# Fast Hardware Units for the Computation of Accurate Dot Products

Andreas Knöfel

Institute for Applied Mathematics
University of Karlsruhe
W-7500 Karlsruhe

## 1. Motivation

Matrix and vector operations based on dot product expressions occur in almost all scientific and engineering applications. The lack of the popular programming languages and computer architectures to provide operators for these data types and corresponding accurate hardware instructions forced users to emulate the vector operations by constructing loops with scalar floating point instructions. Cancellation and immediate rounding in these loops cause uncertain and inaccurate numerical results and aggrevate an error analysis.

The investigation of these effects at the Institute for Applied Mathematics lead to the definition of scalar and vector operations by semimorphism [4], being the basis for a properly defined arithmetic. Operations defined by a semimorphism deliver a result that differs from the correct answer by at most one rounding. The programming languages FORTRAN [10] and PASCAL [11] have been extended for an easy access to numerical data types as well as the corresponding operators with a scalar and vector arithmetic based on semimorphisms. Problem solving routines delivering a verified enclosure of the result have been developed by means of these new programming languages showing the usefulness of the arithmetic. The entire runtime system for all arithmetic operators was written in software due to the uncertain floating point operations delivered by the computer systems. In 1985, the IEEE society defined a standard for scalar floating point operations [9]. Floating point processors based on this standard have been developed for many computers and became faster and faster. The accurate vector operations, however, remained on the software emulation level and their execution time decreased more and more compared with the algorithms using the given scalar arithmetic.

Several accurate hardware vector arithmetic units [2], [3], [7] were proposed in the recent years. For an overview see [1]. They particularly address the vector computer area. For personal computers up to main frames in the following sections two hardware units with high performance, short pipelines and additional hardware less than one multiplier are described.

## 2. The principle of operation

Several principles for the computation of accurate dot products have been developed in the past, but the principle with the most flexibility is the Long Accumulator (LA) [6]. In the LA algorithm the intermediate operations are executed exactly. At first the products are computed to double length. Then they are accumulated into a fixed-point register, called LA. To ensure an exact accumulation at any time, the LA must have

$$S = K + 2E_{max} + 2|E_{min}| + 2l + 1$$

digits of the base b of the input floating point format with l mantissa digits and an exponent between $E_{min}$ and $E_{max}$. K additional digits are for intermediate overflows and one digit is interpreted as the sign of the LA. After the final accumulation the exact dot product value is rounded back into the input floating point format or completely stored for further computations. Figure 1 shows a Long Accumulator. Several products are aligned according their exponent due to accumulation.

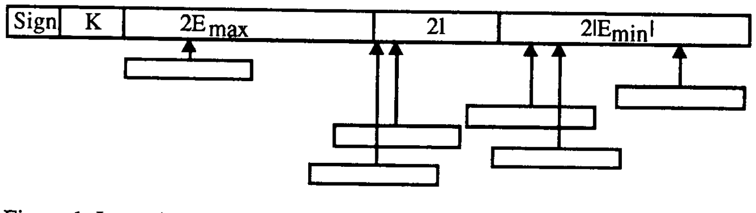


Figure 1: Long Accumulator

In the accumulation step the double length products are shifted to the corresponding position in the LA where they are accumulated. For customary floating point formats like IBM /370 [8], IEEE [9] the products intersect only with 3-5% of the LA. Thus, it is useful to keep the accumulator in a memory and perform only a local accumulation of the product by loading the intersecting accumulator words. After the local accumulation a carry or borrow may propagate over many digits as shown in figure 2.

In software implementations of the LA handling the carry is accumulated in a loop into succeeding words until it is resolved. M. Müller, Ch. Rüb and W. Rülling at the University of Saarbrücken developed a fast solution to accelerate the carry handling [5]. The entire carry processing can be performed by only one addition, since a carry skips an accumulator word only if all digits are equal to (b-1), for b being the base of the number system. After the carry skip, the value of this word is zero. Thus, no addition has to be executed, because the result is predictable. The same effect occurs for borrows. They skip a word only if all digits are zero and after the skip all digits are (b-1). A two bit wide register, keeping two flags, is attached to each accumulator word. One flag indicates, whether all digits of the corresponding LA word are zero and the other one, whether all digits are (b-1). The carry skip over word boundaries is handled by a simple logic that toggles the flags and delivers the address of the accumulator word where the carry is resolved, since one digit is less than (b-1) there. This carry resolve word is incremented in one final addition. The same principle works for the borrows and only the final accumulation has to be performed, as shown in figure 3. The flags have to be actualized after each addition when the word is stored back into memory. Therefore, a word is read from the memory only if both flags are reset. In the other cases a constant is read into the adder.

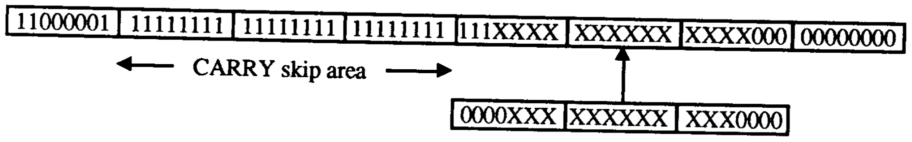
After the accumulation of the last product the result can be read from the accumulator. Here the flags are also important to find the leading accumulator word being the first word whose flag is not equal to the sign of the LA. This leading accumulator word delivers the leading digits and the exponent of the result. As the rounding is concerned it is important if there are trailing digits not equal to zero. This could also be done by a logical OR of all trailing ZERO-flags.

## 3. Hardware description

### 3.1 A sequential circuit

Using a LA stored as a sequence of words with the mentioned flags, a fixed number of accumulation steps is necessary to perform the local accumulation of the product, one step for the carry skip handling and one final carry resolve step. All these steps could be pipelined if the LA is kept in a dual-port-RAM. After the first word, intersecting with the least significant part of the mantissa has been read from the RAM and accumulated, the next word where the next mantissa part must be accumulated can be read from the RAM. The results from the adder and the actualized flags can be written back via the other port of the RAM at the same time. There are no address conflicts because the memory addresses increase and the execution time for the addition lays between the read and the write access to the same address.

It is known in advance, how many accu words intersect with the mantissa. Therefore, the carry skip process and the prediction of the carry resolve address can be performed in parallel to the local accumulation, starting with a fixed offset to the address where the least significant mantissa part is accumulated. At that time it is not known whether a



11000001 11111111 11111111 11111111 111XXXX XXXXXX XXXX000 00000000
CARRY skip area
0000XXX XXXXXX XXX0000

Figure 2: Local accumulation with carry skip

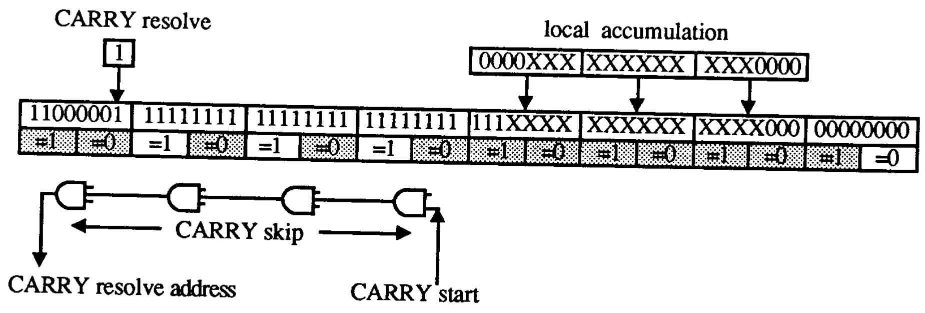


Figure 3: Carry handling with flag logic

final carry or borrow has to be resolved. Therefore, only a copy of the flags is toggled on the way to determine the carry resolve address. To ensure a proper flag setting at any time, the flag values after an addition must be actualized in both the copy and the original flag registers. The final addition can be integrated without any wait-cycles into the read-add-write process, since the carry output of the adder can be added to the carry resolve word in any way. Like in a carry-select-adder the copy of the flags becomes the actual flag setting only if there was a carry after the local mantissa accumulation. This pipelining is shown in figure 4 for a computer with 32 bit data busses, IEEE double input numbers and a 64 bit wide RAM to keep the LA. The accumulation is perfomed in seven cycles within an eight cycle pipeline with the main steps

- operand read,
- multiply & shift and
- accumulate.

In the step "read" the memory accesses and the loop counter and address pointer administration must be handled which could not be performed in eight cycles for many computers. Therefore, eight cycles are a lower bound for the entire pipelining.

Figure 5 shows a block diagram of the circuit. The additional components for the exact accumulation are the RAM with the flags and an address incrementer and a wider shifter and adder. The multiplier must compute the exact product. In almost all computers, however, the lower product part is cut off and ignored. Therefore, the additional amount for the exact multiplication is only a wider output bus. The unit can be designed as a coprocessor or part of a main processor and fits onto one chip being applicable for main frames as well as work stations or PC´s.

| cycles | read | multply & shift | accumulate | | | |
|---|---|---|---|---|---|---|
| 4 | read $x_{i-1}$ | ... | ... | | | |
| 4 | read $y_{i-1}$ | ... | ... | | | |
| 4 | read $x_i$ | $z_{i-1} := x_{i-1} * y_{i-1}$ | ... | | | |
| 4 | read $y_i$ | $z_{i-1} := \text{shift}(z_{i-1})$ | ... | | | |
| 1 | read $x_{i+1}$ | $z_i := x_i * y_i$ | load | | | |
| 1 | | ... | add/sub | load | | |
| 1 | ... | ... | store | add/sub | load | |
| 1 | | ... | | store | add/sub | load carry |
| 1 | read $y_{i+1}$ | $z_i := \text{shift}(z_i)$ | | | store | inc/dec |
| 1 | | ... | | | | store |
| 1 | ... | ... | | | | set flags |
| 1 | | ... | | | | |
| ... | read $x_{i+2}$ | $z_{i+1} := x_{i+1} * y_{i+1}$ | load | | | |
| ... | ... | ... | | | | |

Figure 4: Pipelined accumulation and the overall dot product pipeline
... indicate that the execution is still in process

## 3.2 A parallel accumulation circuit

The progress in hardware design and technology allowed the integration of concepts into computers that seemed to be applicable only for vector processors. Cache memories,wide data busses, RISC instruction sets, parallel execution units and also pipelined floating point operations are the highlights of this new generation of computers. Eight cycles for one pipeline step is too slow for them. But with a few changes in the design the accumulation step can be performed in two cycles and thus may be adapted to the high I/O-bandwidth and fast multiplication times of these computers.

The addresses of all LA-words where an addition must be performed can be predicted. If the accu address is known where the least significant part of the product is accumulated the next part of the product is accumulated to the subsequent accu word and so on. The start address for the carry skip and resolve address prediction has a fixed offset from this start address and therefore the carry resolve address can be delivered in advance. All words influenced may be read in parallel into a sufficiently wide adder and accumulated there in one step as shown in figure 6.

This could only be performed with a multi-port-RAM, but up to six-port-RAMs are state-of-the-art. The width of the LA-words give the designer a parameter to decrease the number of ports down to three. The advantage is that one address decoder manages all ports. The lowest intersecting word is passed always via port 1, it´s successor via port 2 and so on. The drivers of each port must be designed to deliver a value according to the flags. All port drivers and the start signal for the carry skip are controlled by one

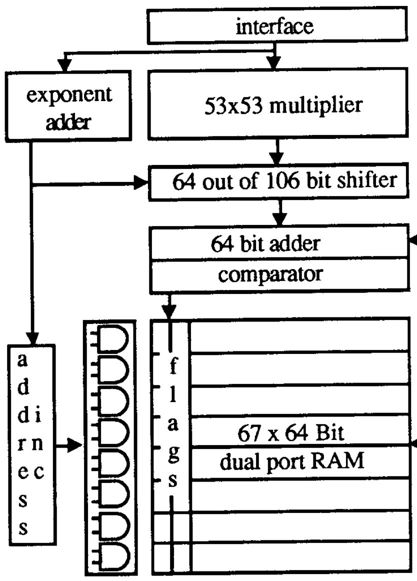


Figure 5: Block diagram

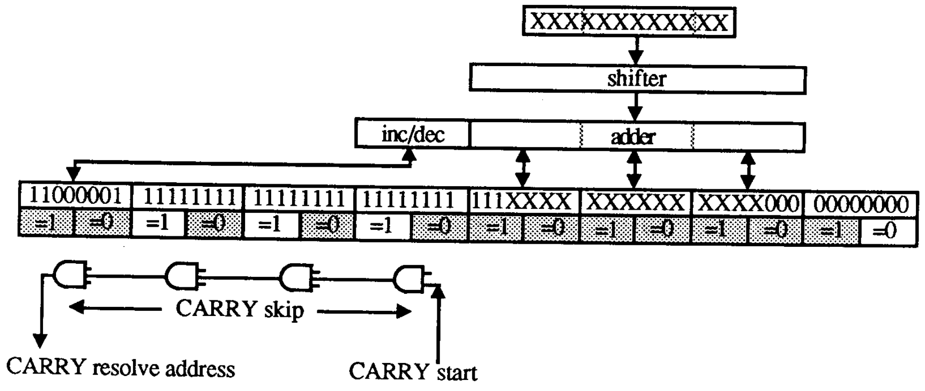


Figure 6: Parallel accumulation

decoder. The carry skip logic determines the carry resolve address, toggles a copy of the flags and then drives the last port where the carry-resolve-word is passed through. This hardwired multi-port driver can be designed very fast so that all words influenced are written to the adder in one cycle. Figure 7 shows the modified block diagram of the RAM.

The entire process with the steps

- decoding of the address
- predicting the carry resolve address and toggle a copy of the flags
- loading the words to the adder
- accumulation
- restore the result and actualize flags

could be performed in two cycles, as shown in figure 8. The accumulation and flag actualization must therefore be performed very fast using sophisticated adders and tree structures in the comparators and decoders. Figure 9 shows a tree structured path to determine the carry resolve word.

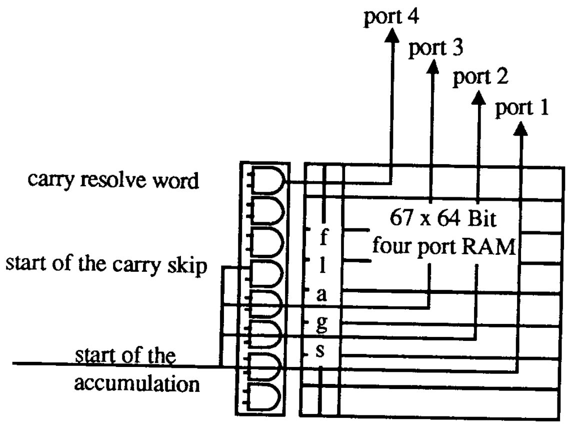


Figure 7: Multi port RAM unit for the parallel accumulation

## 4. Comparison with other circuits

The presented circuits are compared with the proposed solutions of P.R. Cappello andW.L. Miranker [2], R. Kirchner and U. Kulisch [3] and T. Yilmaz, J.F.M. Theeuwen, R.J.W.T. Tangelder and J.A.G. Jess [7]. For all circuits the total amount of transistors has been estimated for the IEEE-DOUBLE format with design data of the Institute for Microelectronics Stuttgart. The result of this estimation and other relevant properties are listed in table 1.

| circuit | # transistors | pipeline length | # cycles accumulation |
|---|---|---|---|
| [2] | 700000 K | ≈ 4200 | 1 |
| [3] | 750 K | ≈ 70 | 1 |
| [7] | 300 K | ≈ 70 | 2 |
| sequential | 200 K | 24 | 8 |
| parallel | 200 K | 6 | 2 |
| multiplier | 80 K | | |

Table 1: Comparison between several circuits.

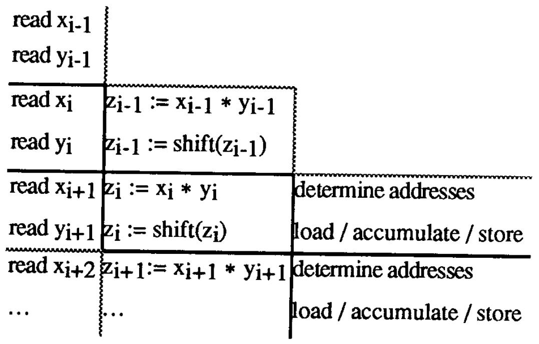

read $x_{i-1}$
read $y_{i-1}$
read $x_i$
$z_{i-1} := x_{i-1} * y_{i-1}$
read $y_i$
$z_{i-1} := \text{shift}(z_{i-1})$
read $x_{i+1}$
$z_i := x_i * y_i$
determine addresses
read $y_{i+1}$
$z_i := \text{shift}(z_i)$
load / accumulate / store
read $x_{i+2}$
$z_{i+1} := x_{i+1} * y_{i+1}$
determine addresses
...
...
load / accumulate / store

Figure 8: The dot product pipeline with the parallel accumulation

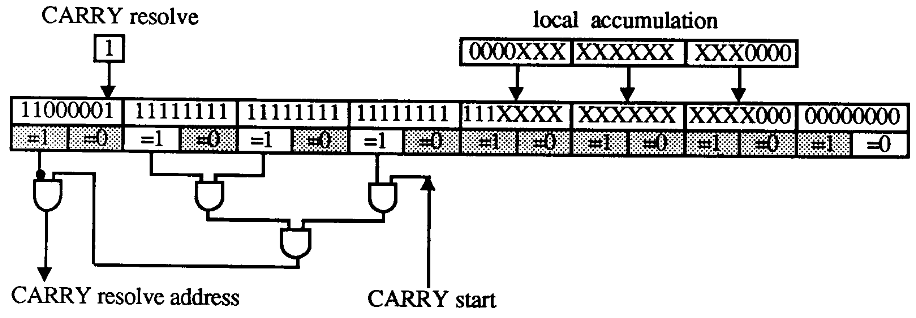


Figure 9: Tree structured carry resolve logic

## 5. Conclusion

The circuits presented in this article are easy to implement with available techniques as one chip and deliver a high performance solution for dot product computations. Beside the RAM for the LA all units can be used for scalar operations to avoid a hardware overhead for scalar and vector units. The additional hardware amount for a combined scalar and vector computation unit is about 120 K transistors and therefore also applicabable for PC´s. The entire data flow from the input interface down to the accumulation has been discussed in contrast to former proposals, where only the accumulation was handled. The pipeline of the first unit has 24 stages and the pipeline of the second unit only 6 stages for the entire process. Thus, these units are also applicable for short vectors and for computations with complex numbers.

## References

[1] G. Bohlender: "What Do We Need Bejond IEEE Arithmetic? ". In: Ullrich (ed): Computer Arithmetic and Self-Validing Numerical Methods, Academic Press, New York, 1990

[2] P.R. Cappello, W.L. Miranker: "Systolic Super Summation". IEEE Transactions on Computers, EC 37-(6), 1988

[3] R. Kirchner, U. Kulisch: "Arithmetic for Vector Processors". Proc. of the 8th Symp. of Computer Arithmetic (ARITH-8), IEEE Computer Society, 1987

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

[5] M. Müller, Ch. Rüb, W. Rülling: "Exact Addition of Floating Point Numbers". Proc. of the 10th Symp. of Computer Arithmetic (ARITH-10), IEEE Computer Society, 1991

[6] S.M. Rump: "Kleine Fehlerschranken bei Matrixproblemen". Ph.D.thesis, Universität Karsruhe, 1980

[7] T. Yilmaz, J.F.M. Theeuwen, R.J.W.T. Tangelder, J.A.G. Jess: "The Design of a Chip for Scientific Computation". Euro ASIC, Grenoble, 1989

[8] International Business Machines Corporation: "IBM System /370: Principles of Operations", IBM GA22-7000

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

[10] International Business Machines Corporation: "IBM High Accuracy Arithmetic - Extended Scientific Computation, Reference", IBM SC33-6462-00

[11] R. Klatte, U. Kulisch, M. Neaga, Ch. Ullrich, D. Ratz: "PASCAL-XSC Sprachbeschreibung und Beispiele". Springer, Heidelberg, 1991