Design and Analysis of Generalized Dynamic Pipeline for Scientific Functions

Suresh Thirumalaiswamy and Syed Masud Mahmud
Dept. of Electrical and Computer Engineering, Wayne State University, Detroit

Abstract—A generalized pipeline has been proposed which can perform the evaluation of scientific functions like Sin(x), Sinh(x), Sinuffling(x), x^x, logarithm, and antilogarithm. In this design, the same pipeline is used for evaluating different scientific functions. This pipeline is dynamic and can achieve significant speed improvement over other pipeline architectures available in the literature. Furthermore, it utilizes hardware efficiently and has more application flexibility.

Index Terms—Pipeline, Arithmetic Pipeline, Dynamic Pipeline, Computer arithmetic, multiplication, square root, trigonometric, logarithm, power.

INTRODUCTION

Pipeline is one of the ways to improve parallelism in processors, thereby increasing its throughput. Arithmetic pipeline is one of the classifications of the pipelined processors. Here, the arithmetic logic unit is segmented into a number of functional units for pipeline operations. Arithmetic pipeline is used in STAR-100 [1] and TI-ASC [2] which has eight pipeline stages, in Cray-1 which has 14 stages and up to 26 stages in Cyber-205 [3]. Texas Instruments’ Advanced Scientific Computer [2] was the first vector processor with multifunction pipelines in its arithmetic processors. The ASC arithmetic pipeline consists of eight stages. This system uses the same pipeline for different function evaluations. All the eight stages in the pipeline are interconnected in such a way that different logic instructions are allowed to use different connecting paths through the pipeline. The ASC pipeline is still a static one performing only one function at a time. Hallin [4] studied the effect of pipelining on system efficiency by implementing three multiplications and two addition algorithms. He studied the problem of pipelining addition and multiplication functions of the arithmetic unit and from the results obtained he concluded that pipelining would be applicable to any combinational logic circuit and is an efficient method for achieving greater bandwidth in execution unit. Kanal et al. [5] proposed a multifunction arithmetic pipeline. This pipeline can perform four functions: multiply, divide, square and square root operations. Two types of building cells are used in this four-function pipeline construction. The two cell types are specified by boolean equations. One cell is a controlled 1-bit adder/subtractor with bypass signal lines, while the other cell is for function selection and boundary carry control. Each stage of the pipeline is essentially a ripple-carry adder/subtractor. This multifunction pipeline operates in a static manner. Somme other pipeline designs [6],[7] are also available in literature which are suitable for VLSI implementation.

All the pipeline systems discussed above are either static or can perform only one particular arithmetic function. This paper presents a multifunction scientific pipeline. Unlike the systems discussed above, this pipeline is dynamic. It can support the evaluation of scientific functions like Sin(x), Sinh(x), Sinuffling(x), x^x, logarithm, and antilogarithm. Hardware algorithms are developed for these functions individually. It is found that the algorithms of many scientific functions have a number of common features. Then a generalized algorithm is developed to evaluate a number of scientific functions. Based on this algorithm a scientific pipeline is developed.

The algorithms for individual arithmetic functions are discussed in Section I with the help of flow charts. In Section II a detailed implementation of the proposed generalized pipeline and its control unit are discussed. Also, the proposed pipeline is compared with two other pipeline architectures in Section III.

1. ALGORITHMS FOR INDIVIDUAL FUNCTIONS

Evaluation of Trigonometric Functions

The trigonometric functions considered in this pipeline are Sin(x), Sinuffling(x), and Sin(x). These functions are evaluated using polynomial approximations within a given precision range [8]. Generally, the trigonometric functions can be written in the form of an infinite power series as follows:

\[
f(x) = \sum_{j=0}^{\infty} a_j \times x^j
\]

To evaluate these functions and to implement them in hardware, the truncated series are re-written in a noded-product form as follows for the general case of (k + 1) terms:

\[
f_k(x) = x^m \times \left( \sum_{j=0}^{k} a_j \times x^j \right) = x^m \times \left( \left( \left( \left( \left( \left( a_0 + a_1 \times x \right) + a_2 \times x^2 \right) + a_3 \times x^3 \right) + \ldots \right) \right) \right) \]

Thus, using the above expansion the function f_k(x) can be evaluated with at most two powers of the variable x, namely x^m and x^k. It is obvious that without the nested formulation, (k + 1) powers of x are required which results in longer computation time. Simple iterative multiplications and additions can be used to evaluate the above mentioned equation. Therefore, a (k + 1) term power series may require (k + 1) multiplications interleaved with (k) additions. Even though the successive coefficients can be obtained by a general formula, we use a lookup table for the different coefficients.

Therefore, the nested-product form for the trigonometric functions considered in the pipeline are:

\[
\begin{align*}
\text{Sin}(x) &= x \times \left( 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \ldots \right) \\
\text{Sinh}(x) &= x \times \left( 1 + \frac{x^2}{2} + \frac{x^4}{24} + \frac{x^6}{720} + \ldots \right) \\
\text{Sin}-1(x) &= \frac{x}{1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \ldots} \\
\text{Sin}(x^2) &= x^2 \times \left( 1 - \frac{x^4}{2} + \frac{x^8}{24} - \frac{x^{12}}{720} + \ldots \right) \\
\text{xy} &= x \times y \\
\text{log}(x) &= \ln(x) \\
\text{antilog}(x) &= e^x
\end{align*}
\]

From the above equations it is obvious that it is the value of ak (here, 0 = k = 5) which varies. Also, we need only two values of 'x' i.e., x and x^2. With these two values of 'x' and storing the values of ak in some hardware, additions can be used to evaluate

\[
\begin{align*}
\text{Sin}(x) &= x \times \left( (a_0 + a_1 \times x) + a_2 \times x^2 \right) \\
\text{Sin}(x^2) &= x^2 \times \left( (a_0 + a_1 \times x) + a_2 \times x^2 \right)
\end{align*}
\]

The value of 'x' is modified for different bits according to the value of each bit of the exponent. Let us consider a numerical example to see how the value of 'x' is modified for different bits of the exponent.

Let \( A = 2^{20} \) and let \( y_1 = y_1 y_2 y_3 \ldots \) where \( y_i \) is the integral part of the exponent, \( y_r \) is the fractional part of the exponent and \( \perp \) is the radix point.

Let us first consider the evaluation of the fractional part of power.

Let \( A = 2^{y_r} \) where \( y_r = y_0 y_1 y_2 y_3 \ldots \) and \( y_0 \) \( E(0.11) \) for \( i = 0, 1, \ldots, n \).

Since the exponent is represented in the binary form, each digit of it can take only two values (0 or 1). Thus, the value of the base 'x' is modified according to the value of each bit of the exponent 'y'. Let us consider a numerical example to see how the value of 'x' is modified for different bits of the fractional part of the exponent.

Let \( A = 2^{0.1875} = 2^{0.00111} \) where the period () is the radix point.

\[
\begin{align*}
A &= 2^{0.001} \times 2^{0.111} \\
&= 2^{0.001} \times 2^{0.111} \\
&= 2^{0} \times 2^{0} \times 2^{0} \\
&= 2^{0} \times 2^{0} \times 2^{0} \\
&= 2^{0} \times 2^{0} \times 2^{0}
\end{align*}
\]
Now, from the last step of the above equation it is clear that the first two values after the radix point is 1. Let us take the cases of 2/1/8 and 21/1/8. These two can be written as shown below.

\[ \begin{align*}
2/1/8 &= 2(1/2 * 1/2 + 1/2) = \sqrt{2} (2) \\
21/1/8 &= 2(1/2 + 1/2 + 1/2) = \sqrt[3]{2} (2). 
\end{align*} \]

Thus based on whether the binary digit of the exponent is 0' or 1', the value of the base is either not square-rooted or square-rooted, respectively. Also, the number of times the base is square-rooted depends on the weightage of the binary digit of the exponent. That is, for a value of '1' in the third position of the exponent the base is square-rooted thrice, because the weightage is 1/8 (= 1/2 * 1/2 * 1/2). Moreover, each time the base value is changed it is multiplied to the previous value of the result. This above algorithm is explained in a detailed fashion in Figure 2. In this flow chart, the initial values for iteration count 'i' is set to the number of bits of 'result' only when a '1' appears in the binary digit of the exponent. This is done because of the following feature of the algorithm. We see that in Eqn. 1 the base is square-rooted thrice and four times when a binary '1' appears in the exponent in the third and fourth position respectively. Therefore the number of times the base is square-rooted depends on the position of the binary digit '1' in the exponent. Thus the base is square-rooted "once" for each binary digit of the exponent. But each time the value of square-rooting is not multiplied to the 'result' as discussed earlier. The value of square-rooting is saved each time and is multiplied to the result only when the binary digit of the exponent is 1'. The above procedure is shown as flow chart in Figure 2.

Therefore, the algorithm can be summarized as follows for the decimal part of power: If the value of binary digit of the exponent is '0', the base is just square-rooted once. When a '1' occurs the base is square-rooted once and multiplied to the value of 'result'. When the iteration ends, the final value will be in 'result'.

In a similar manner, the integer part of the power can be evaluated as shown below.

\[ A = x_i^y_i \]

where \( y_i = y_{i-1} + 2y_{i-2} + y_{i-3} \ldots y_{i-n} \) and \( y_j \in [0,1] \) for \( j = 0, 1, \ldots, n \).

Following the same discussion as in fractional evaluation, the algorithm for evaluation of integer part of the power can be summarized as follows: If the value of binary digit of the exponent is '0', the base is just square-rooted once. When a '1' occurs the base is square-rooted once and multiplied to the value of 'result'. When the iteration ends, the final value will be in 'result'.

**Evaluation of Logarithm**

The algorithm followed here is the iterative technique which has better accuracy over other available methods. Majithia and Levan [9] proposed an iterative technique where in each step a constant is compared with \( \sqrt[]2 \). This could not be expressed exactly for binary expression and errors are introduced in the comparative process. Lo and Chen [10] proposed another iterative technique with a different comparative index. This algorithm is well suited for the integrated pipeline we are proposing. The advantage of both the above methods is that accuracy increases as the number of iterations increases.

The proof is discussed in detail in [10]. The algorithm for evaluating logarithm can be summarized as follows:

Let \( X = \log_2 A \) where \( 1 \leq A < 2 \)

\[ A = 2^x = (2^x_1 + 2^{x_2} + 2^{x_3} + \ldots + 2^{x_n}) \]

Thus, the conditions for the algorithm can be formulated as follows: If the value of \( A^2 \) is greater than or equal to \( 2 \) then the present bit being determined is '1'. If the present bit is less than \( 2 \) then the present bit is '0'. The algorithm is shown as flow chart in Figure 3.

**II. THE SCIENTIFIC PIPELINE**

Based on the discussions from the previous sections the arithmetic modules required for the evaluation of different functions are: (i) the multiplier, (ii) the adder, (iii) the subtractor, (iv) the squarer and the (v) square root arithmetic modules as summarized in Table 1. Of these five modules the adder and the subtractor can be combined into one module.

This section the need for pipelining the individual arithmetic modules and the different stages of the pipeline is first discussed. Then the design of the pipeline stages and finally the control unit of the pipeline are discussed.

Of all these stages the multiplier and the squarer stages have the maximum gate delay of 45A. The different stages of the pipeline could be the modules mentioned in the Table 1. The algorithms used in the implementation of the different functions are highly iterative. Thus, these stages will be used iteratively for 'n' number of times according to the function (as indicated by * in Table 1), where 'n' is the number of bits to be determined in the individual function evaluation. Since the clock of the pipeline is 45A, the time required to evaluate a function will be 45nA. Instead the individual modules can be pipelined in order to reduce the clock of the pipeline.

Based on the above discussions, the individual modules were pipelined to obtain the optimal stages. Thus the multiplier is divided into 3 stages, the adder/subtractor and the square root modules are divided into 2 stages. In this case the clock of the pipeline is reduced to 18A. Thus, the different stages of the optimal pipeline are: (i) a 3-stage multiplier, (ii) a 2-stage adder/subtractor, and finally (iii) the 2-stage square root module as shown in Figure 4.

**Multiplier Design**

Multiplier and adders are widely needed for designing multipliers, and inner-product computers. Wallace trees are the theoretically fastest multioperand adders [11]. The design used here organizes the CSA's like an Overturned Stairs (OS) [12]. A CSA tree of height 12 is required for a 64-input OS tree. The delay for the multiplier module is calculated as follows.

The multiplier has three major delay modules namely, (i) partial product generator which has a delay of 1A, (ii) 64-input OS tree with a tree height of 12 has a delay of 24A and (iii) the carry look-ahead adder which has a delay of 12A. Thus, the total delay of the multiplier module is 45A. The squarer design is same as that of the multiplier and has a gate delay of 45A.

**Adder/Subtractor Design**

The module is designed with the help of Carry Look-ahead Adder (CLA), the design of which can be found in [8,13,14]. The different delay modules of this design are: (i) a presentation adder for subtracting the exponents which has a delay of 8A, (ii) a 6-bit shifter to align the mantissas which has a delay of 7A, and, (iii) the adder/subtractor unit which has a delay of 13A. Thus the total delay of this module is 30A.

**Square Root Design**

The trend in the realization of the square root operation in future mathematical co-processors will be oriented more and more in the direction of direct methods, slowly abandoning the software and microcoded implementations [15]. The algorithm followed here is the Non-restoring algorithm of the Direct method for the extraction of the square root. In this method for binary numbers, at each step of the algorithm, a digit of the result is produced by inspecting the shifted partial remainder which is derived from the previous digit selections [16]. The algorithm can be summarized as follows: The first step is to subtract '01' from the operand as shown in the flow chart in Figure 5. The next step is determined completely by the sign of the remainder. When the remainder is positive, a root bit of '1' is always selected, '01' is appended to the root bit developed, which is the next test value or if the remainder is negative, the root bit is '0', '1' is appended to the root bit developed, which is the next test value. Then the remainder and the new test value are subtracted. Then the remainder is added to the new test value. The sign of the result is obtained due to this addition or subtraction is checked and the previous steps are repeated accordingly for positive/negative remainder. The total gate delay of this module is 30A.

**Pipeline Control Unit**

There are two controls required for the efficient operation of a pipeline. They are the pipeline initiation control and secondly, the in-pipelining control. Let us first consider the pipeline initiation control.

**Pipeline Initiation Control Unit**: A pipeline which can evaluate 'n' functions has 'n' collision matrices each of which is of size (n x r), where 'r' is the total time units required to complete a function evaluation. Since this pipeline...
is dynamic, there may be more than one reservation active at any instant of time. Thus, when a function arrives, it has to be initiated only after taking into consideration all the active reservation tables. Thus at any instant of time, the collision matrix reflects the current state of the pipeline. From this, the decision as to whether a function can be initiated or not can be determined.

Based on the above discussion, the pipeline control unit for the proposed pipeline can be designed as follows [17]. A flag is maintained to determine whether the pipeline is empty, where a '0' for the flag indicates that the pipeline is empty and '1' indicates it is not empty. Thus when a function of (type 'x') arrives, the current state of the pipeline can either be empty or it has one or more functions in it. If the flag is '0', then it implies that the pipeline is empty and thus the function is initiated immediately. If instead, the pipeline has one or more functions in it indicated by a '1' for the flag, then the collision matrix in the bank of shift registers is shifted left by one position. Since each collision vector in a collision matrix corresponds to a particular type of function if the shifted out bit of the collision vector of type 'x' is a '0' in the collision matrix, then this function can be initiated. Also, since this pipeline is dynamic the shifted out bit of other collision vectors are also checked to see if any other function can be initiated if available, in the function queue. When a function is initiated the flag is set to '1'.

Therefore, when a function is initiated its corresponding collision matrix has to be ORed bit by bit with the current collision matrix in the bank of shift registers. This resulting collision matrix reflects the state of the pipeline after the initiations and this is shifted for determining future initiations. Now, the collision matrix corresponding to the initiated function has to be determined. Also, if a shifted out bit from the bank of shift registers is '0', then the first function in the function queue is checked for the availability of the corresponding function so that it can be initiated. Thus to implement these requirements, each function that arrives is associated with a tag and this tag has two parts, Tag1 and Tag2. Thus, Tag1 is used to determine the corresponding collision matrix of the initiated function and Tag2 is used to select the corresponding shifted out bit of the collision vector of type 'x'.

Intrapipeline Control Unit: The pipeline initiation control unit is not concerned with the data handling and the path taken by the function once it has been initiated [17] but only with the initiation of the functions. Once a function has been initiated it follows the path determined by its reservation table. The intrapipeline control unit is concerned about the route to be taken by each initiated function, the selection of function within a stage and to inform each stage that the data is valid. Since this is a dynamic pipeline the path taken is determined by the type of function that has been initiated in the pipeline and may also depend on the result of a stage. Thus the path taken by a function changes dynamically, which necessitates the decoding of the path at each clock. Thus with each function that is initiated a corresponding tag is associated with it. This tag is decoded at each clock at each stage to determine the path to be followed by the function and the function selection within the selected stage.

III. DISCUSSION

In this section the proposed scientific pipeline will be compared with two other pipeline architectures namely, Texas Instruments' Advanced Scientific Computer (TI-ASC) and the Generalized pipeline array proposed by Kamal [5].

In the case of TI ASC, it has eight stages and the pipeline is configured in such a way that different functions can take different paths but only one function can be active at any instant in the pipeline. Thus the pipeline is multifunctional but static. In the case of generalized pipeline array the number of stages depends on the precision of the result required. Also, in this pipeline at each clock cycle only one operation can be initiated but different functions can be initiated at different clocks. Thus this pipeline is also called a sequential pipeline. The proposed scientific pipeline has six stages but more than one operation can be initiated at any instant of time. Thus the pipeline is dynamic unlike the other two pipelines. Also, the different functions can take different paths in the proposed pipeline which is true in the case of TI ASC and not in the generalized pipeline array.

In the generalized pipeline array the number of adder/subtractor cells required increases as the number of bits to be determined in the function evaluation increases. But in the case of the proposed pipeline and also in TI ASC the subfunction modules are fixed and is capable of performing double precision computations.

Also, the clock of the proposed scientific pipeline is much better than the other two pipeline architectures. The clock of the TI ASC pipeline is 60ns whereas that of the proposed pipeline is 27ns. In the case of generalized pipeline array each stage has adder/subtractor cells connected in a ripple carry fashion. The clock of the pipeline depends on the number of cells in the last stage. Thus for an 'n' bit computation (2n+1) cells are required in the last stage. Thus the clock of this pipeline for double precision computations will be 65 gate delays which is much greater than the proposed pipeline.

The generalized pipeline array is capable of evaluating the basic arithmetic functions like addition, subtraction, multiplication and also functions like squaring and square rooting. The TI ASC can compute both scalar and vector operations. The proposed pipeline can not only compute the basic arithmetic functions but can also compute a wide range of trigonometric functions, logarithm base-k, power, antilogarithm and square root. Thus it is much better than the generalized pipeline array and is comparable to TI ASC in scalar function computations.

Finally, the proposed pipeline has much better application flexibility over other two pipelines. This is made possible because of the independent intrapipeline control where the tag associated with each function helps in determining the path to be taken by the newly added function. Thus no extra changes are needed except for the change in the collision matrices. But in the case of other two pipelines the path is set previously and thus has to be rewired if new functions are to be introduced.

IV. CONCLUSION

Thus the proposed pipeline is capable of handling trigonometric functions like addition, subtraction, multiplication and division in a dynamic manner. It is also capable of handling other trigonometric functions like tan(x), tan⁻¹(x), tanh(x), tanh⁻¹(x) without any changes to the presented algorithm. It is also capable of handling other functions with slight changes in the algorithm. Since the stages of the pipeline are optimal it utilizes the hardware efficiently. Also from the previous section it is obvious that it has much better speed and application flexibility than the pipeline architectures compared. Thus this pipeline can be used as a scientific co-processor where scientific computations are high. Also, it can be designed as an ASIC.

REFERENCES


Table 1: List of Arithmetic Modules required for the following functions: Trigonometric, Power, and Logarithm.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Trigonometric</th>
<th>Power</th>
<th>Logarithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squarer</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes*</td>
</tr>
<tr>
<td>Multiplier</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
</tr>
<tr>
<td>Add/Subtractor</td>
<td>Yes*</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Square root</td>
<td>No</td>
<td>Yes*</td>
<td>No</td>
</tr>
</tbody>
</table>

*Arithmetic Modules Needed more than once in the function evaluation.