ISTANBUL UNIVERSITY ENGINEERING FACULTY
JOURNAL OF ELECTRICAL & ELECTRONICS

YEAR : 2002 (409-415)
VOLUME : 2
NUMBER : 1

A FAST DIVIDER IMPLEMENTATION BASED ON THE NEWTON-RAPHSON METHOD USING PARALLEL COMPUTATION UNITS

Ahmet SERTBA[a]

Istanbul University, Faculty of Engineering, Computer Engineering Department
34850, Avcilar, Istanbul, TURKEY

e-mail: asertbas@istanbul.edu.tr

ABSTRACT

In this paper, a special divide hardware unit based on the Newton-Raphson iteration method is proposed. To compute the reciprocal fast in division process, it utilizes fourth order Newton-Raphson reciprocal approximations. By using fast and efficient parallel computation units the divider achives the computations. These units compute the second, third and fourth order Newton Raphson terms more faster than the standart technique using direct multipliers.

Key Words: *Division, Newton-Raphson divider, parallel computation, hardwire units.*

1. INTRODUCTION

As it is well kown, a division can be expressed as the product of the dividend, and the reciprocal of the divisor , q= a/b = a*(1/b). To compute the reciprocal term (1/b) , the multiplicative iterative methods such as Newton-Raphson and Taylor Series expansion can be used. The Newton-Raphson Method with high-order iterations can be written as follows:

$$X_{i+1}=X_i+X_i(1-bX_i)+X_i(1-bX_i)^2+.....+X_i(1-bX_i)^n \quad (1)$$

Flynn[1] shows that the error decreases exponentially as $E_{i+1}=b*E_i^{n+1}$ for an n_{th} order Newton-Raphson iteration given above.

Using standart arithmetic units, it requires four iterations to achieve an error reduction of $E_{i+4}=b*E_i^{16}$ for the first order implementation. On the other hand, for the thirth order and the fourth order Newton-Raphson implementation, in only two iterations, they achieves an error reduction of $E_{i+2}=b*E_i^{16}$ and $E_{i+2}=b*E_i^{25}$ respectively.

In previous work [2-3], the use of the parallel squaring and cubing units were applied to the Newton-Raphson iterative divide unit.

In this work, to reduce the reciprocal error, the fourth order parallel computational unit proposed by [4] is used to compute the fourth order Newton-Raphson iteration.

2. CLASSIC NEWTON-RAPHSON DIVIDE

The classic first-order Newton-Raphson divide unit iteration is $X_{i+1}=X_i(2-bX_i)$. It is clear that the error decreases quadratically for each iteration., $E_{i+1}=b*E_i^2$. To determine the initial value for the reciprocal term of 1/b, a ROM table bokup is used, before the first iteration starts. In this method, each iteration needs two multiplication and one subtraction (2's complement) operations. On the other hand, to compute $(2-bX_i)$ term in a single operation is possible. This term is called as the fused multiply-subtract. After the last iteration, the quotient (q) is computed by

multiplying the dividend (a) by the reciprocal of the divisior (1/b).

Figure 1 shows a Newton-Raphson divide unit with a first order iteration. In the classical technique, the operations can not be computed independently and they depend on the result produced by the previous operation.
Therefore this does not allow any parallel process. The latency of the divide is given as follow, if k iterations are required for the operation:

$$t_{divisio} = t_{lookuptable} + 2k\, t_{mult} + t_{mult}. \quad (2)$$

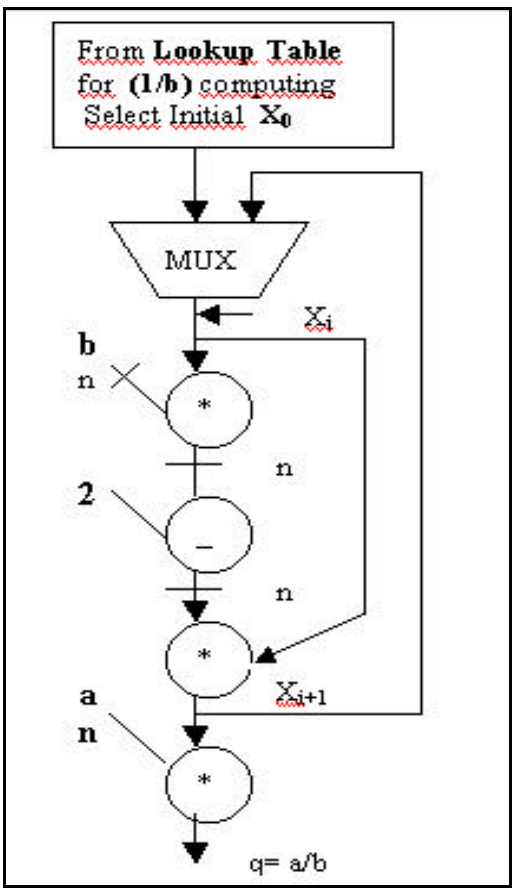

Fig.1. Classic divide unit for the first order

3. STANDART FOURTH-ORDER N.-R. DIVIDE

By means of the k_{th} order iterative function given in (1), the Newton-Raphson fourth order iteration function can be expressed as follow:

$$X_{i+1}=X_i(1+(1-bX_i)+(1-bX_i)^2+(1-bX_i)^3+(1-bX_i)^4 \quad (3)$$

In order to compute an n-bit reciprocal in a single iteration, the look-up table size must be approximately (n/5)x(n/5) bits.

Figure 2 shows a fourth order divide unit implemented using classic technique. The subtraction and additions can be fused with the multiplications as described before.
To implement the fourth-order Newton-Raphson Divide, a single or double multiplier may be used. If k iterations with a single multiplier are used, the latency of the divide is approximately as follow:

$$t_{divisio} = t_{lookuptable} + 5k\, t_{mult} + t_{mult}. \quad (4)$$

With two multipliers are used, the latency of the unit can be given as:

$$t_{divisio} = t_{lookuptable} + 5k\, t_{mult} \quad (5)$$

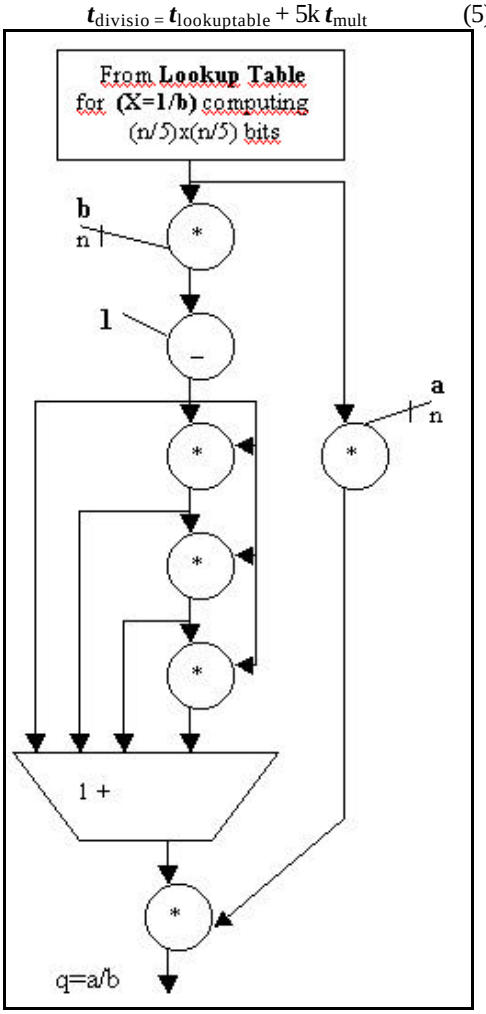

Fig.2. The fourth-order Newton-R. Divide Unit

4. PROPOSED FOURTH-ORDER N.-R. DIVIDE UNIT

In a division operation, to compute the quotient (q) as a parallel process, the special computing units called as parallel computational units may be used To show the efficiency of these units in the divider implementation, the Newton-Raphson iteration can be expressed as the fourth order approximation.

$$q=a/b=aX[1+(1-bX)^1+(1-bX)^2+(1-bX)^3+(1-bX)^4] \quad (6)$$

Where X is the initial prediction of the reciprocal and can be taken the value from the lookup table. Figure 3 shows the hardware section required to implement the fourth order divide unit.

The latency of the divide unit may be given as in (7) , as long as all of the powers of (1-bX) term can be computed directly by using the parallel computational units that take the same time to perform one multiplication.

$$t_{divisio} = t_{lookuptable} + 3\,t_{mult} \quad (7)$$

The divider unit requires tree multipliers, one squaring unit, one cubing unit and one fourth order exponential computation unit. But the two multipliers have small bit lenght (n/5 *n) and are used to multiply (1-bX), (a*X) terms. The third one is full lenght multiplier (n*n) and used to multiply quotient (q=a/b). On the other hand, the parallel squaring and cubing units are described in detail [2]. Also, the fourth-order parallel computation unit is modeled mathematically [4] in the references.

In the next section, the proposed divider is implemented for 24-bit operand, the length of the IEEE single precision floating-point format.

5. THE DIVIDER IMPLEMENTATION

Firstly, by a table lookup, an initial value of the reciprocal of the divisior should be determined. As well known, for the lookup table, a 2^lxm bit ROM with l address bits and an m-bit output words is required.

In the computations, to avoid getting the negative numbers from (1-bX) term, the value stored in each lookup table address should be less than the reciprocal of all possible values of b.

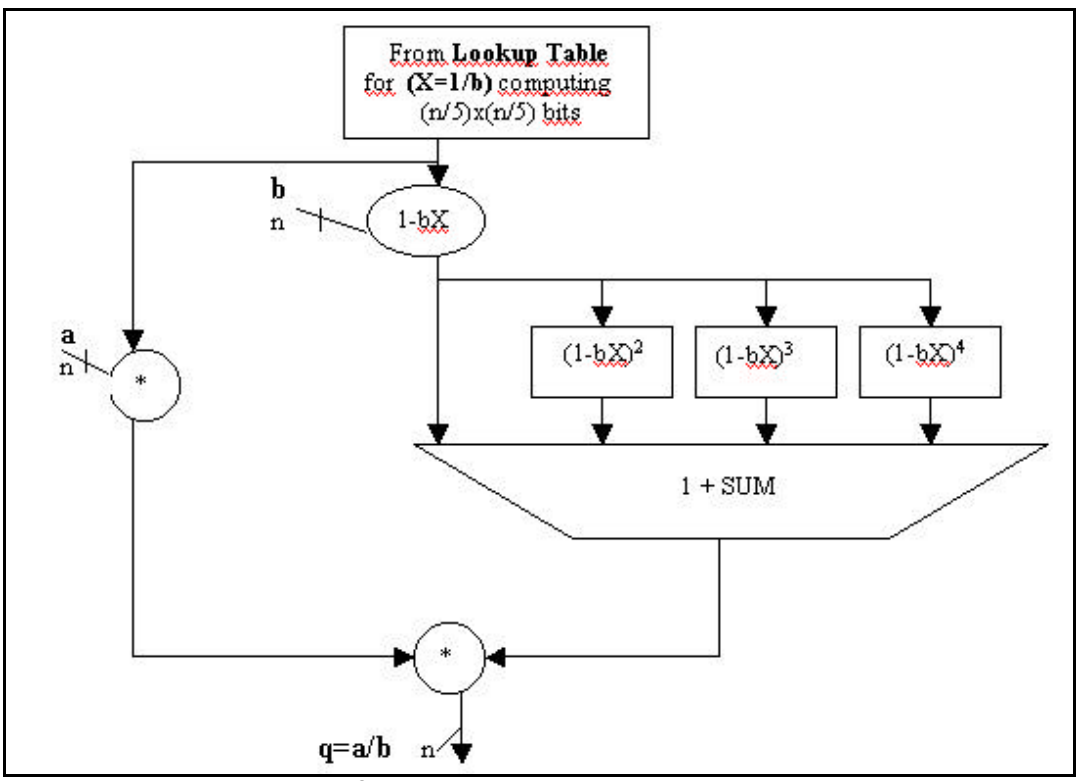

Fig.3. Proposed fourth-order Newton Raphson Division Hardware

Thus, the (1-bX) computation produces positive values and it can be implemented as the fused multiply-subtract. The partial product array of the (1-bX) multiply can be reduced by using Wallace tree technique . Therefore, the area required to implement the partial product array is approximately 30 % the size of the direct multiplication technique.

$(1\text{-}bX)^2$ is computed by squaring the result of the former computation. Figure 4 shows the the squaring unit partial product array reduction for 4-bit operand. The matched terms in the partial product array are grouped and moved one higher value position to the left in the reduced partial product array by using $a_ia_j + a_j a_i = 2*a_ia_j$ and $a_ia_i = a_i$ equivalences.

$(1\text{-}bX)^3$ is computed by cubing of the q=(1-bX). The parallel cubing unit proposed by [2] computes the cube of an operand with 24 bit length. This computation is 25% faster than the direct multiplication technique. Figure 5 indicates the partial product terms and reduced terms required to compute the parallel cube. In this computation, three different reduction techniques are applied to the cubing unit partial product array. The first reduction technique is performed on the three identical tems ($a_ia_ia_i$) of the partial product array. This term is replaced with a single term a_i. The second reduction technique is applied to the partial product terms which include two identical terms. The three terms with two identical bits are replaced by two terms with a weighting of 3 as follow:

$$a_ia_ia_j + a_ja_ia_i + a_ia_ja_i = 3*a_ja_i$$

The third reduction technique is applied to the partial product terms that include three different terms. The six terms on the partial product array are replaced with one three-different term with a weighting of 6 as follow:

$$a_ka_ia_j + a_ka_ja_i + a_ja_ka_i + a_ja_ia_k + a_ia_ka_j + a_ia_ja_k = 6*a_ka_ja_i$$

As a similiar process, $(1\text{-}bX)^4$ is computed as the result of fourth-order of the q=(1-bX).

The parallel unit proposed by [4] computes the fourth exponent of an operand with 24 bit length. Figure 6 shows the partial product terms and reduced terms required to compute the parallel process.

Like the cubing reduction techniques, the four identical , three identical, two identical and four different terms are replaced by one single term, one two term with a weighting of 4, one three term with a weighting of 12, one four term with a weighting of 24 respectively. Additionaly, these reduced terms are shifted two-bit for the two term and the three term with a weighting of 3, three-bit for the four term with a weighting of 3 to the left.

The multiplication of (a*X) is a small multiplication. An efficient multiplier can be used to compute the effect of this term to the result.

The final multiplier computes the result of the (a*X) multiplication and the sum of the (1-b*X), $(1\text{-}b*X)^2$, $(1\text{-}b*X)^3$, $(1\text{-}b*X)^4$ terms. This multiplication is the only full precision multiplication.

6. CONCLUSIONS

A fast and efficient divider unit is proposed in this study. To compute the reciprocal term (1/b), the unit utilizes the fourth-order Newton-Raphson iteration method. Parallel computation units are used to reduce the latency of the division operation. In the computations, a 24-bit operand lenght known as IEEE format is selected.

The latency of the proposed divider is less than the latency of the classic Newton-Raphson Method. Also, to compute the higher-order terms the truncating technique can be used. Thus, the needed hardwire can be significantly reduced.

				a_3	a_2	a_1	a_0
X				a_3	a_2	a_1	a_0
				$a_3\,a_0$	$a_2\,a_0$	$a_1\,a_0$	a_0a_0
			$a_3\,a_1$	$a_2\,a_1$	$a_1\,a_1$	a_0a_1	
	PPA	a_3a_2	$a_2\,a_2$	$a_1\,a_2$	a_0a_2		
	$a_3\,a_3$	a_2a_3	$a_1\,a_3$	a_0a_3			
	a_3a_2	$a_3\,a_1$	$a_3\,a_0$	$a_2\,a_0$	$a_1\,a_0$	_	a_0
RPPA	a_3		$a_2\,a_1$		a_1		
			a_2				

Figure 4. Squaring Computational Unit

						a_3	a_2	a_1	a_0
						a_3	a_2	a_1	a_0
X						a_3	a_2	a_1	a_0
					$a_3a_0a_0$	$a_2a_0a_0$	$a_1a_0a_0$	$a_0a_0a_0$	
				$a_3a_0a_1$	$a_2a_0a_1$	$a_1a_0a_1$	$a_0a_0a_1$		
				$a_3a_1a_0$	$a_2a_1a_0$	$a_1a_1a_0$	$a_0a_1a_0$		
			$a_3a_2a_0$	$a_2a_2a_0$	$a_1a_2a_0$	$a_0a_2a_0$			
			$a_3a_1a_1$	$a_2a_1a_1$	$a_1a_1a_1$	$a_0a_1a_1$			
			$a_3a_0a_2$	$a_2a_0a_2$	$a_1a_0a_2$	$a_0a_0a_2$			
	PPA	$a_3a_3a_0$	$a_2a_3a_0$	$a_1a_3a_0$	$a_0a_3a_0$				
		$a_3a_2a_1$	$a_2a_2a_1$	$a_1a_2a_1$	$a_0a_2a_1$				
		$a_3a_1a_2$	$a_2a_1a_2$	$a_1a_1a_2$	$a_0a_1a_2$				
		$a_3a_0a_3$	$a_2a_0a_3$	$a_1a_0a_3$	$a_0a_0a_3$				
	$a_3a_3a_1$	$a_2a_3a_1$	$a_1a_3a_1$	$a_0a_3a_1$					
	$a_3a_2a_2$	$a_2a_2a_2$	$a_1a_2a_2$	$a_0a_2a_2$					
	$a_3a_1a_3$	$a_2a_1a_3$	$a_1a_1a_3$	$a_0a_1a_3$					
$a_3a_3a_2$	$a_2a_3a_2$	$a_1a_3a_2$	$a_0a_3a_2$						
$a_3a_2a_3$	$a_2a_2a_3$	$a_1a_2a_3$	$a_0a_2a_3$						
$a_3a_3a_3$	$a_2a_3a_3$	$a_1a_3a_3$	$a_0a_3a_3$						
X1	a_3	-	-	a_2	-	-	a_1	-	-	a_0
		a_3a_2	a_3a_1	a_3a_0	a_3a_1	a_2a_0	a_3a_0	a_2a_0	a_1a_0	
X3	*RPPA*		a_3a_2	$a_3a_2a_0$	a_2a_1	a_2a_1		a_1a_0		
			$a_3a_2a_1$		$a_3a_1a_0$	$a_2a_1a_0$				

Figure 5. Cubing Computational Unit

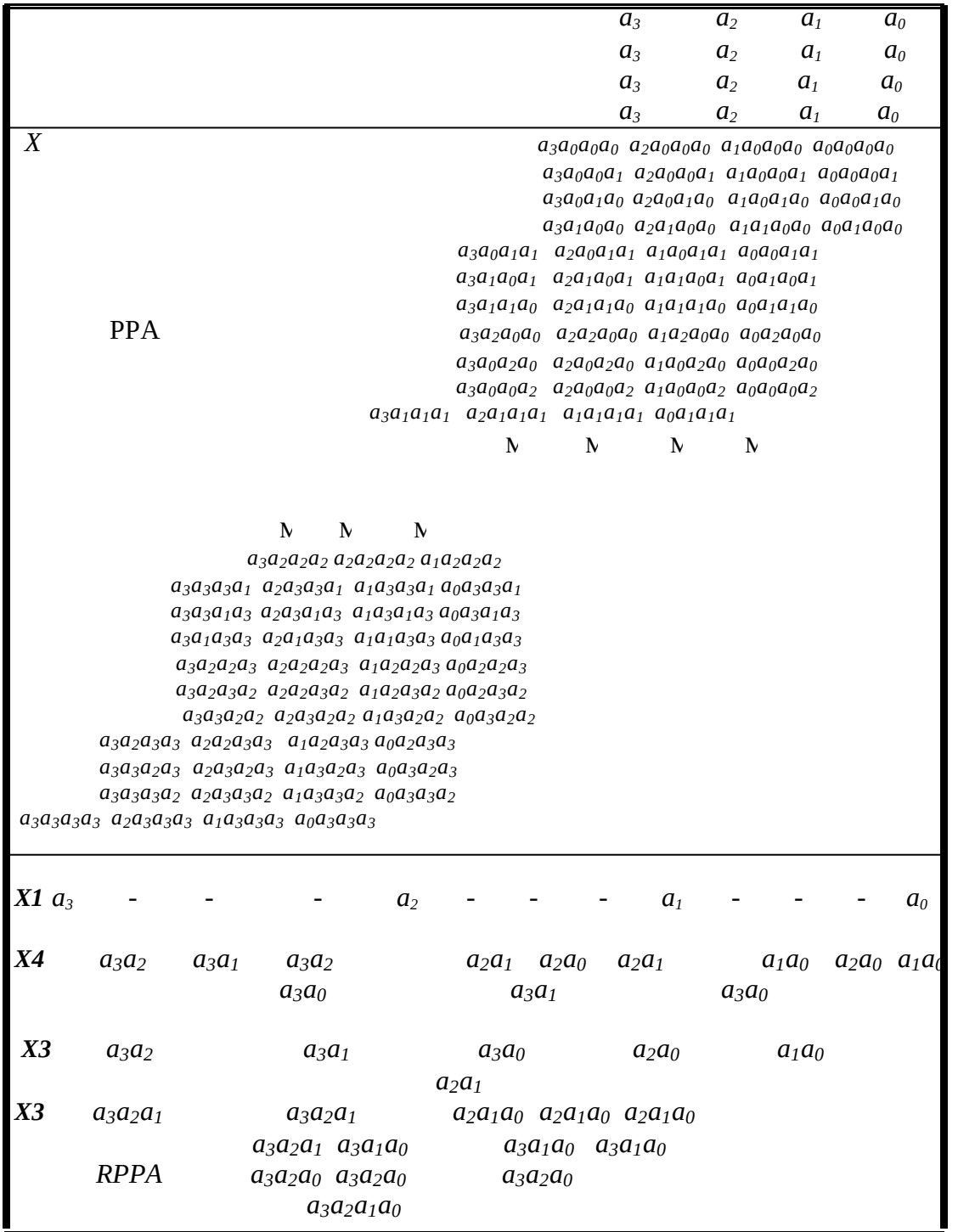

Figure 6. 4. Degree Computational Unit

REFERENCES

[1] **Flynn M.,** ' On division by Functional Iteration.', IEEE Transactions on Computers, Volume C-19, pages 702-706, August 1970.

[2] **Liddicoat A.** and **Flynn M**, (2000), 'Parallel Square and Cube Computations',Asilomar Conference on Signals, Systems and Computers, California, November.

[3] **Liddicoat A.** and **Flynn M.**, (2000), 'Pipelinable Division Unit', CSL-TR-00-809, The Technical Report, Computer System Labrotary, Stanford University, September.

[4] **Sertbaº A.**, 'Fast and Efficient Parallel Computation Units for Exponential Functions', ISCIS XVI International Symp.on Computer and Information Systems', Antalya-Belek, November 5-9, 2001.

Ahmet Sertbaº was born in Ýstanbul in 1965. He received the B.S. and M.Sc. degrees in electronic engineering from the Istanbul Technical University in 1990, the Ph.D. degree in electronic department from Istanbul University in 1997 respectively.

He has worked as Research Assistant at I.T.U. during 1987-1990, research engineer at Grundýg firm during 1990-1992, an instructor at the Vocational School of Istanbul University during 1993-1999. He is currently an Assoc. Professor in the Department of Computer Engineering at the University of Istanbul. His research interests include computer arithmetic circuit design, computer architecture and computer-aided circuit design, circuit theory and applications.