A NOVEL MULTIPLIERLESS HARDWARE IMPLEMENTATION METHOD FOR ADAPTIVE FILTER COEFFICIENTS

Yunhua Wang[1], Linda S. DeBrunner[2], Dayong Zhou[1], Victor E. DeBrunner[2],

[1]School of Electrical & Computer Engineering
University of Oklahoma
Norman, OK 73019 USA
{xiao9, dayong}@ou.edu

[2]Department of Electrical & Computer Engineering
FAMU-FSU College of Engineering
Tallahassee, FL 32310 USA
{linda.debrunner, victor.debrunner}@eng.fsu.edu

ABSTRACT

Adaptive filter implementations require real-time conversion of coefficients to Canonical Signed Digit (CSD) or similar representations to benefit from multiplierless techniques for implementing filters. Multiplierless approaches are used to reduce the hardware and increase the throughput. This paper introduces a novel hardware implementation method that converts two's complement numbers to their CSD representations using a fixed number of shift and logic operations. As a result, we can greatly reduce the power consumption and area requirements for hardware implementation of DSP algorithms in which coefficients are not known *a priori.* Because all CSD digits are produced simultaneously, the conversion speed and thus the throughput are improved when compared to overlap-and-scan techniques such as Booth's recoding.

Index Terms—adaptive filters, field programmable gate arrays, digital arithmetic.

1. INTRODUCTION

"Implementation is everything" in the construction of practical adaptive filters [13]. These practical hardware implementations typically require high throughput, low power consumption and small area. For fixed coefficient filters, multiplierless implementation approaches are used. However, since the coefficients of an adaptive filter are not fixed, general multipliers are needed. Multipliers are expensive in terms of chip area, power consumption, and operation time. For practical high performance adaptive filters, this limitation must be overcome.

Multipliers are typically implemented in hardware using shift-and-add techniques. The number of add operations depends on the number of 1's in the binary multiplier. The number of add/shift operations is directly related to the power consumption and area required. Array techniques are used to achieve high throughput, at the cost of significant increases in power and area.

One effective method to reduce the number of shift/add operations in multiplier hardware is to reduce the wordlength of the multipliers (e.g. filter coefficients). However, reducing the wordlength can significantly degrade the performance of the implemented algorithm.

When the value of the multiplier is known, multiplication can be implemented using alternate number representations for the multiplier, such as the canonical signed digit (CSD) number system [10] or signed power-of-two (SPT) representation [1]. CSD representation [3] is a radix-two number system with digit set $\{-1,0,1\}$ that has the "canonical" property that no two consecutive bits in the CSD number are nonzero. For example, the 2's complement number $x = 10101101 = 0\bar{1}0\bar{1}0\bar{1}01$, where "$\bar{1}$" stands for "-1". This representation replaces the additions arising from a string of ones in a binary number with a single subtraction, so that a multiplier can be realized by incorporating a few adders (or subtractors) and bit shifters.

CSD representation has proven to be useful for implementing multipliers with less complexity, because the cost of multiplication is a direct function of the number of nonzero bits in the multiplier. It is shown in [4] that for a n-bit 2's complement multiplier the number of add/subtract operations never exceeds n/2 and can be reduced to n/3 on average, as the wordlength of multiplier grows. Many researchers have addressed the question of how to convert 2's complement to CSD numbers. Some of these approaches are from the point of view of reducing computational complexity [5][6], but are not suitable for implementation into hardware. Other approaches try to improve the implementation efficiency by limiting the area and power consumption [7][8]. However, some introduce errors, and others are still complex.

If the multiplier is known *a priori,* as is the case for most filter implementations, the CSD expression can be calculated offline and the implementation can be further improved via computational techniques such as Dempster-Macleod's algorithm [9]. Using this technique, more adders can be saved [10]. However, when the multiplier is unknown or can change over time, as is the case for adaptive filters, these techniques are not applicable. To benefit from the CSD implementation advantages, the conversion of numbers from 2's complement to CSD format must be implemented in hardware. Unfortunately, the cost of conversion using methods such as those based on Look-Up-Table (LUT) [11] or Booth's recoding techniques [1] often outweighs the implementation advantages of CSD.

In this paper, we introduce a new hardware implementation method to convert 2's complement numbers to CSD numbers. Our method has several advantages. First, unlike LUT methods, our technique does not require a fixed word length to be known *a priori.* In addition, our proposed method uses a limited number of shift and logic operations, instead of the overlap and scanning used for methods like Booth's recording. This allows the number of computational cycles to be fixed and independent of the wordlength of the multiplier, k. So, the time required is constant. Furthermore, because all the CSD bits are produced

simultaneously, the conversion speed, and thus the throughput, is improved.

Our method can be applied to efficiently implement digital filters with non-fixed coefficients, such as adaptive filters. The implementation can be further improved through the use of parallel processing with a reasonable sacrifice in the area consumption using FPGAs.

This paper is organized as follows. Section 2 describes the new method to convert 2's complement numbers to CSD numbers. Then, we compare our method with Booth's recording and LUT techniques in section 3; Conclusions and future work are presented in section 4.

2. NEW TWO'S COMPLEMENT TO CSD CONVERSION METHOD

Our method to convert a 2's complement number to CSD representation is a simple series of shift and logic operations, which are implemented in six processing steps as shown in Fig. 1 and described in the following paragraphs.

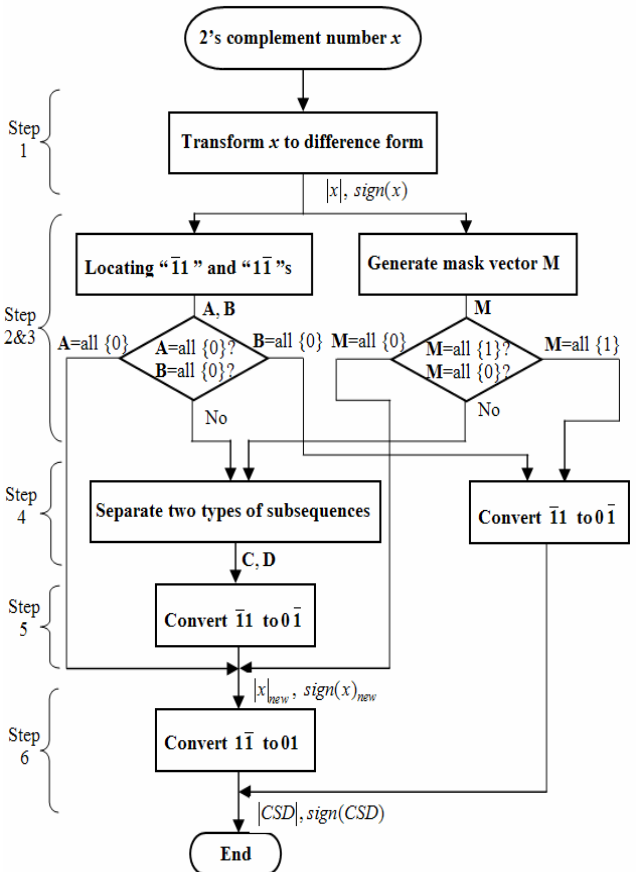

Fig. 1 Block diagram for new 2's complement to CSD conversion method

Step 1: Transform x to difference form: $x = 2x - x$. In order to replace the additions arising from a string of ones in a binary number with a single subtraction, we use the simple concept $x = 2x - x$ to convert x to another form, which contains pairs of "$1\bar{1}$" and "11"s. The difference form is a signed binary representation that can be written as of two binary numbers: the magnitude of x and the sign of x, which together represent the signed binary number. The ones in $sign(x)$ indicate which digit positions have a negative weight. This form can be computed simply with an arithmetic left by one bit $x << 1$ and bitwise logic operations:

Magnitude of x : $|x| = x << 1 \oplus x$

Sign of x : $sign(x) = (x << 1)' \& x$

We can prove the following:

Theorem 1: No two consecutive nonzero bits in the difference form of x have the same sign.

Theorem 2: The difference form of x can be converted to the CSD form by replacing the sequences "$1\bar{1}$" with "01" and the sequence "11" with "01".

Step 2: Locating "$1\bar{1}$" and "11"s. To locate the positions of the "$1\bar{1}$" and "11" strings, we find the digits that are '$\bar{1}$' from the '1's in $sign(x)$, then use "shift/and" operation to get two vectors **A** and **B**.

A= $|x| << 1 \& sign(x)$

where each '1' in **A** corresponds to a string "$1\bar{1}$".

B= $|x| >> 1 \& sign(x)$

where each '1' in **B** corresponds to a string "11".

Note that $|x| >> 1$ denotes a logical right shift by one bit.

Theorem 3: Each '1' in **A** denotes the position of a "$1\bar{1}$" string in the difference form, and each '1' in **B** corresponds to a string "11" in the difference form.

Lemma 3A: There are no consecutive '1's in **A** or **B**.

Step 3: Generate mask vector M. (Note that steps 2 and 3 can be computed concurrently.) Step 2 replaces strings of ones with pairs of"$1\bar{1}$"s and "11"s. To achieve a CSD representation, we want to replace the strings "$1\bar{1}$" with "01" and "11" with "01" to eliminate consecutive nonzero bits. However, we cannot do both "$1\bar{1}$" to "01" and "11" to "01" transformations at the same time using simple logic operations; also, we cannot do the two operations sequentially. For example, if "$11\bar{1}$" and "$1\bar{1}1$" exist in the same sequence, no matter which replacement we do first the result has consecutive nonzero bits, such as "011" or "$01\bar{1}$".

So we have to find a way out. This leads to theorem 4 as follows:

Theorem 4: The zero bits in the difference form of x correspond to zero digits in the CSD form.

Based on Theorem 4, we observe that the zeros in the difference form of x separate the sequence into several parts. We want to transform "$1\bar{1}$" to "01" and "11" to "01" separately beginning with the nonzero bit adjacent to the '0' (working from right to left). We form a mask vector **M** to separate the subsequences. **M** has the same length as x. Whenever the subsequence begins with '1', the corresponding subsequence in **M** is all ones, otherwise it is all zeros. For example, if $x = 01\bar{1}001\bar{1}01\bar{1}1\bar{1}$, then **M** = 01100110000.

Fig. 2 shows the hardware implementation of mask generator, where

$$\mathbf{M}_i = |x|_i \; sign(x)_i' \; |x|_{i-1}' + |x|_i \; \mathbf{M}_{i-1} \; |x|_{i-1} \qquad (1)$$

Step 4: Separate two types of subsequences. Using **C= A** & **M** we determine the subsequence "$1\bar{1}$"s since each '1' in **C** stands for the pair "$1\bar{1}$", at the corresponding position of '$\bar{1}$'. Note that there are no consecutive '1's in **C** because of the inherited property of **A**. Similarly, using **D= B**& ~ **M** , we can get determine the location of the "$1\bar{1}$" sequences. Also, there are no consecutive '1's in **D.**

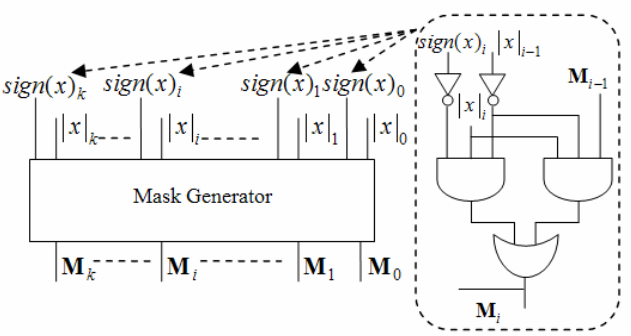

Fig. 2. The implementation of the mask generator

Step 5: Convert $\bar{1}1$ to $0\bar{1}$. We use **C** to do the convert the substrings "$1\bar{1}$" to "$0\bar{1}$" as follows:

$$|x|_{new} = |x| \oplus \mathbf{C}$$
$$sign(x)_{new} = sign(x) \oplus \mathbf{C} \,|\, (\mathbf{C} >> 1) \qquad (2)$$

Step 6: Convert $1\bar{1}$ to 01 . Similar to Step 5, we convert "$1\bar{1}$" to "01" using **D** as follows:

$$sign(CSD) = sign(x)_{new} \oplus \mathbf{D}$$
$$|CSD| = (\mathbf{D} << 1) \oplus |x|_{new} \qquad (3)$$

Theorem 5: The algorithm described in Steps 1-6 generates the CSD representation of a number.

Example: Fig. 3 shows the conversion of *x=101110110* to CSD.

3. COMPARISION WITH BOOTH RECODING AND LUT TECHNIQUES

The radix-4 modified Booth recoding algorithm has been widely used in modern high-speed multiplication circuits. Using a modified Booth algorithm, sequential 3-digit segments of two's complement number are converted into the digit set $\{\pm 2, \pm 1, 0\}$. Although modified Booth's recoding reduces a *k*-bit 2's complement multiplier to $\lceil k/2 \rceil$ digits, it is based on overlap multiple-bit scanning schemes. So, no matter how large the radix is, the number of scan cycles is a function of the multiplier word length *k*. As *k* increases, the number of scan cycles is increases as well.

Our proposed method reduces the number of add/subtract operations to the minimum. Unlike the modified Booth recoding algorithm, the number of operations of our method is fixed. So, the total delay time is also fixed. So, the time is constant regardless of the word length *k*. The detailed performance analysis is given in Table 1.

Compared with the modified Booth recoding algorithm whose operation time is a function of multiplier word length *k*, our method only requires a delay of only 4 shifts and 8 logic gates for the worst case. Furthermore, the throughput can be further improved by incorporating parallel processing. Our method offers is attractive in terms of both throughput and computational complexity.

x	1 0 1 1 1 0 1 1 0 1 0 1	
$2x = x << 1$	1 0 1 1 1 0 1 1 0 1 0 1 0	
$-x$	1 1 0 1 1 1 0 1 1 0 1 0 1	sign extension
x	0 $\bar{1}$ 1 0 0 $\bar{1}$ 1 0 $\bar{1}$ 1 $\bar{1}$ 1 $\bar{1}$	
$\|x\|$	0 1 1 0 0 1 1 0 1 1 1 1 1	
$sign(x)$	0 1 0 0 0 1 0 0 1 0 1 0 1	
$\|x\| << 1$	1 1 0 0 1 1 0 1 1 1 1 1 0	
$\& sign(x)$	0 1 0 0 0 1 0 0 1 0 1 0 1	
A	0 1 0 0 0 1 0 0 1 0 1 0 0	
$\|x\| >> 1$	0 0 1 1 0 0 1 1 0 1 1 1 1	
$\& sign(x)$	0 1 0 0 0 1 0 0 1 0 1 0 1	
B	0 0 0 0 0 0 0 0 0 0 1 0 1	
M	0 1 1 0 0 1 1 0 0 0 0 0 0	
&A	0 1 0 0 0 1 0 0 1 0 1 0 0	
C	0 1 0 0 0 1 0 0 0 0 0 0 0	
B	0 0 0 0 0 0 0 0 0 0 1 0 1	
& ~ M	1 0 0 1 1 0 0 1 1 1 1 1 1	
D	0 0 0 0 0 0 0 0 0 0 1 0 1	
$\|x\|$	0 1 1 0 0 1 1 0 1 1 1 1 1	
⊕ C	0 1 0 0 0 1 0 0 0 0 0 0 0	
$\|x\|_{new}$	0 0 1 0 0 0 1 0 1 1 1 1 1	
$sign(x)$	0 1 0 0 0 1 0 0 1 0 1 0 1	
⊕ C	0 1 0 0 0 1 0 0 0 0 0 0 0	
	0 0 0 0 0 0 0 0 1 0 1 0 1	
$\|(\mathbf{C} >> 1)$	0 0 1 0 0 0 1 0 0 0 0 0 0	
$sign(x)_{new}$	0 0 1 0 0 0 1 0 1 0 1 0 1	
⊕ D	0 0 0 0 0 0 0 0 0 0 1 0 1	
$sign(CSD)$	0 0 1 0 0 0 1 0 1 0 0 0 0	
D << 1	0 0 0 0 0 0 0 0 0 1 0 1 0	
$\oplus \|x\|_{new}$	0 0 1 0 0 0 1 0 1 1 1 1 1	
$\|CSD\|$	0 0 1 0 0 0 1 0 1 0 1 0 1	
CSD	0 0 $\bar{1}$ 0 0 0 $\bar{1}$ 0 $\bar{1}$ 0 1 0 1	

Fig. 3. An example of new 2's compliment to CSD conversion process (the double dash line separate each step)

Another commonly used technique for FPGA-based hardware is Look-Up-Table (LUT) [10], [14]. Many algorithms used in DSP, such as filters, are based on constant coefficient values. So, a Look-Up-Table can be used to implement the multiplier by storing pre-computed partial products of the fixed coefficient in distributed ROM to reduce the logic content. An

advantage of this approach is that the delay is just a memory access; so it is fast. However, a disadvantage is that the table size grows exponentially with the input, so it is space-intensive. So, a LUT approach requires the multiplier's word length to be fixed and the value of multiplier to be known prior to implementation.

Table 1. Performance analysis of the new 2's to CSD conversion method (× indicates the most costful operations in each step, notice: Step 2 and Step 3 can be done simultaneously)

	Operations	# of Shifts	# of logic operations	A/M= all{0}	B=all{0}/ M=all{1}	The worst case
Step 1	$\lvert x\rvert = x << 1 \oplus x$	1	1			
	$sign(x) = (x << 1)' \& x$	1	2	×	×	×
Step 2	$\mathbf{A} = \lvert x\rvert << 1 \,\&\, sign(x)$	1	1	×	×	×
	$\mathbf{B} = \lvert x\rvert >> 1 \,\&\, sign(x)$	1	1			
Step 3	$\mathbf{M}_i = \lvert x\rvert_i \, sign(x)'_i \, \lvert x\rvert'_{i-1}$ $+ \lvert x\rvert_i \, \mathbf{M}_{i-1} \lvert x\rvert_{i-1}$		3			
Step 4	$\mathbf{C} = \mathbf{A} \,\&\, \mathbf{M}$		1			
	$\mathbf{D} = \mathbf{B} \,\&\, \sim \mathbf{M}$		2			×
Step 5	$\lvert x\rvert_{new} = \lvert x\rvert \oplus \mathbf{C}$		1			
	$sign(x)_{new} = sign(x) \oplus \mathbf{C} \lvert (\mathbf{C} >> 1)$	1	2		×	×
Step 6	$sign(CSD) = sign(x)_{new} \oplus \mathbf{D}$		1			
	$\lvert CSD\rvert = (\mathbf{D} << 1) \oplus \lvert x\rvert_{new}$	1	1	×		×
Total cost of delay				3 shifts + 4 logics	3 shifts + 5 logics	4 shifts + 8 logics

Our method does not have the disadvantages of the LUT implementation. We do not require a fixed multiplier word length, nor is it required for the multiplier value to be known *a priori*. Thus, our method can be applied to efficiently implement digital filters with non-fixed coefficients, such as adaptive filters. In addition, our method is simple, requiring only several shifts and logic operations. Since our method produces all of the CSD digits simultaneously, the conversion speed, and thus the throughput, is improved.

4. CONCLUSIONS AND FUTURE WORK

The implementation of adaptive filters cannot benefit from fast, low area filter design techniques that use *a priori* information about the filter coefficients. We propose a novel implementation technique that can be used to construct general multipliers which require less area and achieve higher throughput rates. Our method for converting a number from two's complement representation to CSD representation can be used to implement adaptive filters in FPGAs or other custom hardware. Performance analysis indicates that our design provides better results than are currently available considering both the conversion speed and the computational complexity. Since our technique does not require a specific word length for the multiplier and does not depend on prior knowledge of the multiplier value, it has broad applications. Our method only requires several shifts and logic operations, so we have effectively reduced the complexity of the hardware implementation compared to conventional methods, such as modified Booth recoding and Look-Up-Table based techniques. The throughput of our implementation can be further improved by incorporating parallel processing with only a modest increase in area. We believe that the proposed method will have broad applications in hardware implementations of many DSP algorithms and other multiplication intensive applications, but most especially the implementation of adaptive filters.

We plan to incorporate our method into a multi-input CSD multiplier unit, similar to that proposed in [12], which requires all the CSD digits to be converted simultaneously. This new multi-input CSD multiplier circuit will allow the construction of high throughput adaptive filters in FPGAs or other custom hardware under practical time, space and power constraints. Detailed area and power analysis can be assessed after completion of the implementation.

5. REFERENCES

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs.* London, Oxford Press, 1999.

[2] C. L. Chen, K. Y. Khoo, and A. N. Willson, Jr., "A simplified signed powers-of-two conversion for multiplierless adaptive filters," *ISCAS'96,* May 1996.

[3] P. Pirsch, *Architectures for Digital Signal Processing*, John Wiley & Sons, 1998.

[4] G. K. Ma and F. J. Taylor, "Multiplier policies for digital signal processing*," IEEE ASSP Mag.,* pp. 6-20, Jan 1990.

[5] F. Xu, C. Chang and C. Jong, "HWP: a new insight into canonical signed digit," ISCAS'04, May 2004.

[6] R. Hashemian, "A new method for conversion of a 2's complement to canonic signed digit number system and its representation," in *Proc. Asilomar Conf. Signals, Syst., Computers*, 1997, pp. 904-907.

[7] G.A. Ruiz and M.A. Manzano, "Self-Timed Multiplier Based on Canonical Signed-Digit Recoding," *IEE Proc., Circuits, Devices, and Systems,* vol. 148, no. 5, Oct. 2001, pp. 235-241.

[8] S. M. Kim, J. G. Chung, and K. K. Parhi, "Design of low error CSD fixed-width multiplier," in *Proc. 2002 IEEE ISCAS* Scottsdale, AZ, May 2002, pp. I-69-I-72.

[9] A. G. Dempster and M. D. Macleod, "Constant integer multiplication using minimum adders," *IEE Proceedings: Circuits, Devices and Systems*, vol. 141, pp. 407-413, 1994.

[10] M. A. Soderstrand, "CSD multipliers for FPGA DSP applications," *ISCAS'03*, May 2003.

[11] K. Chapman, "Building high performance FIR filter using KCM," *Xilinx Ltd-UK*, July 1996.

[12] Y. Wang, L. S. DeBrunner, V. E. DeBrunner, and D. Zhou, "A Multi-input multiplier unit suitable for DSP algorithm implementations," *Asilomar Conference on Signals, Systems and Computers*, Oct. 2006.

[13] John Treichler, plenary comments, *IEEE 2006 DSP Workshop,* Jackson, Wyoming, Sept. 2006.

[14] Bill Allaire and Bud Fischer, "Block adaptive filter," *Xilinx Application Note, XAPP 055,* version 1.1, January 1997.