Arithmetic for Relative Accuracy.

R. vanDrunen, L.Spaanenburg, P. Lucassen, J.A.G. Nijhuis and J.T. Udding

Rijksuniversiteit Groningen, Dept. of Computing Science, Groningen (The Netherlands)

Abstract

Of the three factors named in Moore's first Law that drive the advance of computational systems, circuit design receives relatively little mention. We introduce here a circuit variety that allows to include accuracy considerations. It is shown that accuracy–drive can be effectively realised and leads to 60% speed improvement. Details are given of a floating–point unit with full hardware support of complex calculations, specifically tailored to speed–up MD–simulations on the GROMACS scientific parallel computer.

1: Introduction.

Classical computer architecture nomenclature is based on a trade–off in parallelism between instructions (MISD) and data (SIMD). Both categories start from the assumption that all computations execute independent of the data content. Though dyadic operations involving the zero or unity value are commonly exempted, more general accuracy considerations never play a role, largely because of the lack of hardware support. In contrast, it can be recognized that not all subcomputations in polynome evaluation contribute equally to the overall result. As obviously computer capacity may be spent on superfluous calculations, one can expect a considerable improvement in performance from the introduction of circuitry that operates only when numerically sensible.

In the pursuit of higher performance, technology has been a major factor, as any circuit will be faster when fabricated in the next process variety. From a systems perspective, however, performance is relative to the different tasks to be performed, most notably to the *detection* and *recognition*, which can be taken synonymous to respectively *qualification* and *quantification*. For detection it suffices that a phenomenon satisfies a given qualification. Detection may indicate a significant event and should therefore be fast, but not necessarily accurate. The actual nature of the detected condition may be established later on. This recognition of the detected event implies, that the qualified phenomenon is being quantified and such may take some time in order to be accurate.

The different computational requirements in time and accuracy span a *performance window* (Figure 1). Convergence problems and deadlock may cause the design to leave this area; its occurrence must therefore be excluded. Though the advance of microelectronics has made hardware relatively fast and small, larger wordwidth connected with more complex arithmetic operations has somewhat compensated this. In actual fact, the choice has remained between fast and large vs. slow and small (the extreme corners of the performance window), while a number of applications would prefer a selective compromise.

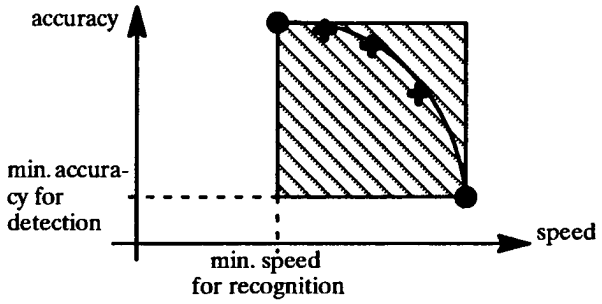


Figure 1 The performance window.

Some typical problem areas, wherein the requirements for detection and recognition are widely differing, are listed in the following:

1. (Collision Avoidance) if an object moves at high speed between obstacles (for instance an autonomous car on a factory floor), it is of utmost importance to note that an obstacle comes within collision distance, and less to know what the exact distance is. Furthermore, the required accuracy is dependent on the actual speed of the vehicle and the relative distance to the object: the closer it gets, the more accurate it has to be and the slower it has to move.
2. (Imaging) if a moving object is displayed on a screen, it is of most importance that the object is displayed. The amount of detail in the displayed picture is of less importance, being dependent on the speed by which the object moves over the screen.

1063-6889/95 $4.00 © 1995 IEEE

3. (Synthetic Aperture Radar, Pulse Compression [1]) where large amounts of integrations are needed on a probabilistic basis, the accuracy of a single computation has less meaning than the mean and variance on the signal ensemble.

These examples show, that in a realistic setting a large amount of data must be manipulated at various data–dependent speed and accuracy restrictions. Such balances between much inaccurate data processed at high speed and less (but more accurate) data at lower speed. Though performance problems can be solved by a next shrink in technology, the simultaneous increase in computational complexity just re–introduces the problem. This calls for appropriate architectures, that are aimed for a large performance window, thereby uncoupling the requirements on detection and recognition.

In a broader sense, the impact of accuracy considerations is to be felt in polynome evaluations, as can be found in a number of computation–intensive simulation tasks such as the Molecular Dynamics (MD) investigation of complex (bio)chemical systems [2]. MD aims to apply Newton's motion equations to a many–body system. The resulting particle trajectories can be analyzed to reveal various physical and chemical properties. Until now, MD simulations are limited to timescales of about 1 nanosecond. This timescale is far too small to observe, e.g., conformational changes in proteins, dynamic phenomena in membranes or reaction sequences in enzymes. As the MD algorithm is well suited for parallelization, the use of a efficient parallel computer seems almost implied.

In Groningen such a computer was built at the University Department of Chemistry. Initially this GROMACS (Groningen Machine for Simulating Chemistry) was designed around a hardware core that performed the inner loop of the force evaluation by means of a huge number of ECL ALUs. In this design the pre– and post processing was concentrated in an array of transputers. At the time where the machine was to be built, general–purpose processors were much more cost–effective than the special–purpose design we had designed. So the next version, that actually got built and used, consists of 32 modules with each an Intel i–860 processor connected in a ring and custom DMA–interfaces. The machine is hosted by a SUN workstation. After years of service, the GROMACS is recently rejuvenated, whereby the processor modules are located two by two on VME–sized boards (Figure 2).

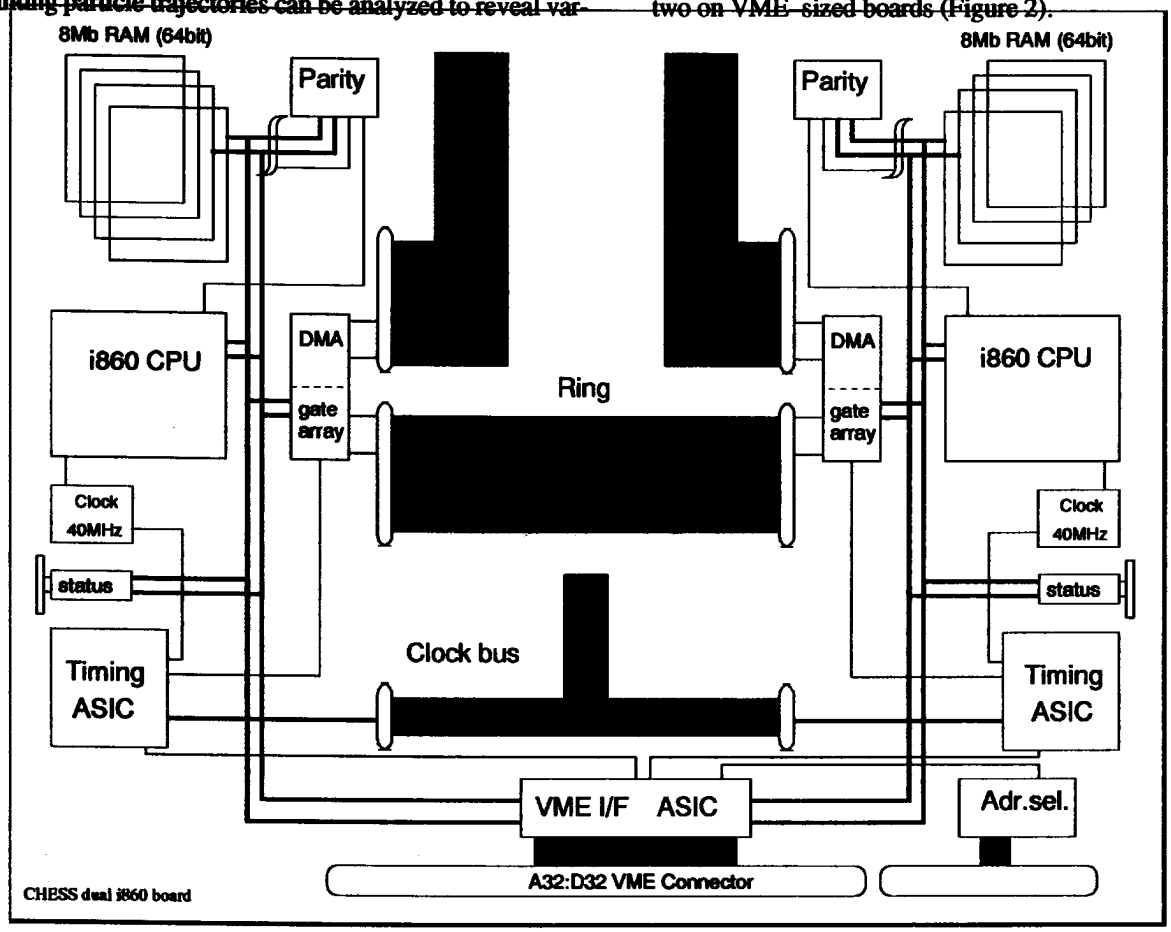


Figure 2 The GROMACS–3 board.

In the past, we have compared the performance of the GROMACS–2 system (a 32 i860–processor machine with GROMACS software) to the performance of a Cray Research Y/MP machine (1 processor running a specially optimized MD code). The simulation performed was a run of 1782 water molecules. Both simulations were done with the same arithmetic–precision in the inner loop. In this typical MD–application a speed–up factor of 8 between the Y/MP and the GROMACS was noted (Y/MP .59 simulation steps per second compared to the GROMACS 4.08 simulation steps per second). The currently tested GROMACS–3 will certainly bring an even larger performance gap, but as simulation time is still measured in days rather than hours the quest for larger improvements is still on.

The description of the molecular forces in the GROMACS program is a model of the real situation. The model is parameterized and fitted to experimental values of the different properties that can be observed when doing MD (such as pressure, temperature, surface tension of liquids etc.). While evaluating the accuracy of the force–field, of the initial conditions of the system and of the experimental values that are used to fit the parameters, it is questioned whether (double) float arithmetic is actually needed in the inner loop of the program. As the hardware currently available does not support reduced accuracy computations, this question seems largely of an academic nature. However, the tendency to answer this question in the negative stimulates an interest in alternative hardware. Though in the past the need for general–purpose hardware has been identified by commercially available processors, the advance of Field–Programmable Gate–Arrays and/or ASIC processor cores provides new roads to innovation.

The long–term goal of the research presented in this paper is therefore to construct a novel arithmetic unit that can do computations in reduced precision and so speed up the process of simulation, while at the same time NOT affecting the overall validity of the resulting physical properties. This reduced–precision arithmetic unit must be constructed to cooperate with the existing i–860 processor or be integrated with suitable ASIC core modules to become a drop–in replacement of the main processor in the GROMACS machine. In the current semiconductor market, the latter approach is not only attractive but has also rapidly become feasible.

This leaves for this paper accuracy–driven arithmetic as central issue. Selective speed–up can be introduced by *local parallellization*. An example is the use of residue number arithmetic, but unfortunately this technique is not very agreeable for detection [3]. An alternative is *on–line enumeration*, where the computation is performed MSB–first and therefore a detection is facilitated at an early stage of the arithmetic process. A popular example in this class is digit–serial arithmetic [4]. This paper introduces a on–line enumeration style based on the use of binary logarithmic arithmetic. In section 2, basic concepts of binary logarithmic arithmetic are stated. Then the benefits are quantified through a study of polynome evaluation. Ensuing we discuss the architecture and implementation of an AD module, by which floating–point arithmetic with reduceable accuracy can be executed in hardware.

2: Arithmetic principles

The multiplication of two numbers A and B reduces to an addition in the logarithmic domain. As an adder is smaller and faster than a multiplier, this seems to be a clear advantage [5]. Lets assume the numbers $A = 2^a$ and $B = 2^b$ defined in a radix–2 logarithmic system. These can simply be multiplied by adding the exponents and taking the antilog of the result: $Z = 2^{a+b}$. Division can be likewise facilitated by subtraction of the exponents. In other words, a multiplication that takes and returns non–logarithmic values needs two tables for conversion to the logarithmic domain, an adder and finally a table for the antilog operation. The logarithm table takes 2^n entries of m–bits, whereas the antilog table takes 2^m entries of n–bits. In order to be more efficient than the Ling multiplier [6], it is mandatory to have $m < n$, which is normally the case. Moreover, where the numbers are constants and/or intermediate results, the tables are not needed and the multiplier becomes a straight adder.

2.1: Logarithmic numbers

Lets detail the logarithmic coding of the numbers A and B into an integer (*characteristic value*) and a fractional (*segment value*) part: $A = 2^{a1+a2}$ and $B = 2^{b1+b2}$, where a1 and b1 are integer and a2 and b2 are fractional parts between 0 and 1.

In the popular *Sign Logarithm Number System*, the data formats are fixed with two separate fields for storing the characteristic value and the segment value respectively. This eases table construction, as now the conversion can be based on two tables of which one is covering the integer value space and the other the fractional value space. For encoding, the table at size 2^n*m can be replaced by a table at size 2^{n1}*m1 and one at size 2^{n2}*m2; for decoding a likewise separation can be accomplished. In the *Level Index Number System* a more generalized view is taken. Here one field for storing the data is taken with a flexible internal boundary, such that when the characteristic value is long it takes less space leaving more room for storing the segment value. This gives it the potential to exploit the dynamic accuracy inherent in binary logarithmic number systems.

Taking the physical limitation of wordsize into account, two effects become noticeable. When the integer field in the data format is not large enough, *overflow* (saturation) occurs. The overflow of a negative exponent indicates a number of which the logarithm will be the same as

that of zero. These numbers are called the *essential zero's* and must be separately handled. Furthermore the number of essential zero's provides a direct indication of the arithmetic accuracy of the system. A related problem is *catastrophic cancellation*, where two almost identical numbers are subtracted and provide the essential zero result due to limited wordlength [7].

When on the other hand the fractional field in the data format is not large enough, value *truncation* (round–off) occurs. Despite the limited range, the accuracy can be enlarged by interpolation. Three basic procedures have been reported in literature [8]: linear interpolation, quadratic interpolation and linear interpolation with non–linear difference.

With *linear interpolation*, the segment value is approximated by ax+b, where the linear coefficients must be set according to an optimization criterium. In [1], 3 optimization criteria are discussed. If the approximation should be as simple as possible, then a=1 and b=0 are the best coefficients. It is shown in [5], that the maximum absolute error is 0.086 per segment. A next possibility is to use the linear term of a Taylor series expansion in a point $x=x_0$. Lastly one may try a further subdivision before applying the piecewise linear approximation. In [1] the optimum values for a and b assuming a 4–segment partition are respectively 1.285610 and 0.006243 in the first segment, 1.050957 and 0.006333 in the second segment, 0.888761 and 0.143537 in the third segment and 0.770244 and 0.231857 in the fourth segment. A maximum absolute error per segment of 0.065 is obtained. It is shown, that a 4–interval partition reduces the error by a factor 6.

In *quadratic interpolation*, one rather takes refuge to a quadratic polynome [9]. An example is the following. Instead of the simple linear approximation of a segment value by ax, we take the more elaborate a*x*(1+*(1–x)/3) to go from conventional binary to logarithmic number representation and a*x*(1+*(1+x)/3) to take the antilog. This reduces the error by a factor 6 and is therefore comparable to the above 4–segment partition.

Under circumstances, we can pre–store the cutpoints of a piece–wise approximation of the quadratic polynome [10]. This leads to the *linear interpolation with non–linear difference*. Assuming the availability of a content–addressable memory such as a PLA, the cut–points can be used as the addresses to generate the exact log–value as stored in the memory structure.

Addition (or subtraction) is less self–evident, as

$$A+B = 2^a+2^b = 2^{a*}(1+2^{b-a}) = 2^{a*}2^d$$

where $d = {}^2\log(1+2^{b-a})$.

In [11], it is suggested to find the value $d = {}^2\log(1+2^{b-a})$ by addressing a look–up table. This of course brings the table reduction problem back again [12]. Several approaches have been used, of which [13] presents a very elegant one. Assume the exponent b–a in the sum formula $d = {}^2\log(1+2^{b-a})$ can be written as s=s1+s2. Then the main conversion error will occur for small s, in which case we simply write $d = {}^2\log(1+2^{s1+s2}) = 2\log(1+2^{s1}) + {}^2\log(1+(2^{s1+s2}-2^{s1})/(1+2^{s1}))$. In order to achieve linear interpolation, the second term has to be approximated into the form $^2\log(1+2^{\delta})$. This is achieved by taking $(2^{s2}-1)*2^s$ for $2^{s1}<.5$, or $(2^{s2}-1)*2^{s1*.5}$ for $0.5<2^{s1}<2$, or $2^{s2}-1$ otherwise. This brings an additional 3.85 bits of accuracy compared to a straight linear interpolation.

2.2: Binary logarithmic

For a discussion on the gate–level, the operations have to be described in the *binary logarithmic system* [5]. Here, the 2–logarithm of a binary number $A = a_n.2^n + a_{n-1}.2^{n-1} + \ldots\ldots + a_0.2^0$ is notated as $^2\log A = j + {}^2\log(1 + A_r/2^j)$, where the *characteristic value* j is defined by ($ja_j=1$ and $a_i=0$ for all $i > j$) and therefore represents the entier of the logarithm $^2\log(a_j.2^j + a_{j-1}.2^{j-1}+\ldots\ldots+a_0.2^0)$, while the *segment value* A_r is the remainder $a_{j-1}.2^{j-1}+\ldots\ldots+a_0.2^0$.

In [14], the DIGILOG hardware model for performing binary logarithmic operations is discussed. It sets out to provide log– and antilog– conversion directly in logic, thereby evading the need for bulky look–up tables. As the conversion logic is small, addition and subtraction can be realized in the binary domain, while multiplication and division will be in the logarithmic domain. As an example, lets look at the multiplication of the two binary numbers A and B, where

$$A=2^j + a_{j-1}.2^{j-1}+\ldots\ldots+a_0.2^0 = 2^j + A_r \text{ and}$$
$$B=2^k + b_{k-1}.2^{k-1}+\ldots\ldots+b_0.2^0 = 2^k + B_r.$$

A straight calculation will provide

$$A*B = 2^{j*}2^k + 2^{j*}B_r + 2^{k*}A_r + A_r*B_r.$$

Upon closer inspection, one finds that the multiplication in the AD representation can be performed by adding the segment values j and k separate from adding the shifted values A_r and B_r, assuming that the productterm A_r*B_r can be neglected.

The multiplication of the binary values 010110 (decimal 22) and 001110 (decimal 14) should result in 000100010000 (decimal 272). In a DIGILOG multiplication with n–bits accuracy, first the leading 1's have to be found. Then of both binary values the n–bits following the leading 1 are taken. The position of the first 1 is the characteristic value and the following n–bits are the segment value in the previous discussion. In the second step of computation the characteristic values and the segment values are shifted and added. Finally the partial results are suitably concatenated, while correcting for the overflow in the segment addition.

In this example a 3–bit accuracy would have sufficed, but a similar conventional computation with 3–bits accuracy would have resulted in 000010000000 (decimal 128),

while the correct result is 000100110100 (decimal 308). This points out the fundamental benefit of the DIGILOG computation: accuracy is maintained dynamically, i.e. with respect to the data value itself and not to its storage requirement. Such basically allows to compute with smaller data registers than in case of a conventional multiplier. In the above example, DIGILOG achieves 88.3% accuracy with 3–bits arithmetic, while conventional binary multiplication delivers 41.5%.

The above formel can be interpreted in line with the usual *Shift–and–Add* (S&A) multiplication, wherein the 1-bits in the value A control the shifting/adding of B. However, by noting that A_r*B_r is again a multiplication, we find a recursion on both A and B with remarkable similarity to LNS multiplication. Moreover, the recursion leads to a successively improving accuracy; hence the name *Accuracy–Driven* (AD) arithmetic [15]. So we can continue by taking A_r and B_r as input and adding to the intermediate result to obtain 000100110000 (decimal 304). Again a restterm is ignored and therefore the recursion can continue. This final recursion then leads to 000100110100 (decimal 308), which is the exact result. In the extreme, the structure will become largely similar to a word–serial multiplication. The important difference is, however, that the result is already more than 12% accurate after one cycle and exponentially improves with every cycle.

Like the multiplication was based on addition, division in the logarithmic number system is based on subtraction. Lets assume a value $A = 2^a$ to be divided by $B = 2^b$. The result can clearly be obtained by subtracting the exponents and taking the antilog of the result: $Z = 2^{a-b}$. As an example, lets look at the division of the two binary numbers A*B and B, where

$$A=2^j + a_{j-1}.2^{j-1}+.....+a_0.2^0 = 2^j + A_r \text{ and}$$
$$B=2^k + b_{k-1}.2^{k-1}+.....+b_0.2^0 = 2^k + B_r.$$

A straight calculation will provide

$$A/B = 2^{j-k} + (A_r - B_r * 2^{j-k}) / B$$

This suggests, that in a AD division the procedure is a neat inverse of the multiplication procedure. The division of the binary value 000100110100 (decimal 308) by 001110 (decimal 14) should result in 000000011111 (decimal 31). In a AD division with 2n–bits accuracy, first the leading 1's have to be found. Then of both binary values the bits following after the leading 1 are taken. The position of the first 1 is the characteristic value and the following bits are the segment value. In the second step of computation the characteristic values and the segment values are subtracted. Finally the partial results are suitably concatenated, while correcting for the borrow in the segment addition.

A similar conventional computation with 6–bits accuracy would have resulted in 000000010010 (decimal 18), while the correct result is 000000010110 (decimal 22). In contrast to multiplication, the accuracy for division is not that high. Binary logarithmic computation shares this problem with conventional floating–point arithmetic.

From the second step, one finds as a result 0100000 (=32). Again a restterm is ignored and therefore the recursion can continue. The final (fourth) recursion then leads to 010110 (=22), which is the exact result. In the extreme, the structure will become largely similar to a word–serial division. The important difference is, however, that the result is already more than 41% accurate after one cycle and exponentially improves with every cycle.

2.3: Impact of accuracy

A recursive relation expresses a dependence of a function on itself. For instance in x=f(a,x) the value of x can not be computed until x=f(a,f(a,x)) is computed and so on. The recursion might be infinite, as for instance in the famous "Tower of Hanoi". If a limit to the nesting can be found, this so–called recursion–depth is bounded. For instance in x=f(a,f(a,f(a,f(a,0)))) the recursion depth is 4. One way to end the recursion is by specifying a limit to the precision, by which the x–value must be calculated.

The way to compute a recursive relation of finite depth can be iterative. Here, each computation delivers new data, to which the relation is again valid. In other words, the recursive relation is unfolded in time and/or space to provide a bounded computational scheme. For instance

$x_0 = f(a,0)$ $x_1 = f(a,x_0)$ $x_2 = f(a,x_1)$ $x = f(a,x_2)$

shows the iterative application of the function f.

In the following, recursive relations are bounded by accuracy constraints and hence iteratively solved. We will therefore largely use the word iterative in this text. As each iteration involves a single shift–and–add operation, we will take it as the basic unit–of–time and assume that a hardware realization will take one machine (i.e. clock) cycle to perform this operation. The simulations can be bounded by limiting the number of iterations (or machine cycles) within a fundamental arithmetic operation (add, subtract, multiply or divide).

Another way to bound the iteration length (i.e. recursion depth) is by limiting the resulting accuracy of a single arithmetic operation. In all simulations, we use a dynamic value storage (i.e. float for single length and double for long length), which gives us a dynamic accuracy.

In the first experiment, numbers from 1 up to 32767 are squared by the AD method and by the usual (S&A) method. The results of this simulation are shown in Table 1. The left column gives the number of iterations necessary to calculate the square of c numbers with an accuracy which is given in the first row. (e.g. 8206 in the second row means: the square results of 8206 out of the 32767 numbers are calculated in 1 iteration with an accuracy of 4%.) In the bottom row the mean number of iterations is provided.

Table 1. Multiplication by the AD and S&A method

AD	4 %	5 %	6 %	7 %
1	8206	9444	10639	11797
2	20478	21102	21696	20970
3	4083	2221	432	0
4	0	0	0	0
$\overline{it}$	1.87	1.78	1.69	1.64

S&A	4 %	5 %	6 %	7 %
1	1374	1732	2101	2476
2	7452	8860	10125	11197
3	13395	14003	14110	14440
4	8940	7317	6262	4654
5	1606	855	169	0
6	0	0	0	0
$\overline{it}$	3.06	2.90	2.76	2.65
c.s	1.63	1.63	1.64	1.62

As is well–known from LNS arithmetic, taking just one one iteration in AD leads to numbers, that are 11.6 % inaccurate [5]. So after multiplying two of those numbers one may expect a 23.2 % error. However, iterating on the restterm rapidly decreases the inaccuracy. As shows from Table 1, 2 cycles already suffices for most of the numbers to reach a better than analog accuracy (i.e. 4%). On the other hand, it goes without saying that to be fully accurate all the 1's in the numbers have to be handled, which takes in the underlying case maximum 15 iterations for the available 15 bits.

We will now perform the same experiment for the usual series/parallel multiplication. These results are also shown in Table 1. The interpretation of the table is the same as above, but an extra row named cs has been added to quantify the different number of iterations. It indicates, that the convergence speed of the AD multiplication is a factor 1.6 to 1.7 higher than by the usual method. This is of course not true anymore, when full accuracy is required, as in both cases all 1's must be handled to obtain a 0% error.

In the foregoing example, we multiplied two equal numbers to circumvent the influence of asymmetry in the partial–product formula. In other words, in the above results only the effect of the improved computational accuracy pro iteration was at stake. Lets now take an asymmetric example. Or more specific, lets multiply 1 with 32767. For reason of the symmetry in the partial–product equation, AD will do this in 1 iteration (after one iteration the multiplicand A=1 will be reduced to A_r=0 and iteration will stop). However, when applying the usual series/parallel multiplication the results in Table 1 still hold. The ratio between AD iterations and Usual iterations will consequently range between 2.65 and 7.5 !!

In almost the same way as AD multiplication is compared to the usual multiplication method, AD division is compared to the usual division method. Only this time the square of the numbers 1 up to 32767 is divided by the number itself. A drastic improvement can not be expected here, as the division would be accurate when computing with full accuracy. Rather will we see the difference between two ways to perform a non–restoring division.

Table 2. Division by the AD and S&A method

AD	4 %	5 %	6 %	7 %
1	2615	3271	3925	4581
2	12411	14257	15795	16680
3	13847	12342	10603	9436
4	3098	2453	2347	2070
5	796	444	97	0
6	0	0	0	0
$\overline{it}$	2.61	2.47	2.36	2.27

S&A	4 %	5 %	6 %	7 %
1	2615	3271	3925	4581
2	9878	11376	12609	13507
3	12938	12857	12216	11916
4	6466	4819	3920	2763
5	870	444	97	0
6	0	0	0	0
$\overline{it}$	2.79	2.63	2.50	2.39
c.s	1.07	1.06	1.06	1.05

In Table 2 the results of the usual division are shown. The number of iterations necessary for AD division is less than for using the usual division. The reason is that AD division can calculate with a negative remainder. The convergence speed is a factor 1.19 better by AD if the needed accuracy is set to 0.01 %. When the needed accuracy is set to 7 % the convergence factor decreases to 1.05.

The difference between AD and S&A division seem to be negligible. Moreover, the technique introduced here as AD division can also be in the usual case, where it is normally called the SRT division. A major difference, however, is buried in the tiny implementation details. Where the usual division is based on a wordwide compare of the two numbers to establish the value for l (the shifting factor for B), the AD division merely aligns the two numbers. The result is a division style that can be implemented using the same hardware as the multiplication [16].

Sofar we looked at divisions, that in principle could give an exact result. The effect of the different ways to creep to the result become more noticeable when the result

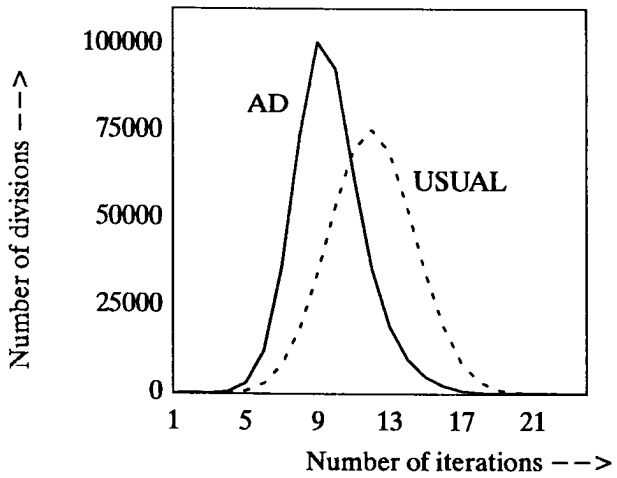


Figure 3. Graphic AD/Usual division.

can never be exact. In Figure 3 we divided again the square of the numbers 1000–10000 to the numbers itself (A^2/A). But this time with a increment of 0.02. In this figure we see clear that the AD technique of dividing is faster than the S&A technique. Accelerations in the order of 30% can apparently be reached.

3: Potential impact

One way to solve the equation $f(x) = 0$ is by iteratively applying the Newton–Raphson formula $x_{k+1} = x_k - f(x_k)/f'(x_k)$. Lets assume a polynomial equation of the nth–order to be written as $f(x) = a_n * x^n + a_{n-1} * x^{n-1} + ... a_1 * x^1 + a_0$, then simply evaluating the function value takes n multiplications. The derivative is slightly more complicated, as it involves a division $f'(x_k) = [f(x_k) - f(x_{k-1})]/[x_k - x_{k-1}]$. In turn, the Newton–Raphson main equation involves a division. Apparently, the iterative equation solving poses a computational problem, involving a number of multiplications and divisions and is therefore very well suited to evaluate the impact of AD arithmetic.

One way to accelerate the Newton–Raphson process is by separating the task of x_{k+1}– and f_k–value enumeration from the task of establishing a derivative. It has been found, that the convergence properties are not damaged by simply not waiting for the correct derivative value to be calculated but rather taking the last computed one. In the following we will evaluate both styles. The former, where all tasks are performed one after the other, will be called the "sequential" style, while the latter, where taking the derivative is handled in parallel with the x– and f–value enumeration is called the "parallel" one. For both arrangements we will apply AD arithmetic or the S&A one.

3.1: Limiting the iterations

For the same accuracy of 24–bits,we found that in the sequential mode of operation the AD method is about 30% faster. We then tried the parallel mode of operation. Compared to the sequential operation, it provides a clear speed–up. Nevertheless, for the same accuracy of 24–bits, the AD method is about 30% faster.

As apparently precision has a significant impact on the number of iterations required to solve a polynomial equation using the Newton–Raphson formula, the question arises whether we can also do with less iterations per single computation. In the next simulation the ability of AD to calculate with less accurate numbers in order to gain speed is tested. Precision will be held at 24–bits, while the maximum number of iterations per computation will be varied from 24 downwards using the parallel mode of operation. As an example we will look in Figure 4 at one solution for the equation $x^3–9x^2–66x+90=0$.

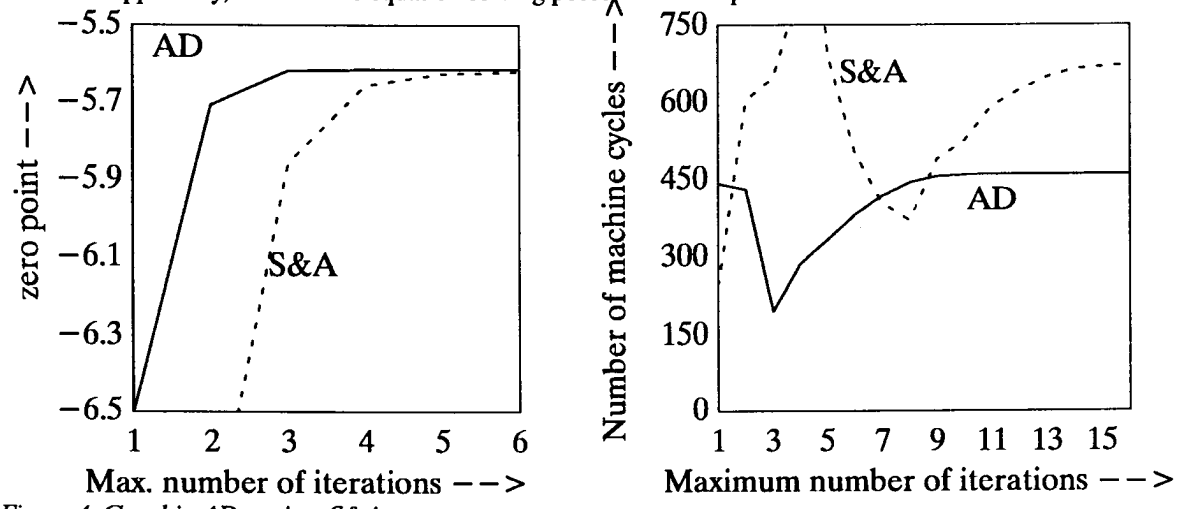


Figure 4. Graphic AD against S&A.

With no iteration limit, AD needs 460 iterations with 24 bits accuracy, while for AD with iteration limit is set to 3 only 195 iterations are necessary. The accuracy, however, is

decreased from about 0.0018% (with no iteration limit) to 0.0527% (with the iteration limit set to 3). But, if we set the iteration limit to 5, AD still needs only 333 iterations and the accuracy has almost not decreased. If we look at these results, we see that, if the iteration limit decreases, the number of iteration decreases also. On the other hand, when the iteration limit is set to 1 or 2, the accuracy is rapidly lost and the AD computation gets into troubles. So, when the iteration limit is set to 3 or 4, we have a good compromise between the accuracy and the number of iterations.

If we now look at the results obtained with the S&A method, we can see that the S&A method has a minimum in the number of iterations when the iteration limit is 8. When the iteration limit is further decreased the S&A method has convergence problems. Much more iterations are needed if we calculate with less accurate numbers. Finally notice that AD converges always while the S&A method does not. So with AD an iteration limit can be set.

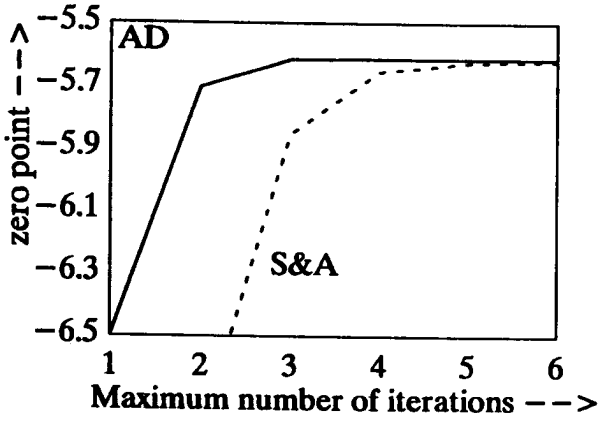


Figure 5. Graphic AD against S&A

3.2: Limiting the wordlength

Next we will again calculate the zero point of an equation with the Newton–Raphson method, but this time we decrease the precision of the multiplier, the divider or both.

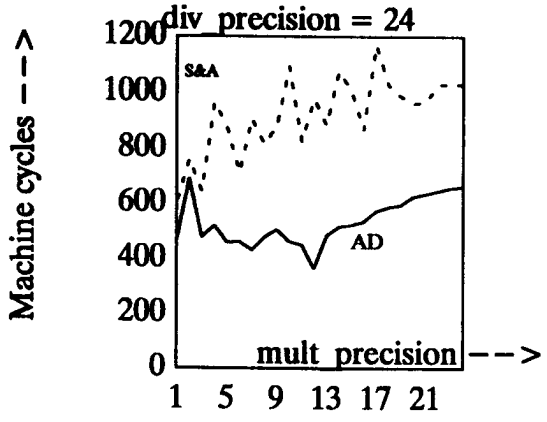
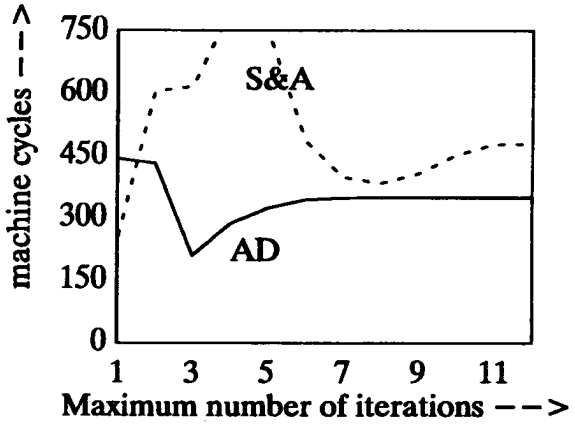


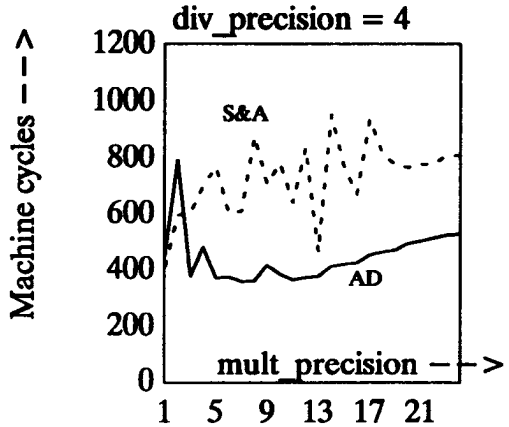
Figure 6 Newton–Raphson with different div_precision.

A related question is, whether the computational precision influences the results. So we repeat the same experiments, taking this time with an arbitrary 14–bits precision and using the sequential mode. As could be expected the impact is significant, but nevertheless the 30% speed benefit of AD for the same accuracy is still present. Again, the parallel operation provides again a clear speed–up. Nevertheless, for the same accuracy of 14–bits the AD method is about 30% faster.

Now, lets reduce the number of iterations per computation. Figure 5 shows that with less iterations, thus with less accuracy, the speed is much higher. Nevertheless AD still gives good results. In contrast, the S&A arithmetic breaks down even earlier and more drastically. The same experiments, but now calculated with an precision of 14 bits, show that consistently the speed is much higher for a shift increase in the final error.



With these simulations we hope to find the ideal combination of precision, error of the answer and maximum number of iterations. For these simulations we use the equation: $x^3 - 9x^2 - 66x + 90 = 0$; the interval is –2, –10. All these simulation use an maximum number of iteration of 24.



In the above figure we can see that the number of machine cycles decreases substantially if we lower the division precision. The solution is almost the same for both division precisions. We can conclude that an division precision of 24 bits for an Newton–Raphson equation solver is not relevant. It costs only more time to reach the zero point. So we set the multiplier precision and changing the division precision of the calculation.

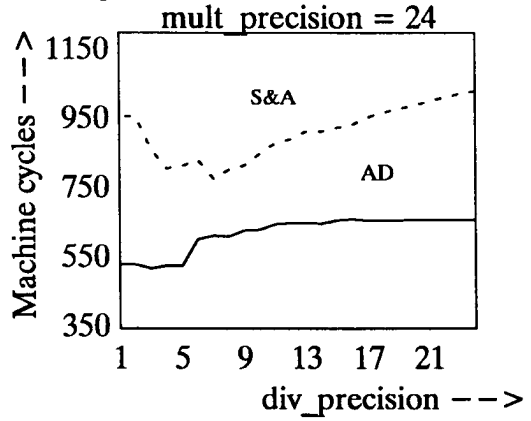


Figure 7 Newton–Raphson with different mult_precision.

3.3: Towards accuracy drive

The last simulations are those with changing precision during the calculation. For these simulations we use again the Newton–Raphson formula For calculation of the solution in the first stage (when f(x)>A) a low precision is enough for the calculation. When A>f(x)>B the precision goes linearly upwards until he reach the maximum precision. After many simulations we found the next ideal function for the precision.

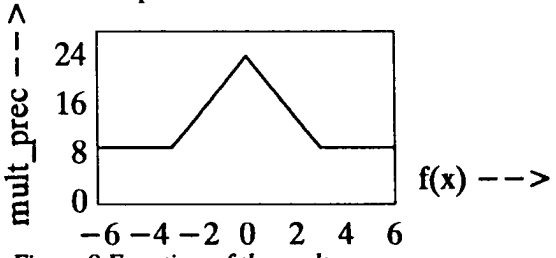


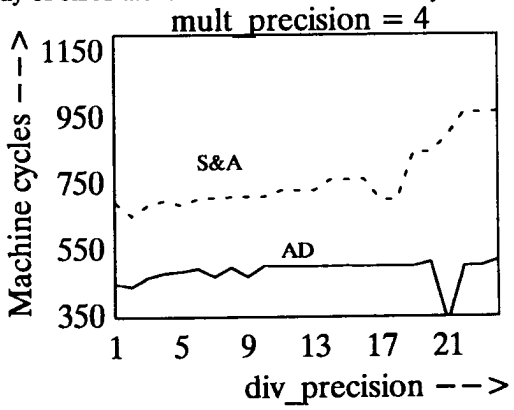
Figure 8 Function of the mult_prec.

We have set the division precision at a value of 2 because previously we found that the precision of the divider had no influence on the answer, only on the number of machine cycles.

We find, that a flexible precision leads faster to a solution of the equation. Let's look at the equation $-2.x^3+33.x^2-17.x-100=0$ in the interval 5, 30 (test3B.

In Figure 9 our expectations are illustrated. First the flexible way makes the most profit. For instance, for test0, the flexible way reaches f(x) = 1 in 400 machine cycles while with the constant precision 750 machine cycles are required. Then (and this is detailed in the second part) the

Figure 7 confirms that the precision of the division has no influence on the solution to the equation. If the multiplication precision is high (24), the answer is fully correct. If the precision is low (4), the answer is not correct anymore. With an equal multiplication precision the number of machine cycles decrease if the division precision also decreased. In this simulation we can see again that the AD way of calculate is faster than the S&A way.



precision gets matched, so that finally from f(x) = 0.1 the both ways are identical. They calculate then both with an precision of 24 bits.

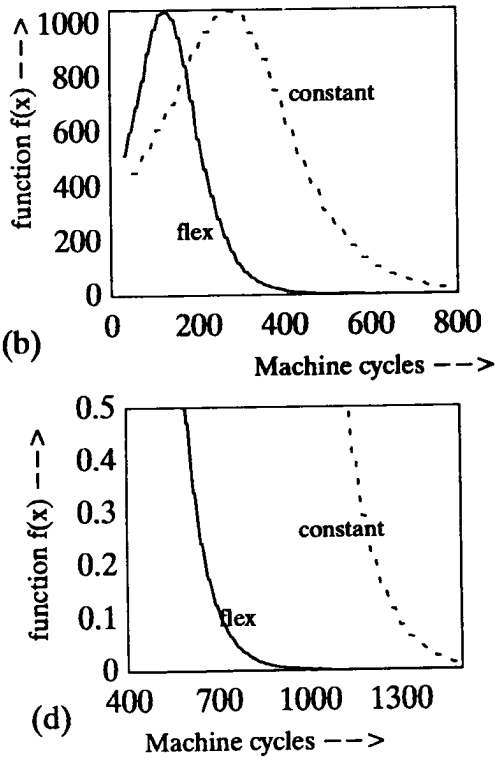


Figure 9 The profit of the flexible precision.

4: Implementation details

The next question concerns the efficiency to implement the above arithmetic. So in the next section we will outline the parts of a single AD arithmetic unit, that can handle the operations addition, subtraction, multiplication, division, power and root for numbers with a large dynamic range.

4.1: Some building blocks

The basic operation in the envisaged module is the parallel detection and subsequent elimination of the leading '1' within a word of arbitrary width. The problem has a more than superficial likeliness with the handling of the carry in digital adder structures. In its most primitive format, the detection of a leading '1' could be handled in ripple mode.

Again, like in the case of the adder, the rippling of the detection signal will lead to ill–determined and probably long propagation times. The optimal way to eliminate the rippling is through a tree–like detection circuit. This promises to bring the delay back from O(n) to $O(^2\log N)$; for instance for a 32–bit word the maximum delay would reduce from 32 time units to 5. The detection tree will have to signal the existence of a leading '1' and to encode its position. We propose here to build the tree from the elementary circuit, sketched in Figure 10. The encoding is valid, if a '1' is detected (*detect* high). In that case, the *ax* and *bx* outputs provide encoded the subscript of the input with the leading '1'. For example if *bit3* is high, then *ax* as well as *bx* are high.

If this circuit is used on the lowest tree level, the bit–inputs are the word–bits of one nibble. On the next level we take the detect–signals as bit–inputs and so on. In this way, we construct 2–bits of the position encoding per tree level. Next to this we have to select the encoding results from the previous level to update the lower encoding bits. To this purpose, we use the circuit as shown in Figure 10. So after two levels we have as encoding: ax bx aX bX, and after three levels we have: ax bx aX1 bX1 aX0 bX0. Beside detection of the leading '1', we also have to eliminate this '1' from the existing word.

We can now put the system parts together by designing the overall control. The operation will proceed as follows. First the words are loaded into the input registers and the output register is initialized. Directly afterwards the leading '1's' are detected, the positions are encoded and saved in the shift latches while the barrell registers are cleared. Ensuing, in five consecutive steps, the words are shifted while simultaneously the inputwords are stripped of their leading '1'. Then the results are offered to the adder, and the above sequence is repeated for the reduced input-

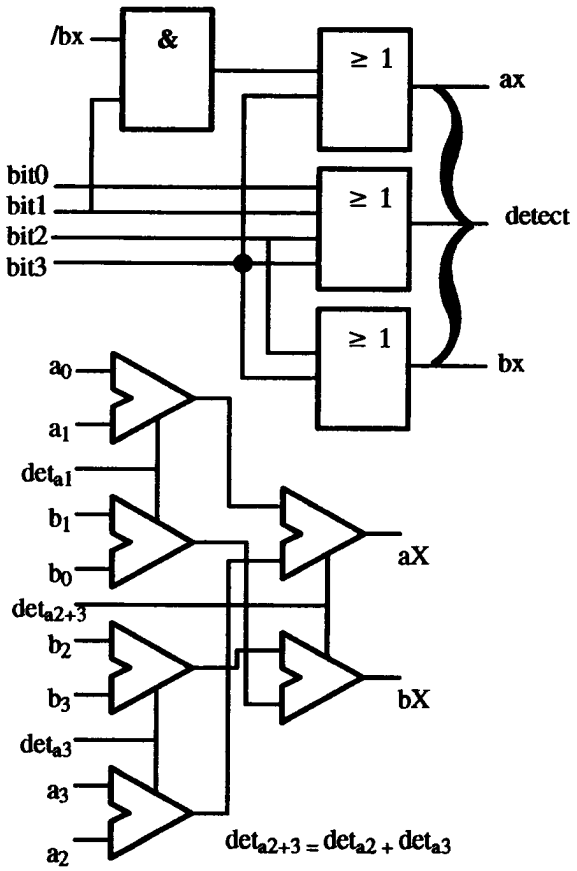


Figure 10. Encoding the leading '1' inside a nibble.

words if the stripped words are non–zero. A second stage may be needed to add this to the intermediate result. The architecture can be condensed by using a single adder with 3 simultaneous inputs, but in the schematic we will still use 2 adder stages for the sake of clarity. To support the potential speed of operation, the structure is minutely pipelined with an internal communication scheme based on "data valid" signals.

4.2: Module architecture

Sofar we has largely concentrated on the multiply function of the proposed arithmetic module. The next step towards integration is to enlarge its applicability to support a wide range of arithmetic operations. The register transfer model of the complete module is depicted in Figure 11. The thick black boxes indicate the registers, the thin black boxes the multiplexers. The triangles D denote the detection of leading 1's, while the elimination of the leading 1's is performed in the traingles E. Not shown in the diagram is the sign manipulation, while the handling of zero values requires no exemptions but is implied in the structure.

To perform addition and/or subtraction, the first half of the module can be skipped but the pipelining sequence

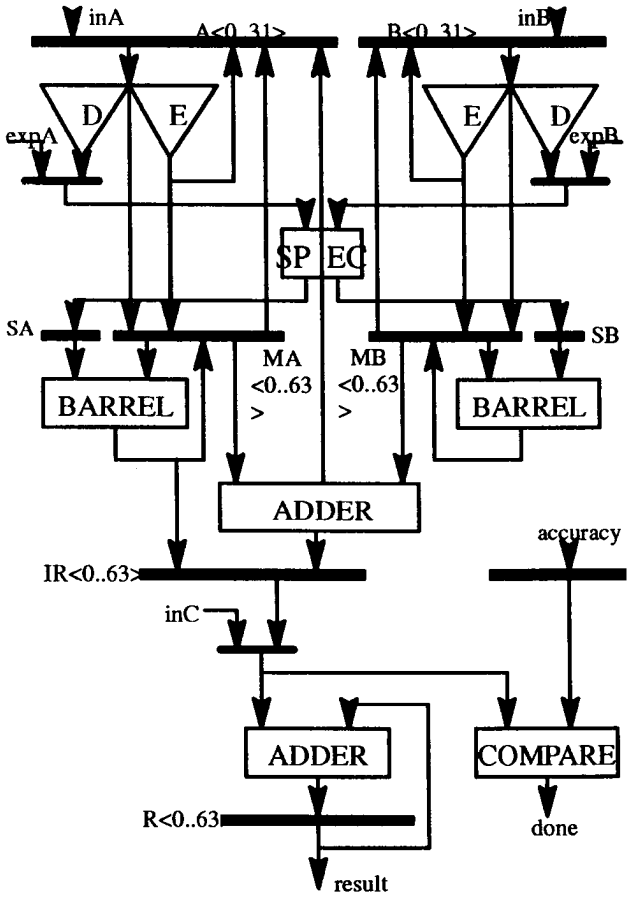


Figure 11. Module register transfer model

must be maintained. For example, addition is performed as:

```
IntAdd: A←inA                    B←inB
        MA←A                     MB←B
        IR←MA+MB
        R←IR
```

Evidently, a larger part of the module will be used for multiplication and/or division. For example, multiplication is performed as:

```
IntMult: A←inA                   B←inB
         MA=E(A), SA←D(B)        MB=B, SB←D(A)
         MA←shift(MA,SA)         MB←shift(MB,SB)
         IR←MA+MB
         R←+ IR
```

As we perform these integer operations with relative accuracy, some degree of floating–point operation is already implied. As an example, the single module floating–point addition is as follows:

```
FloatAdd: A←inA                  B←inB
          SA←expA, MA=A          SB←expB, MB=B
          MA←shift(MA,SA)        MB←shift(MB,SB)
          IR←MA+MB
          R←IR
```

It is clear that this is merely a float–to–fix conversion followed by an integer add. However, the module is already rich enough to support real floating–point arithmetic by the use of two of such integer modules. Lets look at the multiplication of two floats. This is composed of two steps: (a) the addition of the exponents together with the multiplication of the mantissae, followed by (b) the handling of the overflow from the mantissapart by the exponentpart. In contrast to conventional floating–point units, we do not need a separate addition and multiplication part here.

Lets look at the addition of two floats. This is composed of three steps: the alignment of the mantissae, the addition of the mantissae and lastly the handling of the overflow from the addition into the exponent. Using two AD modules this leads to: subtraction of the exponents and storage of the results in a segment register of the mantissa module, barrel shift to replace one mantissa, addition of the mantissae, and handling of the overflow. On the one hand, this looks relatively complicated; on the other hand it simply uses the barrel shifter that is already present for other purposes.

Overall we have implemented the following set of instructions on a single structure:

- the classical arithmetic operations on the A– and B–ports (addition, subtraction, multiplication and division).
- the cooperation over the segment registers towards the mantissa–oriented functionality; instead of a leading '1' detection on the dataport to set the segment register, the segment register is loaded directly and the dataword is shifted according to the new content (addition, subtraction, log, normalisation)
- the classical arithmetic operations on the segment registers; instead of the data on the A– and B–ports, the segments registers are handled and the result is directly outputted to the neighbouring module (addition, subtraction).
- the retro–functionality of the segment registers towards the main dataports; the segment registers are externally loaded, while the A– and B–ports are first defaulted with a single '1'. The result of shifting is then fed back to the A– and B–ports (antilog).

The resulting structure is pipelined at the micro–level, which also allows for partially overlapping instructions, further boosting the basic speed. The current 32–bits design takes about 3000 gates on an FPGA and is prepared for further optimization as a silicon module. From previous (par-

tial) integrations, we expect a final area consumption per module of 2.5 mm^2 in a 0.8 µm CMOS technology running at 100 MHz.

4.3: ... and its use.

From the basic characteristics of the arithmetic technique, the extensionsion towards accuracy–drive was easy to achieve (Figure 11). The results are either produced in a single path or iteratively in steadily decreasing contributions. By simply guarding the contributions in each path through the calculation, the increase in accuracy can be monitored. A prerequisite for this measure is the fact that the contributions becomes steadily smaller. As discussed before, this is a basic characteristic of AD arithmetic.

A short look at the architecture of the INTEL i860 (as is currently the basis for the GROMACS machine) shows one of the primary problems with conventional floating–point arithmetic: next to the multiplier, a separate adder/ subtractor is required. A further inspection brings to bear that next to multiplication no other complex arithmetic calculation is implemented in hardware. Division is microprogrammed; power and root are ignored. The impact of introducing the AD–module will therefore go beyond the support of accuracy considerations. It will also bring a distinct acceleration for accuracte computation. Moreover, it allows to reduce the module count: the floating–point adder has become superfluous (and we do not need the graphic unit). This reduces the implementation by almost 25%.

More uniqueness is present in its application to provide a learning capability to a 8051–core when acting as a neurocontroler for real–time intelligent operation. The handling of measurement data without preprocessing to eliminate non–repeatable data normally withstands in–line learning. The use of a neural network emulator, wherein the learning accuracy can be changed from coarse in the initial phase to fine in the later stages provides an efficient means to suppress deviations in the input data until the learning has progressed far enough to take them into consideration.

These two examples of ongoing system development illustrate the pursued flexibility in balancing accuracy for speed. It also illustrates some of the advantages to be gained when accuracy conflicts can be resolved before dead–lock and/or convergence failure sets in.

Acknowledgements

The initial phase of this project ran under the DFG Schwerpunktprogram 322699 "System– und Schaltungstechnik für hochgradige Parallelverarbeitung". Furthermore we like to thank M.H.M. Luft, P.J.H. Speckreijse, H. van Aartsen, R. Schukken,, G. Poppinga and especially M. Diepenhorst for their cooperation. Lastly our gratitude goes to Chess Engineering in Haarlem (The Netherlands).

References

[1] E.L. Hall, D.D. Lynch, and S.J. Dwyer III, "Generation of products and quotients using approximate binary logarithms for digital filtering applications", *IEEE Transactions on Computers*, Vol. C–19, No. 2, pp. 97 – 105, February 1970.

[2] H.Bekker et al., "GROMACS: A parallel computer for molecular dynamics simulation", (in: *Digest Conference on Physics Computing*), 1992.

[3] E.E. Swartzlander et al., "Sign/Logarithm Arithmetic for FFT implementation", *IEEE Transactions on Computers*, Vol. C–32, No. 6, pp. 526 – 534, June 1983.

[4] M.J. Irwin and R.M. Owens, "Digit–pipelined arithmetic as illustrated by the Paste–Up system: A tutorial", *IEEE Computer*, Vol.20, pp.61–73, April 1987.

[5] J.N. Mitchell jr., "Computer multiplication and division using binary logarithms", *IRE Transactions on Electronic Computers*, Vol. 11 , pp. 512 – 517, August 1962 .

[6] H. Ling, "An approach to implementing multiplication with small tables", *IEEE Transactions on Computers*, Vol. C–39, No. 5, pp. 717 – 718, May 1990.

[7] M.G. Arnold et al., "Redundant logarithmic arithmetic", *IEEE Transactions on Computers*, Vol. C–39, No. 8, pp. 1077 – 1086, August 1990.

[8] M. Combet, H. vanZonneveld, and L. Verbeek, "Computation of the base–2 logarithm of binary numbers", *IEEE Transactions on Electronic Computers*, Vol. 14, No. 6, pp. 863 – 867, December 1965.

[9] D. Marino, "New algorithms for the approximate evaluation in hardware of binary logarithms and elementary functions", *IEEE Transactions on Computers (Short Notes)*, Vol. C–21 , No. xxx, pp. 1416 – 1421, December 1972.

[10] H.–Y. Lo, and Y. Aoki, "Generation of a precise binary logarithm with difference grouping programmable logic array", *IEEE Transactions on Computers*, Vol. C–34, No. 8, pp. 681 – 691, August 1985.

[11] N.G. Kingsbury, and P.J.W. Rayner, "Digital filtering using logarithmic arithmetic", *Electronic Letters*, Vol. 7, No. 2, pp. 56 – 58, 28th January 1971.

[12] F.J. Taylor etal., "A 20–bit logarithmic number system processor", *IEEE Transactions on Computers*, Vol. C–37, No. 2, pp. 190 – 199, February 1988.

[13] F.J. Taylor, "An extended precision logarithmic number system", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP–31, No. 1, pp. 232 – 234, February 1983.

[14] B. Hoefflinger, M. Selzer, and F. Warkowski, "Digital logarithmic CMOS multiplier for very high–speed signal processing", (in: *Digest Custom Integrated Circuit Conference*), San Diego, CA, pp. 16.7.1 – 16.7.5, May 1991.

[15] L. Spaanenburg, A. Siggelkow, and M. Luft "An arithmetic technique for accuracy–driven VLSI systems, *Digest ECCTD*, pp. 161 – 166, Davos, September 1993.

[16] L. Spaanenburg et al., "An accuracy–driven complex arithmetic unit", EuroMicro'94, pp. 491–498, Liverpool (U.K.) September 1994.

[17] F.J. Taylor, "A hybrid floating–point logarithmic number system processor", *IEEE Transactions on Circuits and Systems*, Vol. 32, No. 1, pp. 92 – 95, January 1985.