Implementation of Adaptive Sample Rate Kwan-Martin Notch Filter Using Efficient Realizations of Reciprocal and Squaring Circuit

Richard H. Strandberg, Jean-Claude Le Duc*, Luis G. Bustamante,
Vojin G. Oklobdzija and Michael A. Soderstrand
Department of Electrical and Computer Engineering
University of California
Davis, CA 95616

Abstract

Adaptive algorithms often involve the use of divide and square operations. Rather than implementing generic dividers and multipliers in the adaptive section for the adaptive sample rate notch filter, we have developed prototype architectures. We present a technique of optimizing an implementation of the squarer. In the reciprocal circuit presented, we show how a numerator (scaling) can be easily incorporated into its design to achieve a general purpose divider. Both realizations have been demonstrated in an adaptive sample rate notch filter using Xilinx reconfigurable Field Programmable Gate Arrays (FPGA). A comparison is made between generic circuits and the proposed techniques. The results provide a measure of the advantages of the proposed approaches for use in adaptive signal processing.

1 Introduction

A significant problem in the mobile communications area is narrowband interference that cause difficulties in decoding spread-spectrum BPSK signals. It has been shown that a simple canonical form of second order IIR notch filter can be adapted to eliminate narrowband interference from broadband communication signals [1]. Each notch stage is of the form:

$$H_{N_i}(z) = 1 - H_{BPF_i}(z) \quad (1)$$

$$H_{BPF_i}(z) = \frac{1-r_i^2}{2}\frac{1-z^{-2}}{1-2r_i cos\theta_i z^{-1}+r_i^2 z^{-2}} \quad (2)$$

where: r_i is the pole radius of the i-th section, is $\theta_i = 2\pi\frac{\omega_i}{\omega_s}$ is the angle of the poles on the unit circle where ω_i is the center frequency of the i-th section and ω_s is the sampling frequency. Instead of using conventional adaptation where the filter coefficients are updated, we use a fixed bandpass filter and adapt the sample rate [2]. Our proposed adaptive sample rate (ASR) notch cascadable section [3] is shown in Figure 1. $T(t)$ is the output of the last stage of the cascade. The presence of an additional bandpass filter and sensitivity filter, $H_{s_i}(z)$, is required to implement each cascade with an overall efficient architecture and $\frac{dT}{dT_{s_i}}$ is the instantaneous LMS gradient of the i-th section.

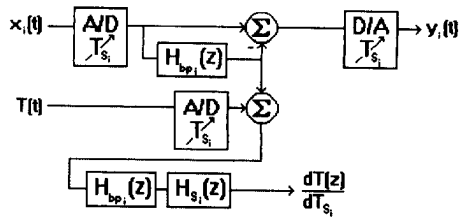

Figure 1: Cascadable ASR notch filter section.

By selecting certain fixed values of r_i and θ_i, we can arrive at greatly simplified implementations of $H_{BP_i}(z)$. For example, if we choose a value of $r_i = \sqrt{\frac{7}{8}}$, then the z^{-2} term of the denominator of will be $\frac{7}{8}$. The $\frac{7}{8}$ term can be implemented by subtracting one-eighth of the term from itself. We can accomplish the $\frac{1}{8}$ factor by simply shifting the bits 3 places. By setting $\theta_i = \frac{\pi}{2}$, the z^{-1} term of $H_{BP_i}(z)$ is zero making a multiplier and accumulator unnecessary. Finally, we implement the scale factor $\frac{1-r_i^2}{2} = \frac{1}{16}$ by simply shifting the bits 4 places. The resulting direct realization of the bandpass filter contains no multipliers and is shown in Figure 2.

*Ecole Supérieure d'Ingénieur en Electrotechnique et Electronique, 93162 Noisy le Grand CEDEX FRANCE

1058-6393/95 $4.00 © 1995 IEEE

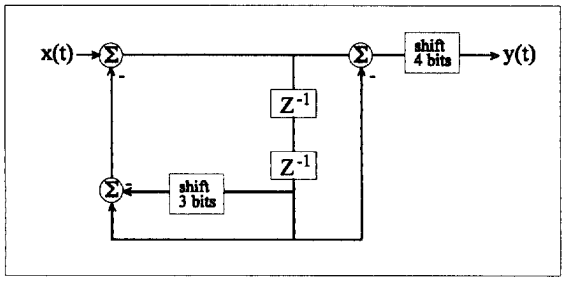

Figure 2: Implementation of bandpass filter.

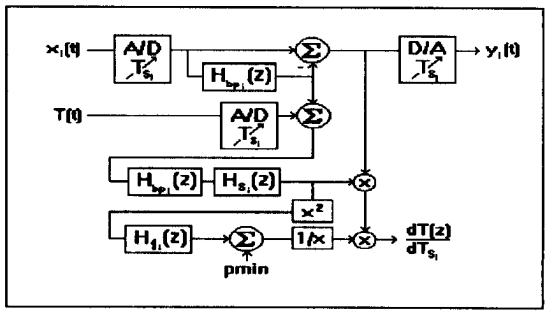

Figure 3: ASR cascadable section with gradient limiting.

Sensitivity filters are also second order IIR filters with simple implementations. The sensitivity filters are described by Equation 3.

$$H_{s_i}(z) = \frac{2r_i z^{-1}}{1 - 2r_i cos\theta_i z^{-1} + r_i^2 z^{-2}} \tag{3}$$

Using instantaneous values of the gradient does not result in a sufficiently robust algorithm. We therefore add a first order forgetting filter, $H_{f_i}(z)$ to the output of the sensitivity filters. The first order forgetting filter is described in Equation 4.

$$H_{f_i}(z) = \frac{1 - r_f}{1 - r_f z^{-1}} \tag{4}$$

where r_f is the radius of the forgetting filter. Setting $r_f = \frac{3}{4}$ or $\frac{7}{8}$ allows us to implement $H_{f_i}(z)$ with efficient realizations like the filters mentioned before. Adding a reciprocal circuit to limit the range of the gradient speeds up convergence of the algorithm. Both gradient-limiting circuit and forgetting filter are shown in our proposed adaptive sample rate notch cascadable section in Figure 3. Although we've shown so far that our technique enables us to implement hardware-efficient digital filters, the gradient-limiting option with forgetting filter increases our requirements by four potentially large circuits: two multipliers, a squarer and reciprocal. Next we present our results for efficient realizations of the squarer and reciprocal.

2 Implementing Square Function

The square function with operand N can be implemented by a table-lookup technique. However, in our case this would require 256 entries of 16-bit data requiring 512 bytes of ROM. Though this might be an appropriate implementation elsewhere, in systems consisting of predetermined cells such as FPGAs where each cell uses entire logic block, such a technique might exhaust the entire capacity of the FPGA chip. Therefore, it may more economical to implement the square function in combinational logic. Although one could use an 8-bit by 8-bit parallel multiplier. Given our intent is to use the square function in several places, minimal hardware implementation is desired [4].

2.1 Method

The method for minimal hardware implementation of a square function is based on the following algorithm. Let us assume that we want to obtain a square of a two bit binary number

$$X = x_1 x_0 \tag{5}$$

We will treat X as a sum of two numbers such that:

$$X = a + b \tag{6}$$

where $a = x_1,0$ and $b = 0,x_0$ (and "," represents a concatenation operation). In such a case:

$$X^2 = a^2 + b^2 + 2ab \tag{7}$$

By substitution of a and b we obtain:

$$X^2 = x_1^2, 00 + 00, x_0^2 + (x_1 x_0), 00 \tag{8}$$

Given $x_i^2 = x_i$, X^2 becomes

$$X^2 = [x_1 + (x_1 x_0)], 0, x_0 \tag{9}$$

which is then

$$X^2 = (x_1 x_0), (x_1 \bar{x}_0), 0, x_0 \tag{10}$$

We will shortly see that the realization of this function is very simple and requires the use of only two gates. Extending this concept to a more elaborate case where X is an 8-bit number, as in our case, we have:

$$X = x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 \tag{11}$$

where:

$$
\begin{aligned}
X^2 &= x_7^2 + x_6^2 + x_5^2 + x_4^2 + x_3^2 + x_2^2 + x_1^2 + x_0^2 \\
&\quad +2\{x_7x_6 + x_7x_5 + x_7x_4 + x_7x_3 + x_7x_2 \\
&\quad +x_7x_1 + x_7x_0 + x_6x_5 + x_6x_4 + x_6x_3 \\
&\quad +x_6x_2 + x_6x_1 + x_6x_0 + x_5x_4 + x_5x_3 \\
&\quad +x_5x_2 + x_5x_1 + x_5x_0 + x_4x_3 + x_4x_2 \\
&\quad +x_4x_1 + x_4x_0 + x_3x_2 + x_3x_1 + x_3x_0 \\
&\quad +x_2x_1 + x_2x_0 + x_1x_0\} \qquad (12)
\end{aligned}
$$

we can rewrite this expression as:

$$
\begin{aligned}
X^2 &= x_7^2 + x_6^2 + x_5^2 + x_4^2 + x_3^2 + x_2^2 + x_1^2 + x_0^2 \\
&\quad +2\{x_7(x_6 + x_5 + x_4 + x_3 + x_2 + x_1 + x_0) \\
&\quad +x_6(x_5 + x_4 + x_3 + x_2 + x_1 + x_0) \\
&\quad +x_5(x_4 + x_3 + x_2 + x_1 + x_0) \\
&\quad +x_4(x_3 + x_2 + x_1 + x_0) \\
&\quad +x_3(x_2 + x_1 + x_0) + x_2(x_1 + x_0) + x_1x_0\} \qquad (13)
\end{aligned}
$$

In binary form this becomes:

$$
\begin{aligned}
X^2 &= x_7, 0, x_6, 0, x_5, 0, x_4, 0, x_3, 0, x_2, 0, x_1, 0, x_0 + \\
&\quad (x_7x_6), (x_7x_5), (x_7x_4), (x_7x_3), (x_7x_2), \\
&\quad (x_7x_1), (x_7x_0), 0, 0, 0, 0, 0, 0, 0, 0 + \\
&\quad 0, 0, (x_6x_5)(x_6x_4)(x_6x_3)(x_6x_2)(x_6x_1)(x_6x_0), \\
&\quad 0, 0, 0, 0, 0, 0, 0 + \\
&\quad 0, 0, 0, 0, (x_5x_4)(x_5x_3)(x_5x_2)(x_5x_1)(x_5x_0), \\
&\quad 0, 0, 0, 0, 0, 0 + \\
&\quad 0, 0, 0, 0, 0, 0, (x_4x_3)(x_4x_2)(x_4x_1)(x_4x_0), 0, 0, 0, 0, 0 + \\
&\quad 0, 0, 0, 0, 0, 0, 0, 0, (x_3x_2)(x_3x_1)(x_3x_0), 0, 0, 0, 0 + \\
&\quad 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (x_2x_1)(x_2x_0), 0, 0, 0 + \\
&\quad 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (x_1x_0), 0, 0 \qquad (14)
\end{aligned}
$$

Let us show only the relevant bit positions as 1's. The corresponding single bit pattern is:

1		1		1		1		1		1		1		1
1	1	1	1	1	1	1	1	1	1	1	1	1		
		1	1	1	1	1	1	1	1	1				
				1	1	1	1	1						
						1								

where the first row represents the individual squares of the bits and the next four rows represent resulting product bits. We can further compress this pattern by moving bits up in the unfilled positions. The resulting bit pattern is given below:

1	1	1	1	1	1	1	1	1	1	1	1	1		1
1		1	1	1	1	1	1	1	1	1		1		
		1		1	1	1	1	1		1				
				1		1		1		1				
						1								

If we were to use a parallel multiplier to produce square function the corresponding partial product matrix would look like the one given below:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	
		1	1	1	1	1	1	1	1	1	1	1		
			1	1	1	1	1	1	1	1	1			
				1	1	1	1	1	1	1				
					1	1	1	1	1					
						1	1	1						
							1							

The depth of this partial product bit matrix is 8 versus 5 in our case. The summation of the partial product bits and reduction of this matrix to two operands would take 5 XOR gate equivalent delays in the later case, while this can be done in 4 equivalent XOR gate delays in our case. That means that the this scheme would yield a faster implementation.

2.2 Implementation

Implementation of squaring function is relatively simple and straightforward. Let us examine first a square block for two bits. The bit pattern is:

$$X^2 = (x_1x_0), (x_1\bar{x}_0), 0, x_0 \qquad (15)$$

To implement X^2 we need only two 2-input AND gates (assuming that the input bits are available in both polarities: true and complement) as shown in Figure 4.

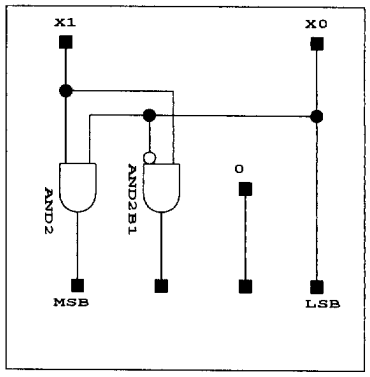

Figure 4: Implementation of two-bit X^2 function.

In case of 8-bit operands the circuit is more complex and it follows a design of a reduced parallel multiplier. However, it can be simplified further as shown in the steps to follow. Let us examine the partial product bit matrix given in the Table 1.

	x_7		x_6		x_5		x_4		x_3		x_2		x_1		x_0
	x_7x_6	x_7x_5	x_7x_4	x_7x_3	x_7x_2	x_7x_1	x_7x_0								
			x_6x_5	x_6x_4	x_6x_3	x_6x_3	x_6x_2	x_6x_1	x_6x_0						
					x_5x_4	x_5x_3	x_5x_2	x_5x_1	x_5x_0						
							x_4x_3	x_4x_2	x_4x_1	x_4x_0					
									x_3x_2	x_3x_1	x_3x_0				
											x_2x_1	x_2x_0			
													x_1x_0		

Table 1: Partial product bit matrix resulting from an 8-bit X^2 function.

x_7x_6	$\bar{x}_7x_6$	x_6x_5	$\bar{x}_6x_5$	x_5x_4	$\bar{x}_5x_4$	x_4x_3	$\bar{x}_4x_3$	x_3x_2	$\bar{x}_3x_2$	x_2x_1	$\bar{x}_2x_1$		x_1		x_0
		x_7x_5	x_7x_4	x_7x_3	x_7x_2	x_7x_1	x_7x_0	x_6x_0	x_5x_0	x_4x_0	x_3x_0	x_2x_0	x_1x_0		
				x_6x_4	x_6x_3	x_6x_2	x_6x_1	x_5x_1	x_4x_1	x_3x_1					
						x_5x_3	x_5x_2	x_4x_2							

Table 2: Simplified product bit matrix resulting from an 8-bit X^2 function.

In the next step we have merged the terms where there are x_i and x_ix_j terms present in the same column. This results in reduction of column hight by one where the term $\bar{x}_ix_j$ replaces two terms, x_i and x_ix_j, and the term x_ix_j is added to the next column. The resulting simplification is shown in Table 2.

Further simplification using the same method would not reduce the number of entries in the column given that all of them (in the middle) are already full. From this point on we will use one row of 4:2 compressors [5]-[6] before we sum the partial products in the Carry Propagate Adder. The total delay for the partial product reduction is equivalent to 3 XOR gate delays. This is 40 percent improvement in speed over use of a parallel 8X8-bit multiplier.

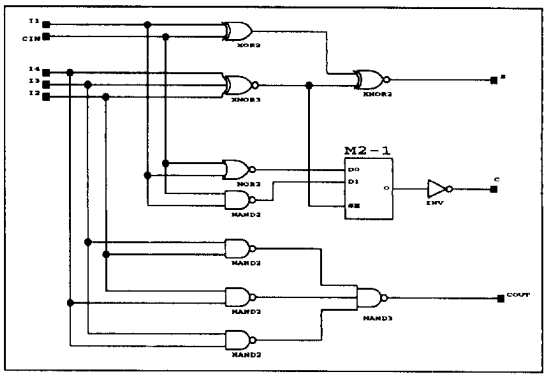

Figure 5: Schematic of compressor used.

To add the four rows of the simplified product bit matrix we have above, we could use just one row of (4:2) compressors. A (p, k) compressor is a counter with some of its inputs and outputs intentionally design for inter-connection such that they are not available for regular inputs and outputs. P represents the number of available inputs and k represents the number of available outputs. The design of the compressor we are using in our squaring circuit is shown in Figure 5.

Instead of using just one row of compressors we are going to mix full adders and compressors. We will by that way optimize the size of our circuit. We use the (4:2) compressor only when we have at least 3 bits per column in our matrix. When we need a 3-bit adder we will use a full adder. The result of this row of compressors and full adder will be two 16-bit numbers, S and C. To get the square we will have to add those two numbers with a classical 16-bit adder.

3 Reciprocal Circuit

As binary multiplication can be accomplished as a series of add and shift operations, reciprocation is done by a series of subtraction and shifts. Our design is a fixed point 8- bit (input) divisor with 10-bit dividend and quotient. First we will show what the dividend is for our case and then describe the operation of the circuit in general and by example.

Since we are using an analog to digital converter with

clock	d_6	d_5	d_4	d_3	d_2	d_1	$d_0 = q_9$	q_8	q_7	q_6	q_5	q_4	q_3	q_2	q_1	q_0
1	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1
2	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1	0
3	0	0	0	0	1	0	0	1	1	1	0	0	0	1	0	0
4	0	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0
5	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0
6	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0	0
7	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1
8	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1
9	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0
10	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1

Table 3: Register values of DREG and QREG for the example $\frac{271h}{19h} = 19\text{h}$.

input voltage range from -5.00 to +5.00 volts with 8-bit binary representation, the value of the voltage step size is $\frac{10}{2^8} = 0.0390625 \cong 0.04$ volts. The value of the step size is critical in determining the dividend. A hexadecimal input divisor of 19 (denoted 19h) represents 1.00 volt and our reciprocal circuit should produce an output of 19h. The dividend therefore needs to be 19h^2 or 271h.

Next we present the basic procedure of realizing the reciprocal. Instead of shifting the divisor from left to right as is done in division by hand, we shift the dividend from right to left. The algorithm is described in shorthand form as follows:

$DREG \Leftarrow 0$
$QREG \Leftarrow Dividend$
for QREG number of bits do
  if $DREG \leq Divisor$
    $DREG \Leftarrow SL(DREG)$
    $QREG \Leftarrow SL(QREG)$
  else
    $DREG \Leftarrow DREG - Divisor$
    $DREG \Leftarrow SL(DREG)$
    $QREG \Leftarrow SL(QREG)$

where the quotient register is denoted QREG, the dividend register is DREG, the least significant bit (LSB) of DREG equals the most significant bit (MSB) of QREG and SL(register) denotes a shift left operation.

Next we present an example of the division: $\frac{271h}{19h} =$ 19h. We obtain the following register values of DREG and QREG in Table 3. The MSB of DREG is denoted d_6 and the LSB is denoted d_0. A 7-bit divisor is sufficient for an 8-bit twos-complement number guaranteed to be positive. We obtain the final result at the eleventh clock cycle. Although we needed the dividend of 271h to implement our reciprocal, any 10-bit dividend could be easily loaded to implement a general-purpose division. Modifications are necessary to accomodate negative numbers.

4 Conclusion

An adaptive sample rate notch filter has been developed and implemented in hardware. Efficient realizations have been proposed for the following circuits: simple canonical bandpass filter, squaring and reciprocal.

References

[1] T. Kwan and K. Martin, "Adaptive detection and enhancement of multiple sinusoids using a cascade IIR filter," *IEEE Transactions on Circuits and Systems*, vol. 36, pp. 937–945, July 1989.

[2] R. Strandberg, M. Soderstrand, and H. Loomis, "Elimination of narrow-band interference using adaptive sampling rate notch filters," in *Proceedings IEEE Asilomar Conference on Circuits, Systems and Computers*, (Pacific Grove, CA), pp. 861–865, Nov. 1992.

[3] M. Soderstrand, H. Loomis, and K. Rangarao, "Elimination of narrow-band interference in BPSK-modulated signal reception," in *Proceedings IEEE Internationl Symposium on Circuits and Systems*, (Singapore), pp. 2798–2801, June 1991.

[4] R. H. Strandberg, J.-C. Le Duc, Z.-Y. Yang, L. G. Bustamante, V. G. Oklobdzija, and M. A. Soderstrand, "Reconfigurable processor for real-time adaptive sample rate notch filtering," in *Proceedings IEEE Asilomar Conference on Circuits, Systems and Computers*, (Pacific Grove, CA), Oct. 1994.

[5] V. Oklobdzija and D. Villeger, "Multiplier design utilizing improved column compression tree and optimized final adder in cmos technology," in *Proceedings of the 10th Anniversary 1993 International Symposium on VLSI Technology, Systems and Applications*, (Taipei, Taiwan), May 1993.

[6] V. Oklobdzija and E. Barnes, "On implementing addition in vlsi technology," in *IEEE Journal of Parallel Processing and Distributed Computing*, no. 8, 1988.