DECISION FEEDBACK EQUALIZER WITH TWO'S COMPLEMENT COMPUTATION SHARING MULTIPLICATION

Hunsoo Choo, †Khurram Muhammad and Kaushik Roy

School of Electrical and Computer Engineering, Purdue University
West Lafayette, IN 47907, USA
†Texas Instruments Inc., Dallas, TX 75243
{chooh,kaushik}@ecn.purdue.edu, k-muhammad1@ti.com

ABSTRACT

We present an architecture of a high performance decision feedback equalizer based on a computation sharing multiplier. The computation sharing multiplier (CSHMR) uses a redundant number scheme and targets removal of computational redundancy by computation re-use. Use of CSHMR leads to high performance FIR filtering operation by re-using optimal precomputations. A decision feedback equalizer (DFE) implementation based on CSHMR in a 0.35 μ technology shows 34% improvement in the operating speed over DFE using Wallace tree multiplier.

1. INTRODUCTION

The decision feedback equalizer is composed of two linear filters (*feedback and feedforward filter*) and a non-linear decision unit. The order of the filter in the DFE depends on the channel characteristic and symbol rate. At a high symbol rate, high order FIR filter with faster processing speed is desired. Hence, computation complexity and speed of adaptation are critical considerations for a DFE. In our work, we focus on reducing the computation complexity of the filter used in the DFE. A *computation sharing multiplier* (CSHM) architecture, which identifies common computations and shares them between different multiplications was suggested in [1, 2]. We modified it so that CSHM can work with two's complement number and used carry-save optimization technique to improve its performance. The resulting multiplier is used to implement a DFE which is shown to have lower complexity and higher operating speed. CSHMR and DFE are modeled in VHDL and their layouts are obtained using auto placing and routing. Pathmill and powermill are used to get the simulation results.

The rest of this paper is organized as follows. In section 2, carry-save optimization technique is described. Section 3 explains the algorithm and the architecture of CSHMR. The implementation of DFE is presented in section 4. Sections 5 and 6 present the numerical results and conclusions, respectively.

2. CARRY-SAVE REDUNDANT NUMBER SCHEME

A straight-forward way to implement arithmetic functions is to compute the final result by propagating carry. Hence, in a sequence of such operations, carry is propagated at the end of every operation. We refer to this approach as *non-redundant number scheme* (NRS). Alternatively, a relaxed rule can be applied such that two numbers, the sum of which is equal to the final result, can be generated as two outputs of an arithmetic operation [3]. For example,

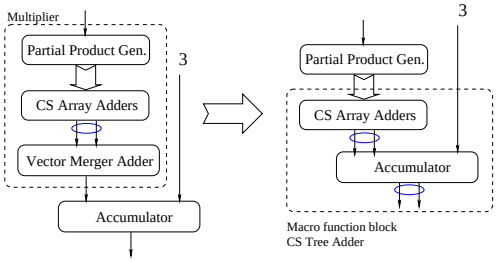

Fig. 1. Carry-sum redundant number scheme.

consider filtering operation where an accumulator follows a multiplier. The carry-save multiplier (CSM) is used for multiplication and CS adder is used as an accumulator.

The CSM can be decomposed into three function blocks: partial products generation, CS adder block and vector merger adder (VMA) [3]. Suppose the CS adder block adds all partial products and generates two signals representing 7 and 9 (see figure 1). The VMA adds these two final outputs using carry propagation to generate the end result 16 in NRS. If 3 is subsequently added, another carry propagation must complete to obtain the final result in NRS. In this approach, the delay of VMA contributes a large part of total delay, since a carry has to propagate from LSB to MSB in VMA. Alternatively, the three values may be added without propagating carry in the intermediate sum (of 7 and 9) which is kept as two outputs in NRS and to which the third number (3) is added. This removes a VMA delay from the overall operation. This principle is used in constructing array multipliers. We will refer to this scheme as *carry-save redundant number scheme* (CSRS).

3. MULTIPLIER ARCHITECTURE

3.1. Computation Sharing Multiplication

A set of bit sequences can be selected for pre-computation such that the result of a multiplication can be obtained by add and shift operations of these pre-computations. For instance, $(1011)\cdot x$ can decomposed as $(0011)\cdot x + 2^3\cdot(0001)\cdot x$. If both $(0011)x$ and x are available, the entire multiplication process is reduced to a few add and shift operations. We will refer to these chosen bit sequences as *alphabets*. When multiplying a vector by a scalar, one can define an *alphabet set* to be a set of *alphabets* that spans all the coefficients in the vector. A multiplier architecture which minimized the total precomputations for multiplying a vector by a scalar using optimal alphabets was introduced in [1, 2] which is referred to

as CSHM.

In a filter with coefficient vector **C**, the ith coefficient c_i can be decomposed as

$$c_i = \sum_{k=0}^{L} 2^{m_k} \alpha_{k,j} \quad (1)$$

where m_k is a shift value and $\alpha_{k,j}$ are $alphabets$ that belong to the jth *alphabet set*, for $k = 0, 1, 2, \ldots, L$. (Note that many alphabet sets can be composed). If we multiply the scalar x to both sides of equation (1), the multiplication $c_i \cdot x$ can be expressed as $c_i \cdot x = \sum_{k=0}^{M-1} 2^{m_k} \alpha_{k,j} \cdot x$. Hence, multiplication $c_i \cdot x$ can be significantly simplified to add and shift operations of $\alpha_{k,j} \cdot x$, which is multiplications of x and all the elements of the predetermined alphabets.

3.2. Multiplication Algorithm for two's complement

The advantage of CSHM is due to the shift operation of precomputed values to obtain partial products of different bit levels [1]. To use shift operation in two's complement multiplication as in sign magnitude multiplication, *Baugh-Wooley algorithm* can be used [4], [5].

The product of two two's complement numbers, is expressed as follows

$$\begin{aligned} P &= \left(-2^{N-1} a_{N-1} + \sum_{i=0}^{N-2} 2^i a_i\right)\left(-2^{N-1} b_{N-1} + \sum_{j=0}^{N-2} 2^j b_j\right) \\ &= \sum_{i=0}^{N-2}\sum_{j=0}^{N-2} 2^{i+j} a_i b_j + 2^{2N-2} a_{N-1} b_{N-1} + 2^N - 2^{2N-1} \\ &\quad + \left(\sum_{j=0}^{N-2} 2^j \overline{a_{N-1} b_j} + \sum_{i=0}^{N-2} 2^i \overline{b_{N-1} a_i}\right) 2^{N-1} \end{aligned} \quad (2)$$

Equation 2 shows that the multiplication of two's complement numbers can be written in a form which involves only positive bit products. Figure 2 shows how this algorithm works in the case of a 4x4 multiplication. The first three rows are referred to as

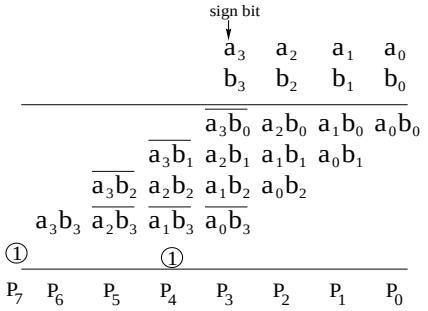

Fig. 2. Example of Baugh-Wooley algorithm: 4x4.

PM (*partial products with magnitude part*) and generated by one NAND and three AND operations. The fourth row is called as PS (*partial products with sign bit*) and generated by one AND and three NAND operations with a sign bit.

Consider the partial products of PM. Suppose $b_2 = b_0$ in figure 2. Then the third row can be obtained by shifting the first row by 2 bits. Likewise, shift operation can be used to obtain a partial product of different bit level as in sign magnitude multiplication.

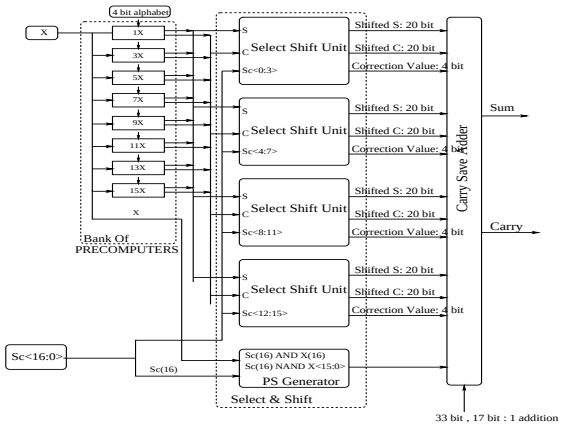

Fig. 3. Architecture of computational sharing multiplier using CSRS.

3.3. Computational Sharing Multiplier with Redundant Number Scheme

Figure 3 shows the basic architecture of the CSHMR which is composed of three sub-blocks: Precomputer, Select&Shift (S&S) and CS adder tree. For 17x17 multiplier implementation, the optimal alphabet set $\{1, 3, 5, 7, 9, 11, 13, 15\}$ [2] is used and each alphabet is represented as a 4 bit unsigned number.

Suppose we are computing $X \times Sc$. The precomputer generates the partial products which correspond to PM with input X and every alphabet, and adds them. The summation of all the partial products are represented in carry-save (CS) redundant number and stored in the precomputer for re-use. The S&S consists of four *Select/Shift unit*s (SSU) and PS generator. The scalar Sc comes in and is partitioned into smaller bit sequences having the same length as the alphabets excluding the sign bit. Depending on these bit sequences, each SSU selects proper precomputed values (C and S), and shifts them to generate the correct values corresponding to the partial products of PM of target multiplication. The SSU generate one more output: the *correction value*. For example, suppose we have already computed the PM of $\mathbf{a} \cdot \mathbf{b}$, where $\mathbf{b}$ is $b_3 b_2 b_1 b_0$ (figure 2), and want to compute the multiplication of $\mathbf{a} \cdot \mathbf{c}$, where $\mathbf{c}$ is $b_3 b_1 b_0 0$. According to the Baugh-Wooley algorithm, the PM of $\mathbf{a} \cdot \mathbf{c}$ can be obtained by shifting the PM of $\mathbf{a} \cdot \mathbf{b}$. However, additional value `1 0 0 0' has to be added to compute the correct result, which is generated by '0' at LSB. Hence, a correction term is required to compensate for this additional value. Table 3.3 shows the correction values based on shift signal.

Shift Signal	Correction Values <19:16>
00	0000
01	0001
10	0011
11	0111

Table 1.

The S&S also generates the PS using one AND and several NAND bit operations with a sign bit of scalar. Finally, all the outputs from S&S are generated. These are inserted into the carry-save adder which is the last stage of the multiplier. The additions of $2^N and - 2^{2N-1}$ of the Baugh-Wooley algorithm are integrated into this CS adder. Figure 4 shows an example of CSHMR multi-

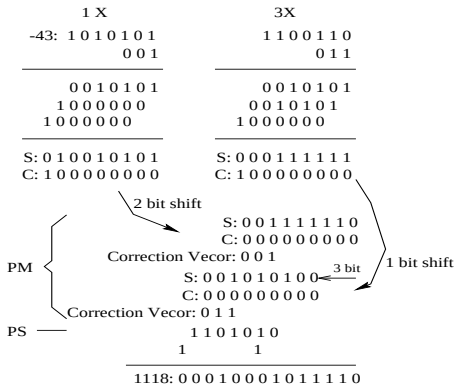

Fig. 4. Example of CSHMR computation : $-43(1010101) \times -26(1100110)$.

plication procedure: -43×-26. Here, -43 is the input X. In this example, each number is represented as 7 bit two's complement number. We use $\{1, 3, 5, 7\}$ as alphabet set which are represented in 3 bits. In the precomputer, $1X, 3X, 5X$ and $7X$ are precomputed and stored as CS redundant number. Figure 4 shows the $1X$ and $3X$ precomputation. When the scalar -26 comes in, 6 bits except the sign bit are divided into two sub sequences which are 100 and 110. 100 is two bit shifted number of one 001 and 110 is one bit shifted number of three 011. Hence, each S&S selects and shifts C and S of $1X$ and $3X$ depending on the sub sequences. Depending on the number of shifts, correction values are generated by S&Ss. For one bit shift, correction value is 001 and 011 is the correction value generated for two bit shift. The S&S also generates the PS. Finally, all the generated numbers are added after proper shift operation. This addition is performed in the CS adder tree.

3.4. Implementation of CSHMR

The use CSRS makes the maximum delay of the precomputer reduced to one full-adder and one inverter delay. Considering the binary numbers of $1, 3, 5$ and 9, they have only one or two 1's in their binary representation. If we generate output in redundant number system, no addition is required. Figure 5 (a) shows that shifting is enough to implement $1X, 3X, 5X$ and $9X$. For $7X, 11X, 13X$ and $15X$ implementation, there is only one full-adder and one inverter delay. There are only three 1's in the binary representations of 7, 11, 13 and 15. Hence, three operands has to be added for precomputation of $7X, 11X, 13X$ and $15X$. After one full-adder addition, the number of operands is reduced to two. These two numbers are used as redundant number outputs. Figure 5 (b) shows the implementation of $7X$.

Each SSU of the CSHMR is composed of two select/shifts of the CSHM and a correction. Each select/shift deals with **C** and **S** of the precomputed values in parallel. The control unit right shifts the input bit sequence to find matching alphabet and generates enable and shift signals. Zero or shifted **C** and **S** are generated by the ISHIFT depending on zero and shift signals. ISHIFT_1 generates "11110 . . . 00" and ISHIFT_2 generates "00000 . . . 00" when *zero* signal is '1'.

4. IMPLEMENTATION OF DFE

A conventional minimum mean square error (MMSE) DFE using LMS algorithm for adaptation is implemented [6, 7]. Direct

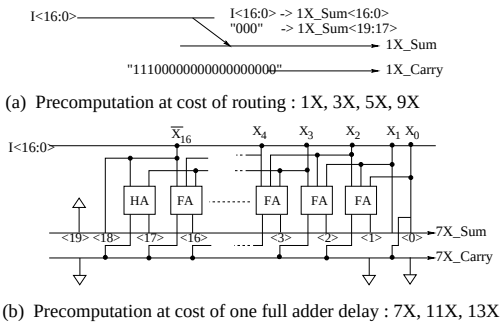

Fig. 5. Implementation of precomputer.

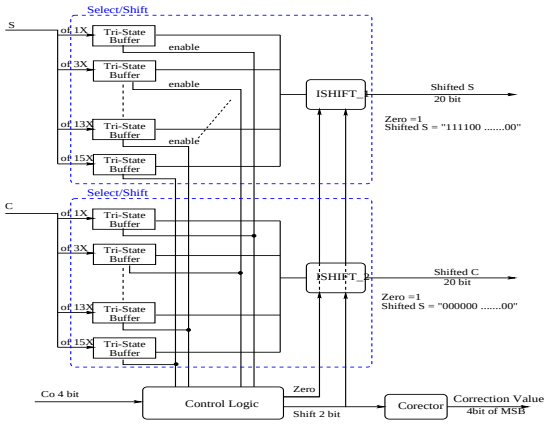

Fig. 6. Select & Shift unit implementation.

form FIR filter is used for DFE. If FIR filter is implemented with

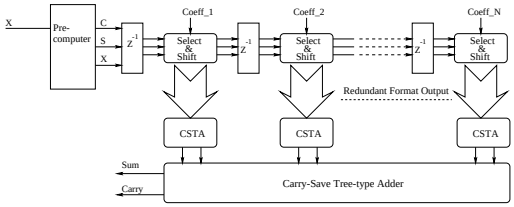

Fig. 7. FIR filter implemented with CSHMR.

CSHMR, precomputer is at the input and all the multipliers in the filter are replaced with the S&S and CS adder blocks. All the S&Ss share the precomputation results of a single precomputer. Figure 7 shows the implementation of FIR filter.

Figure 8 describes the detailed architecture of DFE, and figure 9 shows the implementation of LMS algorithm. Feedforward and feedback FIR filters are replaced with the FIR filter of figure 7 with 5 filter taps. In the implementation, the accumulators of each FIR filter are integrated into one large carry-save tree adder which also generates output in CS redundant number format. The vector merger adder, which computes the final result in non-redundant format, is inserted only when it is required.

5. RESULTS

DFEs using carry-save multiplier (CSM), Wallace tree multiplier (WTM), CSHM and CSHMR are implemented for comparison.

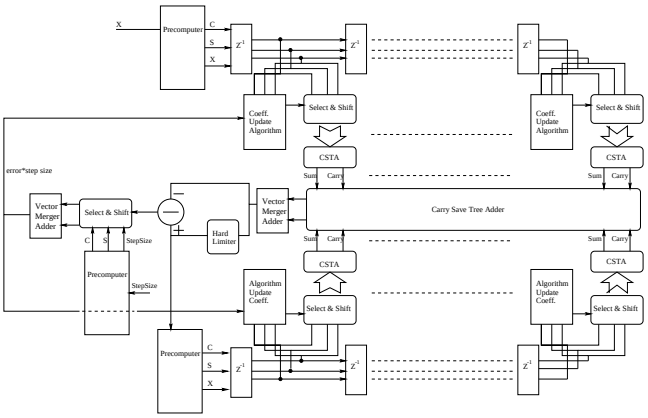

Fig. 8. DFE implementation with CSHMR.

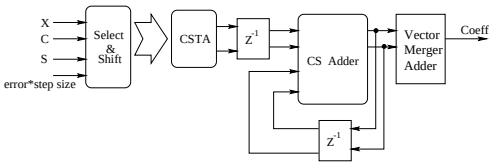

Fig. 9. Implementation of LMS algorithm.

Figure 10 shows the speed and area of DFEs implemented using different multipliers. The maximum performance of the DFE using the CSHM is improved by 13% compared with that of the DFE using WTM. The performance of the DFE using the CSHMR is improved by 34.8% by trading-off area. Figure 11 describes

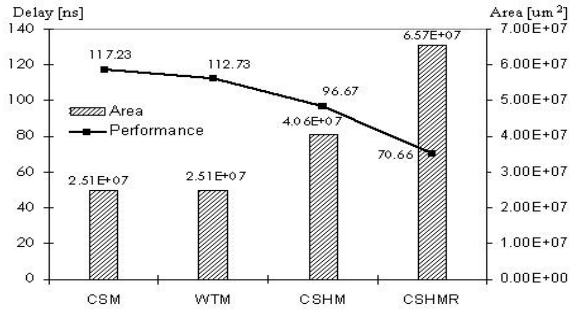

Fig. 10. Performance and area comparison between DFEs using different multipliers.

the measured power consumption of DFEs. The power consumptions of DFEs using CSHM and CSHMR at the maximum speed show increased power consumptions by 57% and 165%, respectively over the DFE using WTM. The power delay products (PDP) of the DFE using the CSHM and the CSHMR are increased by 34.7% and 66.4%, respectively.

To compare the power dissipation at the same operating speed, *voltage scaling* can be used. Reduction of supply voltage degrades the performance by trading off the power dissipation which is proportional to the second power of the supply voltage. Hence, we can obtain the power dissipation improvement in two ways; reduced supply voltage and the reduced frequency. The supply voltage of the DFE using CSHM is reduced to 2.8 V. While, a supply of 2.2 V is applied to the DFE using CSHMR. For similar performance,

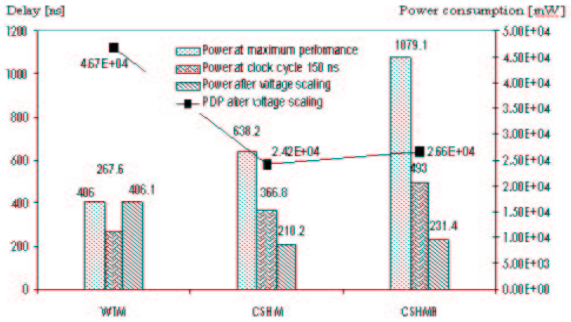

Fig. 11. Power and PDP of DFEs.

the power dissipation of DFE using CSHMR is 43% lower than DFE using WTM.

The numerical results show that the CSHMR leads to high performance filtering operation at the expense of the area and the power. However, by a proper voltage scaling, high performance can be achieved without loss of power consumption. Comparing the CSHM and the CSHMR, the CSHMR leads to much higher performance filtering operation.

6. CONCLUSION

In this paper, we presented the architecture of computation sharing multiplier for two's complement numbers. This multiplier reduces the computation complexity in filtering operation by computation re-use. A high performance decision feedback equalizer is implemented based on the suggested multiplier using carry-save redundant number scheme. Using voltage scaling, high performance and low power DFE can be achieved simultaneously.

7. REFERENCES

[1] J. Park et. al., "Non-adaptive and adaptive filter implementation based on sharing multiplication," in *IEEE ICASSP*, 2000, vol. 1, pp. 460-3.

[2] Khurram Muhammad, *Algorithmic and architectural techniques for low power digital signal processing*, Ph.D. thesis, Purdue University, 1999.

[3] L. Rijnders, Z. Sahraoui, P. Six, and H. De Man, "Timing optimization by bit-level arithmetic transformations," in *European Design Automation Conference with Eruo-VHDL*, 1995, pp. 48-53.

[4] C.R. Baugh and B.A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C22, pp. 1045-1047, Dec 1973.

[5] S. S. Nayak and P. K. Meher, "High throughput vlsi implementation of discrete orthogonal transforms using bit-level vector-matrix multiplier," *IEEE Transactions on Circuits and Systems -II: Analog and digital signal processing*, vol. 46, no. 5, May 1999.

[6] John G. Proakis, *Digital communication*, McGraw-Hill, Inc, third edition, 1995.

[7] S. Haykin, *Adaptive Filter Theory*, Prentice Hall, third edition, 1996.