Asynchronous Arithmetic Algorithms for Data-Driven Machines*

Carol L. Bridge, P. David Fisher, and Robert G. Reynolds%

Michigan State University; E. Lansing, Michigan 48824

ABSTRACT

With data-driven machines a statement associated with a given processing element fires the moment its input operands become available.[1] In order to take full advantage of this computer structure and achieve maximum throughput, the processing elements themselves should also be asynchronous; i.e., instruction execution times should be data dependent to minimize overall delays. Five general procedures are described that may be used to design self-timing processing units: task completion prediction, task completion detection, operand preprocessing, pre-estimation of input operand values, and significance control. Analysis and simulations suggest that the greatest potential for speed improvement over synchronous counterparts comes with self-timing algorithms for both division and the evaluation of special functions.

INTRODUCTION

A paradox exists with instruction-driven machines: While performing a scientific or technical computation, the machine may be compute bound, yet its arithmetic unit may be idle a significant percentage of the time. This paradox is due in part to the sequential nature of the arithmetic algorithms executed at the operation level. One approach to solve this shortcoming is through the application of data-driven machines in which asynchronous arithmetic units execute data-dependent algorithms.

To understand how asynchronous arithmetic units (AAUs) improve data-driven machine performance consider the example in Figure 1a. Nodes and arcs in the graph represent processing elements (PEs) and data paths, respectively. Tokens may be placed on the arcs to denote the instantaneous status of the machine with the presence of a token indicating that data are available on that data path. So, the current state of the machine illustrated in Figure 1b, as indicated by the token placement, is: division has been completed, multiplication is in progress, and the summation unit is waiting for its second operand. Instructions are executed when the necessary data operands are available; there is no need for a system clock to synchronize either inter or intra-processing element operations. Consequently, throughput may be improved by using asynchronous rather than synchronous arithmetic units in the PEs.

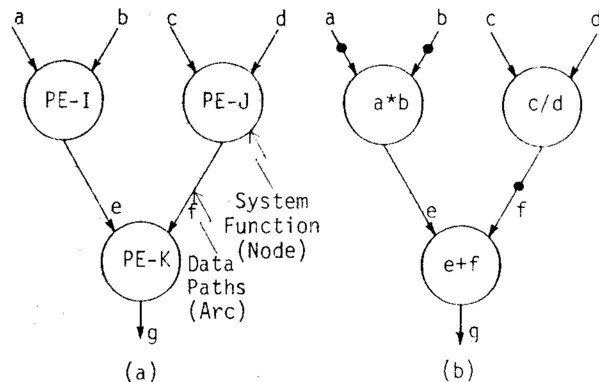

Figure 1. (a) Graph illustrating the interconnection of three processing elements, PE-I, PE-J, and PE-K. In (b) these three processing elements are programmed to perform multiplication, division, and addition, respectively, to yield g = (a*b + c/d). The solid circles on the arcs indicate the presence of valid data on the data paths. PE's need not be limited to elementary arithmetic operations; moreover, they may be dynamically reprogrammed to execute new operations.

This paper investigates the potential of using asynchronous instead of sequential arithmetic units as processing elements in data flow machines. First, we present an asynchronous model of computation for these processing elements. Then we consider specific methodologies for implementing both standard arithmetic operations and special arithmetic functions in asynchronous arithmetic units.

---

*This research was supported in part by the National Science Foundation under Grant No. MCS79-09216.

%The first and second authors are in the Department of Electrical Engineering and Systems Science; the third is in the Department of Computer Science.

CH1630-3/81/0000/0056$00.75 © 1981 IEEE

AAU COMPUTATIONAL MODEL

Three axioms establish the general performance characteristics of the asynchronous arithmetic unit (AAU).

* Axiom I -- An activity commences without delay within an idle AAU whenever the input information necessary becomes available to it.

* Axiom II -- When an activity is completed within an AAU, the results are immediately made available as inputs to downstream PEs. Then the AAU goes into the idle state unless new input operands are available.

* Axiom III -- AAU latency must be minimized for a given set of standard hardware constraints, e.g., power dissipation limits, chip area conservation requirements, and inherent gate propagation delays. By latency we mean the time interval between the moments when an AAU activity commences and when the AAU returns to the idle state.

Based upon these three axioms, five general procedures may be specified which lead to improved PE throughput. Each of these procedures takes advantage of input or output data dependencies; these procedures are illustrated in Figure 2 and described below:

* Procedure A -- monitors the current activity in the system function unit (SFU) and detects when the activity is complete, i.e., the SFU is no longer busy. The SFU performs the required arithmetic operation as part of the PE's hardware structure.

* Procedure B -- predicts concurrently when the system function unit activity will be completed while the system function unit is busy operating on input data.

* Procedure C -- uses the previous data operands--a(k-1), b(k-1) and c(k-1)--as an initial estimate to the system function unit of the next operands--a(k), b(k) and c(k).

* Procedure D -- preprocesses one or more of the input data operands before they are made available to the system function unit. In Figure 2, a and b are transformed into a* and b*, respectively.

* Procedure E -- utilizes a significance control output data path to control the number of significant digits generated within the system function unit.

The procedures are not all independent since they must often be applied in combination. The application and general usefulness of these five procedures are explored in the next sections. The resultant algorithms are compared with conventional data independent algorithms. For purposes of our discussion, we will not consider either inter-system-function control or system function programming.

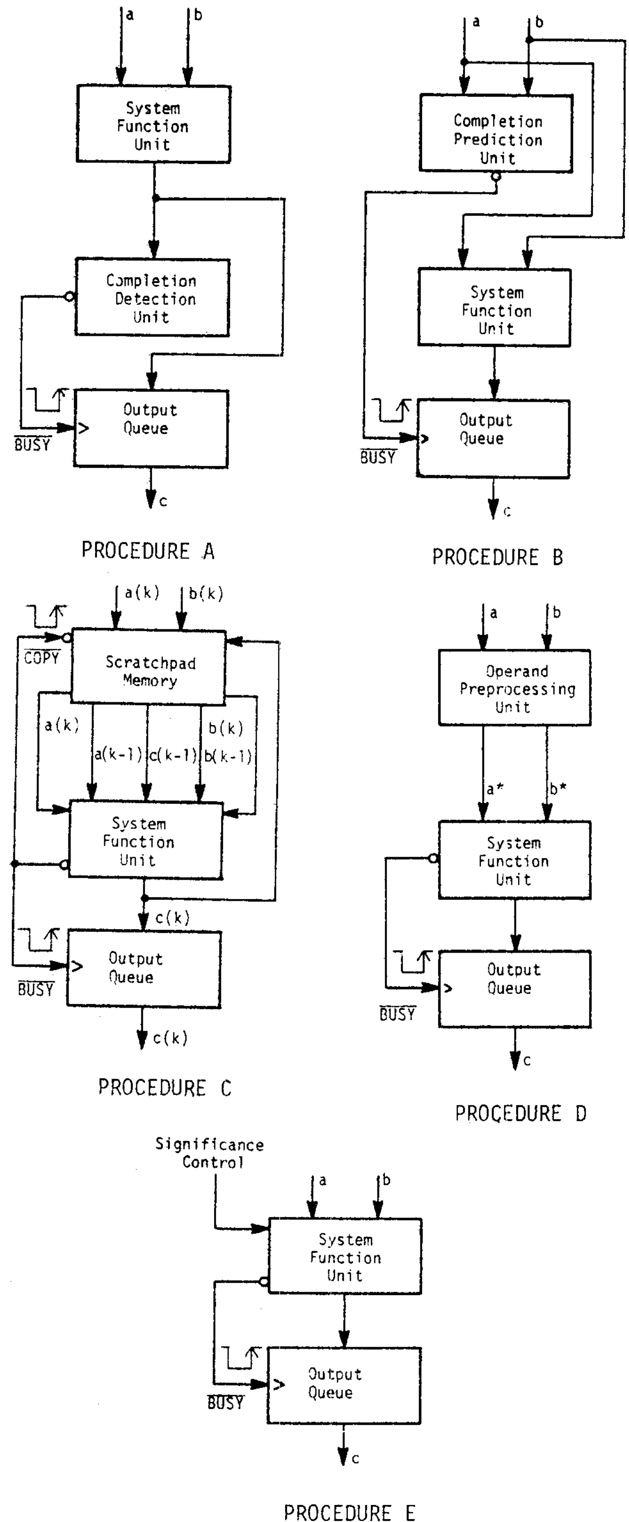

Figure 2. Block diagram of five data-dependent procedures that may be employed to reduce the latency in asynchronous arithmetic units (AAUs).

ADDITION, SUBTRACTION AND MULTIPLICATION

Data-dependent algorithms for FXP addition and subtraction can be considered simultaneously, except for one algorithm refinement that is used when the subtrahend arrives before the minuend. For this special case, it is possible to use Procedure D to process the subtrahend. Specifically, the subtrahend may be complemented as soon as it arrives at the AAU's input. If this precomplementing takes place in a controlled add/subtract circuit comprised of XOR gates, then a savings of three gate delays ($3\Delta_g$) can be realized, where Δ_g is a unit gate delay. To put this savings into proper perspective, we must recognize that a two-level, 32-bit carry-lookahead adder (CLA) has a total latency of $12\Delta_g$.[2] Thus, a 20 % speedup is possible by preprocessing the subtrahend, since the 32-bit adder/subtractor unit is comprised of a controlled complementer followed by a CLA. With reference to Procedure D in Fig. 2, if a and b are the minuend and subtrahend, respectively, the $a^* = a$ and $b^* = -b$. The controlled add/subtract circuit will, therefore, reside in the operand preprocessing unit, while the system function unit will contain the CLA.

The longest carry propogation length determines the length of time required to perform binary addition. Reitweisner,[3] among others, has shown that the average maximum propagation length is $\log_2 p$, where p is the number of bits to the right of the radix point. The ability to take full advantage of this fact to speed up the addition process depends upon the addition algorithms implemented in the system function unit.

Gilchrist, et al., utilized Procedure B to design an asynchronous data-dependent adder.[4] Their system function unit and carry completion unit were a ripple-carry adder and a carry propagation unit, respectively. The carry propagate unit predicts when all of the carries have finished propagating. For a 40-bit adder, they found this algorithm to be 8-times faster on the average than the corresponding ripple-carry adder. We investigated replacing the ripple-carry adder in the system function unit with a CLA and found analytically and through simulations that no improvements in speed up can be made on the average by employing either Procedure A or B. This is because the overhead involved in either predicting or detecting the completion of the addition operation always exceeds the average saving realized by implementing the remainder of the algorithm.

Similar results were encountered when we attempted to apply Procedure E to binary addition. Specifically, if p is the number of bits to the right of the radix point in operands A and B, Procedure E transforms p into p^* through some rounding process such that $p^* < p$. For the ripple-carry adder case, there exists a one-to-one mapping between the reduction in the significance of the operands and the corresponding reduction in the total add time. For example, a 75% reduction in significance results in a 75% reduction in add time. But for the case where the CLA is used in the system function unit, no speedup improvements are realized, because, as before, the overhead exceeds the average potential savings.

With addition we found that the relative improvements possible with self-timing algorithms over their data-independent counterparts depended strongly upon the root algorithm used in the AAU's system function unit. And the same is true for self-timing multiplication algorithms.

Lehman[5] and others[6,7] have considered a variety of data-dependent speedup techniques for recursive add-and-shift multiplication algorithms which use combinations of Procedures A and B. Each of these techniques utilizes the string recoding rule

$$\sum_{i=n}^{m} 2^i = 2^{m+1} - 2^n$$

to reduce the number of individual add and shift operations required. Also, in his modified short-cut (MSC) procedure, Lehman takes into account the fact that strings with isolated 1's and 0's can be recoded. This MSC procedure yields a self-timing algorithm that reduces the average number of additions or subtractions to $m_q/3$ and the maximum number to $(m_q+2)/2$, where m_q is the number of bits in the multiplier. Also, this procedure reduces the average number of individual shift operations by 30%.

But there is further room for increasing the average speed of data-dependent add-and-shift multiplication algorithms. One such technique would be to use Procedure C to preprocess the first operand that arrives by treating it as the multiplier and performing the canonical recoding operation to obtain a new multiplier operand with minimal weight.[8] This will eliminate the need to proceed with recoding during the recursive segment once the second operand arrives. Also, Procedure E may be used in one of two ways to increase the speed of recursive add-and-shift multiplication. First, the input operands can be rounded to the desired significance, thus reducing the average and maximum number of adds and shifts proportionately. Secondly, the input operands would be recoded so that multiplication could proceed from left-to-right using an iterative procedure like the one used in the IBM 360 multiply unit[9] or the on-line multiply technique proposed by Trivedi, et al.,[10] for use in pipeline multiply units. In either case, the multiplication process would terminate when the minimum number of iterations has taken place to achieve the desired product significance.

Now let us consider the prospect of using cellular-array multipliers, such as a Pezaris array,[11] in conjunction with data-dependent speedup techniques. Specifically, various schemes involving Procedures A and B were explored. But, while some array elements may have the correct results early, any attempt to either predict or detect the completion of the multiplication process actually reduces the average speed of the original array or involves an inordinate amount of auxiliary hardware. Not only is there a significant increase in the gate count, but also there is a serious loss of regularity in the original multiplier array structure. This greatly increases the total length of the interconnection paths between cells. These latter two facts result in less efficient utilization of chip area. The only significant self-timing speedup improvements with cellular-array multipliers comes with Procedures B and E. The input operands would be truncated to the desired precision

and then applied as usual to the array inputs. If all truncated bits are forced to zero, then they won't produce any carries. This reduces the length of the critical path in the array by an amount proportional to the reduction in input operand significance. Since the critial path through the array represents the multiplier's latency, its overall latency will be reduced in a "predictable" fashion.

DIVISION

Division is the elementary arithmetic operation which takes the most time and, therefore, has the greatest potential for speed improvement using data-dependent procedures. Let N, D, and Q be the dividend, divisor, and quotient, respectively, such that

$$N = \sum_{i=1}^{m_n} n_i 2^{-i}, \qquad D = \sum_{j=1}^{m_d} d_j 2^{-j},$$

$$Q = N/D = \sum_{k=1}^{m_q} g_k 2^{-k}, \qquad n_i,\ d_j,\ q_k \in \{0,1\}.$$

Also, let us restrict N and D to the following ranges:

$$0 \leq N < 2^{-1}, \qquad 2^{-1} \leq D < 1.$$

So, $d_1 = 1$.

We may define a new parameter δ as follows:

$$\delta = 1 - D \leq 2^{-1}. \tag{1}$$

One division algorithm that is amenable to data-dependent speedup procedures is the convergence division algorithm.[12] With this algorithm, the divisor is forced toward unity, leaving the dividend equal to the quotient, subject to some error limit. The convergence equation is given below:

$$Q = \frac{N}{D} \approx \left(\frac{N}{D}\right) * \left(\frac{R_0}{R_0}\right) * \left(\frac{R_1}{R_1}\right) * \ldots * \left(\frac{R_n}{R_n}\right).$$

For sufficiently large n,

$$D_n = D * R_0 * R_1 * \ldots * R_n \rightarrow 1.$$

Thus,

$$N_n = N * R_0 * R_1 * \ldots * R_n \rightarrow Q.$$

And since the denominator

$$D = 1 - \delta \geq 2^{-1} \quad \text{and} \quad \delta \leq 2^{-1} \tag{2}$$

then the R_i's are chosen according to the following:

$$R_i = 1 + \delta^{2^i}, \qquad i = 0, 1, \ldots, n.$$

After the ith iteration, the denominator becomes

$$D_i = D * R_0 * R_1 * \ldots * R_i = D_{i-1} * R_i = (1 - \delta^{2^{i+1}}). \tag{3}$$

Because $N < 2^{-1}$, $\delta^{2^{i+1}}$ becomes a convenient measure for the closeness of N_i to Q.

With synchronous convergence division algorithms, the number of iterations is determined by the worst case. This occurs when

$$\delta = \delta_{max} = 2^{-1}.$$

If 2^{-m_q} is the desired precision for Q, then

$$\left. \delta_{max}^{2^{i+1}} \right|_{i=n} \leq 2^{-m_q} \tag{4}$$

or

$$n = \left\lfloor \log_2 m_q \right\rfloor - 1 \tag{5}$$

If, for example, the desired quotient precision is 2^{-16}, then $m_q = 16$ and $n = 3$. So four iterations are required to achieve the desired quotient.

But there is a 50% probability that $\delta \leq 2^{-2}$, a 25% probability that $\delta \leq 2^{-3}$, etc. So there is great potential for terminating the division process early using Procedure A. The completion prediction unit would search for the leading zero in D and then set n. For example, if $m_q = 16$ and $D = 0.1110\ldots$ then $n=1$ instead of 3 as for the synchronous case. So only half of the number of iterations would be required. This represents a speed up of 50%, since the determination of n can proceed concurrently with the first iteration.

Procedure C may also be used to speed up convergence division. If the divisor D arrives first, then 1/D may be computed before the numerator arrives. In effect, the convergence division execution time is equivalent to one data-dependent multiply time.

Procedure E may also be used to speed up the computation, since the maximum number of iterations required is dependent upon m_q (see Eq. 4). However, this dependence is very weak. The most efficient use of Procedure E comes in performing the operation

$$D_i = D_{i-1} * R_i.$$

Subtract-and-shift division algorithms may also be sped up through the application of data-dependent procedures, although not to the extent convergence division methods can.

ELEMENTARY SPECIAL FUNCTIONS

Elementary special functions may be executed by algorithms that reside at either the software (instruction) level or hardware (operation) level. Efficient software algorithms handle iterative tasks

by terminating execution prior to the maximum number of iterations needed in the worst case. They also allow greater flexibility and make less demand on hardware resources. On the other hand, hardware-level special function algorithms generally execute the total number of iterations for a task without testing for early termination. As advanced computer structures move toward taking full advantage of very large-scale integrated circuits, a decided shift from the instruction level to the operation level will most likely occur in special function implementation. So it is important to carry over from one to the other the principle of minimizing the execution time by exploiting data-dependent termination procedures.

These techniques can make use of several of the data-dependent procedures described earlier. In general Procedure A can always be employed to detect when the absolute error at the end of the ith iteration is sufficiently small to terminate execution instead of waiting for the worst-case exit time. The operand preprocessing unit in Procedure D can contain a ROM or logic array to provide a first estimate of the result or to recode the input operands. Procedure C can also be utilized to acquire the first estimate of the results for applications in which the incoming time k-1 is highly correlated with data arriving at time k. This would be particularly true when the data-driven machine is applied to such problems as digital filtering, digital control, or simulation of a boundary-value problem, e.g., a real-time hydrodynamics problem. Procedure E can be used to adjust dynamically the maximum absolute error parameters in order to meet overall speed and significance objectives for a particular problem.

Hardware-based special-function algorithms may be classified as either general-purpose algorithms, such as the polynomial evaluation schemes proposed by Tung[13] and Ercegovac[14], or special-purpose algorithms, such as Volder's CORDIC function technique[15] and the convergence computation methods of Chen.[16] The latter class of hardware algorithms are based on certain specific properties of the functions being evaluated and are designed with speed efficiency in mind at the expense of having only a limited domain of application.

We will demonstrate the application of these procedures by considering the following special-purpose elementary function algorithm: Newton's method for determining the square root Y of a number of X ($Y = \sqrt{X}$) is specified by the following recursive procedure:

BEGIN NEWTON'S METHOD

Step 1 -- Read X and E
Step 2 -- Make an initial estimate of Y
Step 3 -- Repeat $Y \leftarrow 2^{-1}[Y + X/Y]$ until $|Y - X/Y| < E$

END NEWTON'S METHOD

For certain operation-level implementations, the maximum absolute error E may be fixed and only the final result rounded to the desired significance.

Let us restrict our attention to the conventional radix number system with $r = 2$ and

$$X = \sum_{i=1}^{m_x} x_i 2^{-i}, \quad Y = \sum_{j=1}^{m_y} y_j 2^{-j}, \quad x_i, y_j \in \{0,1\}$$

and, moreover, restrict X to the range

$$2^{-2} \leq x < 2^{-1}.$$

Application of the basic synchronous algorithm begins with an initial estimate of Y, e.g., $Y = 2^{-1}$, followed by the number of iterations required for the worst-case convergence. For example, if $n_x = 32$ and we require Y to be accurate to a precision of 2^{-16}, then the worst case number of iterations would be 3. And this would collectively involve 3 divides, adds, and shifts.

But the following procedures can be used to reduce the average execution time for this root-execution algorithm by more than 50%:

* If the value of input operand X at time k is highly correlated with the input operand at time k-1, then Procedure C can be used to obtain a better initial estimate of Y. This estimate would be used in conjunction with Procedure A to exit early from the computation (see Table I). Procedures D and B can be used in a similar manner. But here the input operand X would be applied to the input of a ROM or gate array to acquire the first estimate.

TABLE I

Number of iterations required to implement Newton's method as a function of the desired precision, N_p, and number of leading bits, N_k, known to be correct initially.

N_k \ N_p	1	2	4	8	12	16	20	24	28	32
1	0	1	1	2	3	3	3	3	4	4
2	0	0	1	2	3	3	3	3	4	4
4	0	0	0	1	2	2	3	3	3	3
8	0	0	0	0	1	1	2	2	2	2
12	0	0	0	0	0	1	1	1	2	2
16	0	0	0	0	0	0	1	1	1	1

* Because the intermediate value of Y obtained at the ith iteration is highly correlated with Y obtained at the (i-1)st iteration, the division step (X/Y) can "converge" much more quickly than the worst case using Procedures C and A (see Table II).

* Through significance control, Procedures E and B can be used to reduce the required number of iterations (see Tables I and II).

TABLE II

Number of iterations required to find the reciprocal of y as a function of the required precision, N_p, and the number of leading bits known to be correct.

N_k \ N_p	1	2	4	8	12	16	20	24	28	32
1	0	1	2	3	3	4	4	4	5	5
2	0	0	1	1	2	2	2	3	3	3
4	0	0	0	1	2	2	2	3	3	3
8	0	0	0	0	1	2	2	2	2	3
12	0	0	0	0	0	1	1	1	2	2
16	0	0	0	0	0	0	1	1	1	1

* Application of the above three sets of procedures always assumed that the input data X had worst case values. In principle, Procedure A can be used alone to test when the error criteria is met. For example, if $m_x = 32$, $m_y = 16$, and the initial estimate for $\sqrt{X}$ is $Y = 2^{-4}$, then there is about a 33% chance that one of the three worst-case iterations can be bypassed.

We have examined several other general-purpose and special-function operation-level algorithms that are either used or have been proposed for use in evaluating special functions. In general, the data dependent procedures that were applied to Newton's method for evaluating square roots apply equally well to other existing special function algorithms. But there are some exceptions.

One standard method for polynomial evaluation of special functions is to represent the function in an infinite power series, truncate the series to k + 1 terms and then re-express the truncated approximation in a nested form as shown below:

$$f(x) \approx f_k(x) = \sum_{j=o}^{k} a_j * x^{m+j\,n} = x^m * \left[\sum_{j=o}^{k} a_j * x^{jn}\right]$$

$$= x^m * [((\ldots(((a_k * x^n) + a_{k-1}) * x^n + a_{k-2}) * x^n + \ldots + a_1) * x^n + a_o)]$$

The advantage of this formulation is that at most two powers of the variable x, namely x^m and x^n are required; moreover, the inner loop in the iterative procedure is quite simple--one multiply and one add. While this formulation is quite simple for sequential algorithms where k is chosen based on worst-case considerations, it is not a desirable approach for asynchronous special-function algorithms, because there is no early termination capability.

The CORDIC trigonometric computing technique represents another sequential algorithm that must be modified before data-dependent procedures can be fully exploited to reduce execution time. With the CORDIC technique,[15] a vector is rotated from some starting point

$$Y_1 = R_1 \sin\theta_1, \qquad X_1 = R_1 \cos\theta_1.$$

to some final position

$$Y_{n+1} = K_n R_1 \sin(\theta_1 + \lambda_n), \quad X_{n+1} = K_n R_1 \sin(\theta_1 + \lambda_n)$$

where

$$\lambda_n = \xi_1 90^o + \sum_{i=2}^{n} \xi_i \alpha_i, \qquad \alpha_i = \tan^{-1} 2^{-(i-2)},$$

$$\xi_i = \pm 1 \qquad K_n = \prod_{i=2}^{n} \left[\sqrt{1 - 2^{-2(i-2)}}\right].$$

Traditionally, n is fixed, so the α_i's and K_n are precomputed and stored in ROM. So, the only decision in moving from the initial position to the final position is the sequential selection of the ξ_i's. The problem with directly adapting this algorithm to asynchronous algorithm design is that if n is to be a variable to allow early termination, then all possible K_n's would have to be stored. The second problem with the algorithm lies with the fact that it is a sequential search operation that requires a rotation at each iteration. String recoding techniques that would reduce the number of rotations from n to an average of $O[\log_2]n$ are not possible with the current formulation of the CORDIC algorithms.

CONCLUSION

This paper investigates the potential of using asynchronous instead of sequential arithmetic units as processing elements in data-flow machines. Five general procedures are described that may be used to design self-timing processing units: task completion prediction, task completion detection, operand preprocessing, pre-estimation of input operand values, and significance control.

The potential for successfully employing these procedures was found to be related to the granularity of the arithmetic operation. Addition and subtraction operations cannot be improved appreciably using data-dependent procedures. Multiplication can be sped up if add-and-shift type algorithms are to be used but not when cellular-array multipliers are. Division and special-function operations appear to hold the greatest potential for speedup through the application of data-dependent procedures. As advanced structures move toward taking full advantage of very large-scale integrated circuits, a decided shift from the instruction level to the operation level will most likely occur in special function implementation. And it will be important to carry over from one to the other the principle of minimizing special-function execution times by exploiting data-dependent termination procedures.

Although these procedures were considered in terms of their application to data-dependent ma-

chines, in principle they may also be applied to sequential single-processor or multiple-processor instruction-driven machines.

ACKNOWLEDGEMENTS

T. L. Chang originally suggested the operand preprocessing procedure. He also had many other useful comments and suggestions. The authors also wish to thank Ginny Mrazek for typing this manuscript, as well as Bill Pearson for preparing the figures and tables.

REFERENCES

1. J. B. Dennis, "Data Flow Supercomputers," IEEE Computer, vol. 13, pp. 48-56, (Nov. 1980).

2. Hwang, K., Computer Arithmetic: Principles, Architecture, and Design, John Wiley and Sons, New York, pp. 88-91, (1979).

3. Reitwiesner, G. W., The determination of carry propogation length for binary addition," IRE Trans. or Electronic Comp., vol. EC-9, pp. 35-38, (Mar. 1960).

4. Gilchrist, B., Pomerene, J. H., and S. Y. Wong," Fast carry logic for digital computers," IRE Trans. on Electronic Comp., vol. EC-4, pp. 133-136, (Dec. 1955).

5. Lehman, M., "Short-cut multiplication for binary digital computers," Proc. IEEE. (London), vol. 10, pp. 496-504, (Sept. 1958). 6.Smith, J. L. and A. Weinberger, "Shortcut multiplication for binary digital computers," System Design of Digital Computer at the National Bureau of Standards: Methods for High-Speed Addition and Multiplication, NBS Circular No. 591, Sec. 1, pp. 13-22, (Feb. 1958).

6. Smith, J. L. and A. Weinberger, "Shortcut multiplication for binary digital computers," System Design of Digital Computer at the National Bureau of Standards: Methods for High-Speed Addition and Multiplication, NBS Circular No. 591, Sec. 1, pp. 13-22, (Feb. 1958).

7. Tocher, K. D., "Techniques of multiplication and division for automatic binary computers," Quart. J. Mech. and Appl. Math., vol. 11, pp. 364-384, (Aug. 1958).

8. Reitweisner, G. W., "Binary arithmetic," Advances in Computers, vol. 1, Academic Press, New York, pp. 231-308, (1960).

9. Anderson, S. F., Earle, J. G., Goldschmidt, R. E., and D. M. Powers, "The IBM System/360 Model 91: floating-point execution unit," IBM J. Res. and Develop., vol. 11, pp. 34-53, (Jan. 1967).

10. Trivedi, K. S., and M. D. Ercegovac, "On-line algorithms for division and multiplication," IEEE Trans. on Comp., vol. C-26, pp. 681-687, (July 1977).

11. Pezariz, S. D., "A 40-ns 17-bit by 17-bit array multiplier," IEEE Trans. on Electronic Comp., vol. C-20, pp. 442-447, (April 1971).

12. Hwang, K., op. cit., pp. 242-254.

13. Tung, C., A Combinational Arithmetic Function Generation System, Ph.D. Dissertation, University of California, Los Angeles, Ca., (1968).

14. Ercegovac, M. D., "A general hardware-oriented method for evaluation of functions and computations in a digital computer, IEEE Trans. on Comp., vol. C-26, pp. 667-680, (July, 1977).

15. Volder, J. E., "The CORDIC trigonometric computing technique," IRE Trans. on Electronic. Comp., vol. EC-8, (Sept. 1959).

16. Chen, T. C., "Automatic computation of exponentials, logarithms, ratios and square roots," IBM J. Res. and Develop., vol. 16, pp. 380-388, (July 1972).