is read.

For the rounding, the flag bits are used to quickly find the two leading nonzero LA words where the 53 result mantissa bits are extracted, and to indicate if there are bits left in the LA to the right of these two LA words. This information is needed to get the correct rounding in all four rounding modes.

3 Coprocessor architecture

In fig. 3, the block diagram dot product coprocessor is shown. The main components are the PCI bus interface and a 4×64 bit register file for the communication to the main processor, the multiplier, shifter and adder to do the multiplication and accumulation and the LA RAM with the carry resolution logic to hold the dot product value.

I/O-Interface: The SPU uses the standard PCI bus to connect to PC's with Intel i486 or pentium processors. To the main processor, the SPU is a memory-mapped device. This simplifies the integration of the SPU into the system. The processor just performs a sequence of memory read and write instructions which are interpreted and executed by the SPU. For test purposes, a Weitek EMC interface is also included.

Register file: The register file acts as the interface between the 32 bit PCI bus and the 64 bit architecture of the SPU. It consists of four 64 bit words which are seen by the processor as eight 32 bit registers. The register file is used to store the operands and the result of a dot product operation.

Multiplier: The multiplier circuit performs the $53 \times 53 \rightarrow 106$ bit multiplication of the input mantissas. To achieve a minimum gate count, the multiplication is partitioned in four pipeline steps. In each step, a 27×27 bit multiplication is performed, the

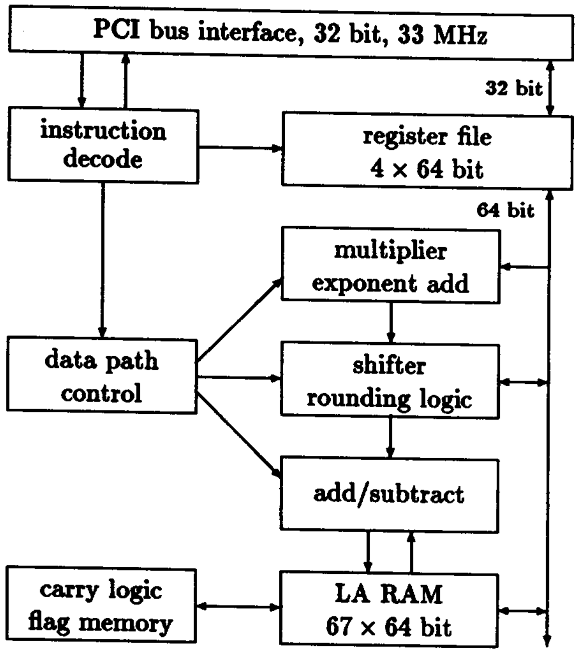

Figure 3: SPU block diagram

four partial products are added in order to get the 106 bit product. The 27 x 27 bit multiplier core uses the Modified-Booth algorithm and a Wallace tree adder structure. The partial product addition is done together with the multiplication steps in the same clock cycles using a 54 bit feedback input into the Wallace tree. Thus the 53 x 53 bit multiplication takes 4 clock cycles total. The addition of the 11 bit exponents is done with a simple ripple-carry adder.

Shifter: The shifter's task is to extract the three parts of the product mantissa for the three additions into the LA. Three steps are performed: an input multiplexer switches the right portions of the product mantissa to the two 64 bit inputs of the barrel shifter. The shift width is generated from the product exponent. The shift is partitioned into a coarse and a fine shift. Each part is built up with tristate inverters which are connected like an eight input multiplexer. The first level shift performs a shift in multiples of eight bits, the second level shift performs the fine shift of 0 to 7 bits.

Long Accumulator: The LA together with the add/subtract unit and the carry logic performs the accumulation of the shifted product mantissa parts as explained in section 2. For the add/subtract unit, a 64 bit carry select adder is used.

Rounding: The rounding of the LA contents to an IEEE DOUBLE number is performed by the same functional units. The flags are used to find the two leading nonzero LA words. In the 64 bit adder, this value is converted from the 2's complement LA representation to the sign-magnitude representation needed for the result. Using a "leading 1" detection circuit, the shifter is used to extract the first 53 bits that form the result mantissa. From the LA address and the shift width, the result exponent is computed. The rounded value is then put into the register file where the main processor picks it up.

Control units: The instruction decode unit and the data path control unit have been designed so that during a dot product computation, the different functional units of the coprocessor operate in parallel. The pipelining can be divided into three parts: operand loading, multiplication and accumulation. All three parts take roughly the same time, so this results in an efficient pipelining. The detailed pipelining of a dot product accumulation is shown in fig. 4.

The main processor does the operand loading (second column). The minimum time per 32 bit transfer is 2 clock cycles. Once the four transfers for the two 64 bit operands of a product accumulation are completed, the coprocessor starts the processing.

In the third column, the operand check and the mantissa multiplication is performed. In two clock cycles, the two 64 bit operands are checked for IEEE exceptional values such as NaN or infinity, in a third clock cycle the necessary action is performed if such an exception is detected. If both operands are valid numbers, the four multiplication cycles follow and the product mantissa is transferred into the shifter input latch.

In the fourth column, the product shift and accumulation are shown. During the three addition cycles, the carry resolve address is computed using the flag values. If the third add generates a carry, the flag bits are updated with their new values and a fourth addition cycle is performed for the carry resolution. If no carry has to be resolved, the carry add cycle is omitted and the "load into shifter" operation in the third column can be performed two clock cycles earlier than indicated.

So, the peak performance of the SPU will be one multiplication and addition every 9 or 11 clock cycles, at a clock rate of 33 MHz this equals 6 MFLOPS. The actual performance will depend on the time the main processor needs to fetch the operands from some operand source and to transfer them to the SPU.

4 Hardware Design

The SPU chip design has been done using the Compass EDA tools. As a first step, a VHDL be-

cycle	read	exception and mult.	shift and accumulate			
1–4	read x_1					
5–8	read y_1					
9–12	read x_2	decode+exceptions$_1$				
13–17	read y_2	multiply$_1$				
18	read $x_{3,L}$	instruction decode$_2$	shift+load$_1$			
19	•	test x_2	add$_1$			
20	read $x_{3,H}$	test y_2	store$_1$	shift+load$_1$		
21	•	test exceptions		add$_1$		
22	read $y_{3,L}$	mult LL$_2$		store$_1$	shift+load$_1$	
23	•	mult LH$_2$			add$_1$	
24	read $y_{3,H}$	mult HL$_2$			store$_1$	update flags
25	•	mult HH$_2$				load cy$_1$
26						add cy$_1$
27						store cy$_1$
28		load into shifter				
29	read $x_{4,L}$	instruction decode$_3$	shift+load$_2$			

Figure 4: Product accumulation pipelining

havioral model of the SPU has been developed. This model has been verified by extensive simulations using the PASCAL-XSC software dot product implementation as a reference. The gate-level description of the control logic has been obtained through logic synthesis of the VHDL code. The data path has been constructed using standard cells and cell generators.

Large and dense layout blocks (e.g. RAM) constitute a routing barrier for the place and route software tools. To improve the routing capabilities of large functional blocks, a layout compiler with stretchable feed-through regions has been written. The improved layout compilers for the dual port RAM and the carry select adders are able to provide additional wiring tracks between the bit slices and compacted building blocks.

The initial placement was done manually for the big blocks like register file, multiplier, shifter, accumulator and rounding. The other cells were placed automatically by the place-and-route tool.

The SPU has been fabricated as a 0.8 μ CMOS sea-of-gates ASIC (Gate Forest [3]). The die size is about 11 x 11 mm^2. On this area, there are about 200 000 active transistors. The largest blocks in the layout are the accumulator memory and carry resolution logic with about 74 000 transistors and the 27 x 27 bit multiplier and partial product summation with about 50 000 transistors.

5 Software support

The PCI interface on the SPU chip provides a memory mapped connection to the main processor. During startup of the system, the SPU requests a memory block of 4K bytes. SPU instructions are encoded into the addresses in this 4KB block of memory, so the main processor simply performs memory read and write instructions to execute SPU commands.

The SPU instructions are basically read and write access to the register file, to an internal status register and to the LA memory, reset the accumulator to zero, add or subtract a product, and rounding of the LA contents. The read and write instructions are used to save and restore the state of the SPU, while the "clear LA", "add product" and "round" instructions are used to perform a dot product computation. To the main processor, these instructions look like memory accesses to different addresses in the 4KB SPU memory window.

In the programming languages PASCAL-XSC and C-XSC, SPU support is directly available by using the appropriate compiler. Whenever an accurate dot product evaluation is requested, the compiler uses the SPU to do the computation. In figure 5, a short program excerpt to compute the dot product of two vectors is shown using PASCAL-XSC in explicit and in operator notation, in C using standard IEEE arithmetic, and in C-XSC. The PASCAL-XSC and C-XSC versions use an accurate dot product evaluation either in software or on the SPU chip.

The first PASCAL-XSC example uses an explicit notation for accurate dot product expressions using the PASCAL-XSC # construct. This tells the compiler to evaluate the expression exactly, the asterisk specifies round-to-nearest for the result. In the sec-

PASCAL-XSC:
```
var a,b: rvector[1..100];
    c: real; i: integer;
:
c := #*( for i := 1 to 100
         sum a[i]*b[i] );
```

PASCAL-XSC:
```
var a,b: rvector[1..100];
    c: real;
:
c := a*b;
```

C:
```
double a[100],b[100],c;
int i;
:
for (c=0,i=0; i<100; i++)
    c += a[i]*b[i];
```

C-XSC:
```
rvector a(100),b(100);
real c;
:
c = a*b;
```

Figure 5: PASCAL-XSC, C-XSC and C dot product computation

ond PASCAL-XSC example, the operator notation is used for the vector dot product. The vector multiplication is a predefined operator in PASCAL-XSC. In both examples, essentially the same code is generated, so there is only one entry in the timing table below.

In the C-XSC example, the predefined vector data type "rvector" is used. The multiplication uses the accurate dot product operation either in software or using the SPU, just like PASCAL-XSC.

The C example shows the traditional way of performing a vector dot product. Here standard IEEE multiply and add operations are used to calculate the result. If n is the vector length, $2n - 1$ roundings are performed during this computation in contrast to one rounding in the other examples using the accurate dot product.

The following table displays execution timings for the four example programs. The indicated time is the time for one calculation of a dot product of the length 100. All times are in ms. The measurements have been done on an Intel i486 PC system using Borland C++ version 3.1 and the PASCAL-XSC and C-XSC versions for this C compiler.

	software	SPU	IEEE
PASCAL-XSC	23.01	0.39	
C-XSC	21.71	0.39	
C			0.38

Table 1: Example execution times

As can be seen, the accurate evaluation via the SPU is about as fast as the (incorrect) evaluation via standard IEEE arithmetic. The software implementations for the accurate dot product in PASCAL-XSC and C-XSC are slower than the SPU by a factor of about 60.

As another example, the verifying linear system solver of PASCAL-XSC has been recompiled to make use of the SPU. In this case, the SPU version of the linear system solver is about 5–7 times faster than the original version using the software dot product of PASCAL-XSC.

6 Conclusion

The SPU chip presented in this paper is a vector arithmetic coprocessor for use in PC systems. For the data format IEEE DOUBLE, it provides the accurate dot product operation and it supports the computation in vector and matrix spaces on the computer. The chip architecture consists of a fixed point summation register, the carry resolution is done in two steps. It connects to the PC via the PCI bus or via the Weitek EMC socket.

The SPU chip is the first hardware implementation of the arithmetic demanded by the GAMM-IMACS "Proposal for Accurate Floating-Point Vector Arithmetic" [5]. It fulfills all the requirements of this proposal. The SPU processes IEEE exceptional values such as infinity or NaN, IEEE traps or interrupts are not supported at this time.

The coprocessor eliminates the speed penalty of existing software algorithms for the computation of an accurate dot product while it is always accurate compared to dot product evaluation via IEEE floating-point arithmetic.

References

[1] American National Standards Institute, Institute of Electrical and Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985, New York, 1985.

[2] American National Standards Institute, Institute of Electrical and Electronics Engineers: *IEEE Standard for Radix-Independent Floating-Point Arithmetic.* ANSI/IEEE Std 854-1987, New York, 1987.

[3] M. Beunder: *The CMOS GATE FOREST: An Efficient and Flexible High-Performance ASIC Design Environment.* In: IEEE Journal of Solid-State Circuits, vol. 23, no. 2, April 1988.

[4] G. Bohlender: *What Do We Need Beyond IEEE Arithmetic?* In: Ch. Ullrich (ed.): Computer Arithmetic and Self-Validation Numerical Methods, Academic Press, New York, 1990.

[5] GAMM-IMACS: *Proposal for Accurate Floating-Point Vector Arithmetic.* In: Mathematics and Computers in Simulation, vol. 35, no. 4, IMACS, 1993.

[6] R. Hammer, M. Hocks, U. Kulisch, D. Ratz: *Numerical Toolbox for Verified Computing. Volume I: Basic Numerical Problems.* Springer-Verlag, New York, Berlin, 1993.

[7] J. Kernhof et al.: *A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic.* 20th European Solid-State Circuits Conference, Ulm, 1994.

[8] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich: *PASCAL-XSC Language Reference with Examples.* Springer-Verlag, New York, Berlin, 1992.

[9] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, A. Wiethoff: *C-XSC. A C++ Class Library for Extended Scientific Computing.* Springer-Verlag, New York, Berlin, 1993.

[10] A. Knöfel: *Fast Hardware Units for the Computation of Accurate Dot Products.* In: Proceedings of the 10th Symposium on Computer Arithmetic, IEEE Computer Society, 1991.

[11] A. Knöfel: *Hardwareentwurf eines Rechenwerkes für semimorphe Skalar- und Vektoroperationen unter Berücksichtigung der Anforderungen verifizierender Algorithmen.* Ph.D. thesis, University of Karlsruhe, 1991.

[12] U. Kulisch, W.L. Miranker: *A New Approach to Scientific Computation.* Academic Press, 1983.

[13] U. Kulisch, W.L. Miranker: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[14] M. Müller, Ch. Rüb, W. Rülling: *Exact Accumulation of Floating-Point Numbers.* In: Proceedings of the 10th Symposium on Computer Arithmetic, IEEE Computer Society, 1991.

[15] PCI Special Interest Group: *PCI Local Bus Specification, Revision 2.0.* April 30, 1993.