Mrs. Pushpawati Changlekar, Mrs. Sujatha.S, Mrs. P. Anita / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 3, Issue 1, January -February 2013, pp.1912-1915

Id:31381Implementation of Binary Canonic Signed Digit Multiplier using Application Specific IC

Mrs. Pushpawati Changlekar, Mrs. Sujatha.S, Mrs. P. Anita
Assistant Professor, TCE Department, C.M.R. Institute of Technology, Bangalore-37

Abstract

This paper presents a novel high-speed Binary CSD (BCSD) multiplier which takes advantage of the benefits coming from the Canonic Signed Digit (CSD) number system, while overcoming the inherent overhead due to the CSD ternary representation. BCSD is a binary number system which allows representing any CSD number using the same word-length used by the two's complement representation. Thus, multipliers which make use of the BCSD technique exhibit a considerable advantage especially when the multiplicand belongs to a set of coefficients stored in a memory in its BCSD notation.

Key words: Canonic signed digit, Ternary number system, Optimally convertible block, T2I transformation,ASIC.

I. Introduction

Recent advances in digital signal processing require high speed multipliers for real-time applications. CSD (Canonic Signed Digit) ternary representation has been largely exploited thanks to its capability to perform multiplications using a minimum number of adders. Nevertheless, CSD multiplication requires that the multiplicand is first CSD encoded, thus causing an overhead that acts on both the speed and the complexity. For this reason, the use of CSD multiplication has been limited to applications where the CSD multiplicand is hard-wired into the multiplier, as typically happens when performing fixed coefficients multiplications. The growing need for both scalable architectures and adaptive signal processing pushes into the direction of having high-speed multipliers where the multiplicand can be chosen from a set of coefficients stored in a memory. In this scenario, signed digit multiplications can be still performed assuming that the signed representation of the multiplicand is already available into the memory, which in turn causes a large overhead due to the redundancy that the ternary representation inherently introduces. BCSD (Binary CSD) encoding allows taking advantage of the CSD representation while requiring no memory overhead. Synthesis results show that BCSD multiplier embedding the BCSD decoding, needed to map the BCSD multiplicand back into its CSD notation, exhibits reduced complexity and higher working frequency when compared with its signed digit equivalent, thus fully justifying its adoption in a considerable number of applications, ranging from adaptive filtering to scalable or serial FFT processing only to mention a few.

II. Canonic Signed Digit Representation

Canonic Signed Digit is based on the ternary number system (-1, 0, and 1). It is a unique representation of a binary number with minimum number of 1 and -1 digits. One of the main applications of CSD numbers, therefore, is in multiplication operation where it allows a minimum number of combined additions and subtractions to produce the product. It can be shown that for a n-bit multiplication the number of major operations (add/subtract and shift) never exceeds *n/2*, and on average this number is reduced to *n/3*, as the word size grows. The price we have to pay, however, is to convert the multiplier coefficient from 2's complement to its CSD equivalent. In case of fixed point filter coefficients this conversion has to be done once for each coefficient, but for variable filter coefficients, such as in adaptive filters, the conversion has to be repeated for each change, and hence it may not pay off because of the conversion delay. The other problem with CSD format is the need for extra spaces. Due to the ternary nature, each non-zero digit in a CSD representation requires two bits, one for the magnitude and one for the sign.

In general, there are three main issues involved in using CSD number system. First is the need to convert numbers from 2's complement to CSD format, and vice versa. This is important because much of the arithmetic and logic operations, done in an ALU**,** are typically in binary and 2's complement format, and to perform a multiplication using CSD number system one needs to do at least one conversion. The second issue deals with the space requirement. In binary logic a single memory space is all needed for a digit (bit), while in *ternary* logic the sign of the digit requires another extra space.

This is very critical, not only because it demands more hardware resources, but because it also reallocation of space which effects the over all speed. The third issue dealing with CSD numbers is the need for both addition and subtraction operations instead of only addition in case of 2's complement multiplication. This presentation addresses the first two issues. An efficient conversion from 2's complement to CSD number system is developed. In addition a new technique is presented to accommodate the signs of the non-zero digits into the data-word with no extra space added. A simple algorithm is proposed to generate a CSD number from its 2's complement equivalent, and with some minor modifications the algorithm can switch to a reverse process, that is, to convert a CSD number to its 2's complement equivalent. As pointed out earlier, the algorithm efficiently generates the
converted code and allocates it into the original 2's complement number without any extra space needed for the digit signs.

Canonic Signed Digit (CSD) Number System: A ternary number (with 0, 1 and -1 digits) **X** = xn-1,xn-2, , ... xl, x0 is a CSD number if xi.xi-l = 0, for all i = 1, 2, ..., n-1. For example, X = -0100-0-000100-01 = **("-"** stands for "-1") is a CSD number. However, the
number X = 0--0-01100010-101 has the same value and it is a ternary but not a CSD number. Now we introduce a new type of 2's complement number which is shown to be very useful and it simplifies conversion from 2's complement to CSD number format.

Inverse-2's Complement Number: The Inverse-2's complement (I-2's Complement) of a binary number A is obtained by first finding the 2's complement of A and then replacing all 1s by -1s except for the MSB which remains unchanged.
An I-2's Complement of a binary number is unique and is equal to the number itself.
Conversion of a number to its 1-2's complement format is called *"T2I Conversion".* Conversely, conversion of a number from its 1-2's complement format to a regular 2's complement format is called *"I2T Conversion".* For example, consider a binary number A=010010111010 = 1210. Applying a T21 conversion results in A=10--0-000--0; where "-" sign, in the bit structure, stands for -1. Or let B=110100101101=-723. The converted version of B in I-2's complement format becomes B=00-0--0-00--- .

Optimally Convertible Blocks: To a given 2's complement number we first add a sign extension bit equal to the sign bit, and then proceed. A block of binary digits is called *Optimally Convertible Block* (OCB) if it starts with a sequence of two or more 1s and terminates with the first 0 of two or more consecutive 0s which appear in the block for the first time. That is, an OCB can not contain two or more consecutive Os. Alternatively, an OCB is also terminated if the end bit (the MSB) is reached. Any other block in a binary number that can not be identified as an OCB or as part of an OCB is called a *non optimally convertible block* (NCB). For example, given B=110100101101 we first add a sign extension and partition the number to get 111,010,01011,01. As noticed there are two OCBs in B as follows: 01011, and 111. Also there are two NCBs in B as: 01 and 010.
It is important to note that OCBs are the only convertible blocks in a 2's complement number that are capable of reducing the number of non-zero digits. That is, to minimize non-zero digits in a 2's complement number it is only necessary to find all the OCBs and convert them using T2I conversion.
Given a binary number A in 2's complement format, the number of non-zero entries (1s and -1s) in A is reduced to a minimum if every OCBs in A are converted using the T21 conversion. If every OCB in A is converted, using the T21 conversion technique, then no two adjacent digits in A can be non-zero. This is because conversion of an OCB in A generates at least one zero next to each non-zero digit, and also a NCB can not contain any block of two or more adjacent non-zero digits.

III. Binary Canonic Signed Digit Representation

Normally to fully represent a CSD number it is required to assign a location for the sign of each nonzero digit. This evidently increases the word-size and raises it to almost double of that of its equivalent binary number. However, as one of the properties of a CSD number a non-zero digit (1 or -1) always precedes by a zero digit, except for the MSB**,** which is trivially identified by a sign-extension bit. We can certainly take advantage of this property and assign the position next to each non-zero digit to its sign. In our representation we adopt the conventional procedure and assign 0 to +1 and 1 to -1; i.e., in a CSD number digits 1 and -1 are represented by 01 and 11 respectively. This representation of a CSD number is called Binary Coded Canonic Sign Digit (BCSD) number. The mentioned method allows mapping any CSD number into its BCSD representation, which preserves the benefits coming from the CSD

numbers properties and at the same time overcomes the word-length expansion due to the ternary representation.
Algorithm to convert 2's complement number to BCSD number is given below,

Notice that for each OCB found by the algorithm (j = 1) the first two bits are skipped. This is because the first two bits in every OCB are 11 and the conversion must generate 0 - (- stands for -1). But converting a CSD to a BCSD number forces any negative digit (-1) to be represented by 11. Another interesting point note is the way the MSB is handled. By introducing an extra sign extension identical to the sign bit (xn= xn-1) the algorithm can carry all the way to the final bit without any interruption. At the end the sign extension becomes the sign bit for the MSB, if the MSB is nonzero.

Conversion examples,
2's Complement
0010,1101,1101,0010
0111,0110,1101,0010
BCSD Equivalent
11 10,1001,0011,1101
0110,1001,0100,1101

IV. BCSD multiplier Architecture

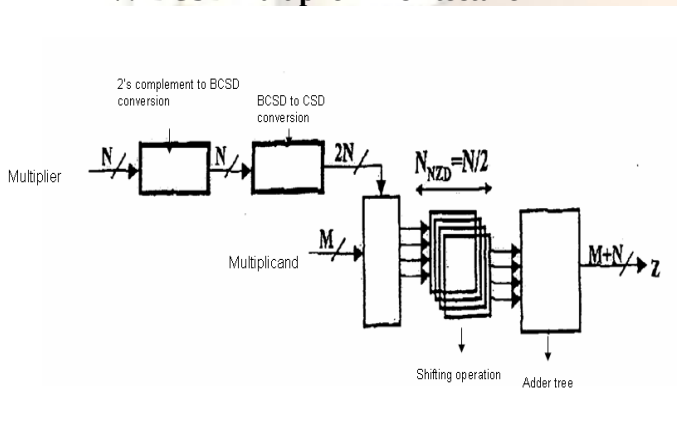

Figure 1 . BCSD multiplier architecture

The above architecture gives a unique method of multiplication which involves less number of adders to perform the multiplication procedure.
The inputs i.e multiplicand and multiplier are given in 2's complement format. The multiplicand is BCSD encoded using algorithm mentioned above. The encoded BCSD number is decoded to CSD number and then multiplied with the multiplier in 2's complement format using CSD multiplication.
The bits of the CSD number i.e. starting from pair of LSB numbers the following operations are performed on the multiplier. The procedure repeats for every pair of bits
of the CSD multiplicand and the following operations are performed on the multiplicand and the result is the product of the given numbers. First we need to initialize product register to 0 and then For 00 only left shift of the product register. For 01 adding product register with the multiplier and left shifting the product register. For 11 subtracting multiplier from the product register and then shifting the product register.

V. Implementation and results

The simulation results of the given multiplier are shown in VCSim from SYNOPSYS.

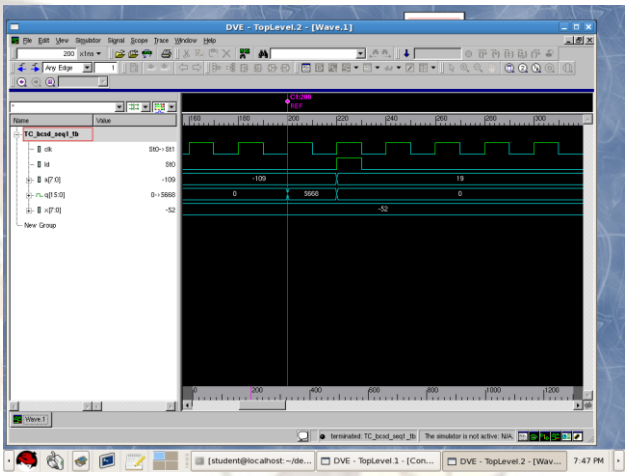

Figure 2. Simulation results of BCSD multiplier.

The synthesis of BCSD multiplier and Booth multiplier are done using Design Compiler and the designs are constrained using following constraints,

1. analyze -format Verilog-library work./codes/bcsd.v Analyzes HDL files and stores the intermediate format for the HDL description in the specified library.
2. elaborate -library work bcsd Builds a design from the intermediate format of a Verilog module, a Verilog entity and architecture, or a VHDL configuration
3. compile Performs logic-level and gate-level synthesis and optimization on the current design.
4. report_area This command gives the unconstrained area of the design.
5. report_power This command gives the unconstrained power of the design.
6. report_timing This command gives the unconstrained timing of the design.
7. reate_clock -period 1.5 clock -name main_clock Creating a virtual clock for the design which gives the timing information.

8. set_clock_uncertainty -setup 0.255 main_clock
9. set_clock_uncertainty -hold 0.122 main_clock The above commands will set the setup and hold times for the clock.
10. set_clock_transition 0.01 main_clock Sets the clock transition for the clock. set_output_delay 0.4 [all_outputs] -clock main_clock

This command sets a output delay for the current design. We assume that there might be a another module which needs data from the current design.
set_max_delay 3 -from [all_inputs] -to [all_outputs]
This command sets the combinational delay present in the design.
compile -map effort_high Here a good amount of effort is put by the tool in generating the netlist.

After synthesis the design is passed through PRIMETIME tool which is the timing sign off tool from SYNOPSYS.

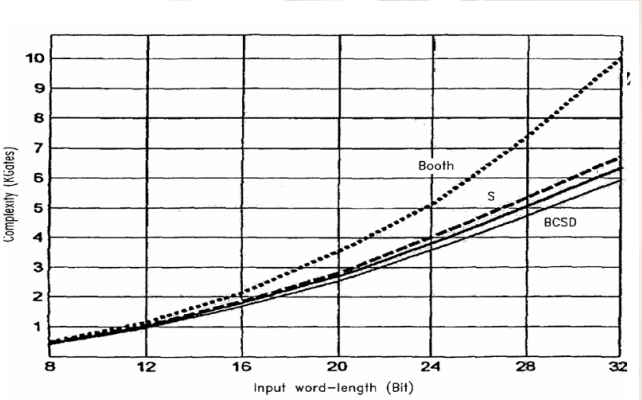

Figure 3 complexity comparison

To take into account the impact of the input word-length, synthesis has been carried out for input word-lengths ranging from 8 to 32 bits. Since CSD and Booth multipliers

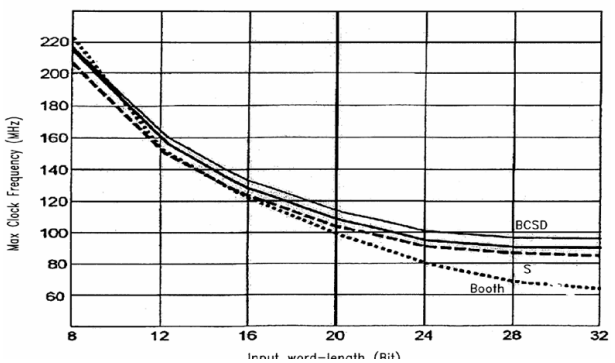

Figure 4. Speed Comparison

suffer the redundancies due to the ternary representation,
they have been implemented as *2's* complement multipliers, thus embedding also the Booth and CSD encoding functions respectively. It is worth to notice that the complexity gain of the BCSD multiplier respect to the Booth and the CSD multipliers increases with the increasing of the input word-length. This is largely due to the presence of the encoding functions included in the Booth and the CSD multipliers, whose load becomes considerable even for modest word lengths. In terms of speed, both CSD and BCSD multipliers have almost the same behavior, which depends on the fact that CSD encoding has the same recursive nature of the BCSD decoding, even if the latter exhibits a smaller complexity which in turn results in an overall speed improvement. The BCSD multiplier which makes use of the extended BCSD decoding performs better than the conventional scheme in terms of both complexity and speed, but as drawback it requires an extra-bit for each BCSD encoded operand.

VI. Conclusions

Encoded BCSD multiplicand has been proposed results for different input word-lengths have revealed a remarkable gain in both complexity and speed, thus justifying its adoption in a wide context of applications.

REFERENCES

[I] A. Peled, "On the hardware implementation of digital signal processors", *IEEE Trans. On Acoustics, Speech, and Signal Proc*, vol. 24(1), pp. 76-86, Feb. 1976.
[2] R. Hashemian, "A New Method for Conversion of a 2's Complement to Canonic Signed Digit Number System and its Representation", *IEEE 3d" Asilomar Con$ on Signals Systems and Comp.*, vol. 2, pp. 904-907, Nov. 1996.
[3] P.M. Kogge, H.S. Stone, "A Parallel Algorithm for the Eficient Solution of a General Class of Recurrence Equations", *IEEE Trans.*, vol. 22(8), pp. 786-793, Aug. 1973.
[4] R.P. Brent, H.T. Kung, "A Regular Layout for Parallel Adders" *IEEE Trans. on Comp.*, vol. 31(3), pp. 260-264, Mar. 1982.
[5] J. Sklansky, "Conditional sum addition logic", *IRE Trans. On Electron. Comp.*, vol. 9(6), pp. 226-231, Jun. 1960.
[6] O.L. MacSorley, "High Speed Arithmetic in Binary Computers", Proc. IRE, vol. 49, Jan. 1961.
[7] Advanced ASIC Chip Synthesis using Synopsys Design Compiler, Physical Compiler, and primetime- Himanshu Bhatnagar, Kluwer Academic Publishers.