A Logarithmic Time Method for Two's Complementation

Jung-Yup Kang[1] and Jean-Luc Gaudiot[2]

[1] Mindspeed Technologies,
Inc., Newport Beach, CA 92660, USA
[2] University of California at Irvine,
Irvine, CA 92612, USA

Abstract. This paper proposes an innovative algorithm to find the two's complement of a binary number. The proposed method works in logarithmic time ($O(logN)$) instead of the worst case linear time ($O(N)$) where a carry has to ripple all the way from LSB to MSB. The proposed method also allows for more regularly structured logic units which can be easily modularized and can be naturally extended to any word size. Our synthesis results show that our method achieves up to 2.8× of performance improvement and up to 7.27× of power savings compared to the conventional method.

1 Introduction

Signed binary numbering representation [5, 7, 11, 12] (based on two's complement numbers) is a nearly universally used numbering representation in the computing world. Thus, in computer systems which are based on this two's complement representation, operations to find the two's complement of a signed binary number are frequently executed. This is indeed true for applications which require to find the absolute value of signed binary numbers. For instance, motion estimation operations of MPEG encodings [9, 10, 15, 17] require to find the absolute value of the difference (of pixel values) for each pixel position for each block comparison. Another example would be multipliers [2, 8, 13] that need to find the two's complement of the multiplicand for the negative encodings of Booth algorithms [2, 3, 13, 14, 18].

Despite the frequent need for finding the two's complement of a signed binary number, two's complementation of a binary number is still carried out using the conventional way of complementing each bit and adding 1 to the complemented number. By doing so, we cannot ignore the possibility of a carry propagating all the way from the LSB (Least Significant Bit) to the MSB (Most Significant Bit). However, we have learned that the speed of finding the two's complement of a binary number is critical to the performance improvement for a group of applications. Our recent study indicates that if the two's complement of the multiplicand of a multiplication was found fast, there can be up to 40% of performance improvement [8]. It has also been reported by Hashemian [6] that in some specific parallel processing applications, it is more effective to expedite the two's complementation by using special-purpose hardware rather than by using an adder and inverters.

Therefore, in this paper, we present an efficient algorithm and architecture to find the two's complement of a binary number in a truly logarithmic time ($O(logN)$) instead of worst case of $O(N)$ time when using an adder. The proposed method also allows for more regularly structured logic units which can be easily modularized and can be naturally extended to any word size. In the next section, the conventional methods to find the two's complement of a binary number and their problems are discussed. Then, our logarithmic method and its possible implementations will be introduced. Finally, the evaluation of the algorithm followed by the module generation techniques for our two's complementation algorithm will be discussed before the summary of this paper.

2 Conventional Methods

As mentioned before, conventionally (and by definition), the two's complement of a binary number is found by complementing each bit and adding 1 to the complemented number. However, by doing so, there is the possibility of a carry propagating all the way from the LSB to the MSB. Therefore, the time complexity of finding the two's complement of a binary number is at least that of one addition (plus the complementation of each bit). However, even this delay is too large for fast multiplier architectures [8] and not efficient for some specific parallel applications [6, 10, 15, 17].

There is another well-known conventional method in which all the bits after the rightmost "1" in the word are complemented and all the other bits are left untouched. For example, the two's complement of the binary number 001010_2 (10_{10}) is 110110_2 (-10_{10}) (Figure 1). For this number, the rightmost "1" happens in bit position 1 (the check mark position in Figure 1). Therefore, values in bit positions 2 to 5 can simply be complemented while values in bit positions 0 and 1 are kept as they were.

Our method is an extension of the latter algorithm. We observed from this algorithm, that two's complementation comes down to finding the conversion signals that are used for selectively complementing some of the input bits. If the conversion signal at any position is "0" (the red crosses in Figure 1), then the value is kept as it is and if the conversion signal is "1" (the green check marks in Figure 1), then the value is complemented. All the conversion signals to the left of the rightmost "1" are 1 and all the conversion signals to the right of the rightmost "1" (and the conversion signal for the rightmost "1") are 0. For example, for data word 00101000_2, the conversion signals would be "11110000_2." Applying these conversion signals to the input (complementing only the most significant 4 bits in this case) would result in the two's complement of the input (11011000_2).

However, this searching for the rightmost "1" could be as time consuming as rippling a carry through to the MSB since the previous bits information must be

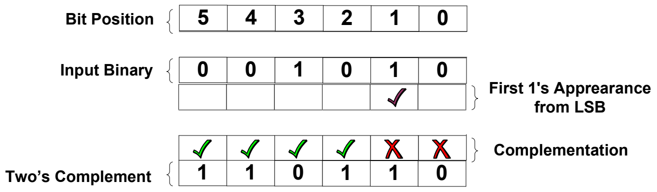


Fig. 1. Two's Complement Conversion Example

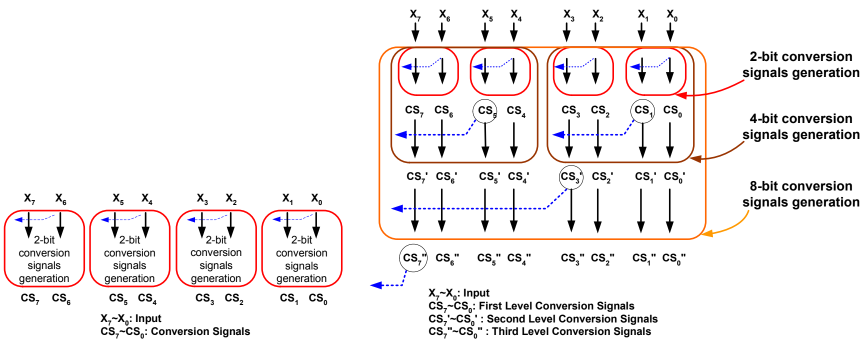
transferred to the MSB to determine which one is the rightmost "1." Therefore, we must find a method to expedite this detection of the rightmost "1." As we shall see, this search for the rightmost "1" can be achieved in logarithmic time using our binary search tree-like structure.

One possible way to implement some fast logic which will produce the adequate conversion signals would be to wire each input bit along with the preceding less significant bits to an OR-gate that accepts that many inputs. This allows for each input bit to determine whether there was a "1" in any lower order bit position and to produce its own conversion signal. However, in such cases, although it seems possible to produce the conversion signal in ideal constant time, each input bit must drive a significant number of wires (up to the number of input bits for the LSB). This is not considered to be practical nor efficient (in terms of implementation).

3 Proposed Logarithmic Method

Consequently, in this section, we describe an efficient (in terms of speed and implementation) algorithm which determines the conversion signals needed to perform two's complementation. We first find the conversion signals for a 2-bit group by grouping two consecutive bits (the grouping always starts from the LSB) from the input and find the conversion signals in each group as shown in Figure 2(a). Then we find the conversion signals for a 4-bit group (formed by two consecutive 2-bit groups). Then we find the conversion signals for an 8-bit group (formed by two consecutive 4-bit groups). This divide-and-conquer approach is pursued until the whole input word has been covered.

When grouping two 2^n-bits groups, the leftmost conversion signals from the right group contain the accumulative information of its group about whether a "1" ever appeared in any bit position of its group, so that a conversion signal should force all the conversion signals from the left group all the way to the "1" if it is itself is a "1." For instance, as shown in Figure 2(b), if CS_1 (the leftmost conversion signal from the right group) = "1," the conversion signals from the left group (CS_2 and CS_3) should be forced to a "1," regardless of their previous values. If CS_1 = "0," nothing happens to the conversion signals from the left group. This variable control is shown with a dashed arrow. Likewise, CS_5 may affect conversion signals CS_6 and CS_7. The same goes for CS_3' which may affect the conversion signals (CS_7', CS_6', CS_5', and CS_4').


2-bit conversion signals generation
4-bit conversion signals generation
8-bit conversion signals generation
X7~X0: Input
CS7~CS0: Conversion Signals
CS7~CS0: First Level Conversion Signals
CS7'~CS0' : Second Level Conversion Signals
CS7"~CS0" : Third Level Conversion Signals

(a) Conversion Signals for Pairs of Bits (b) Conversion Signals for Eight Bits

Fig. 2. Determining the Conversion Signals

The inputs to the 2-bit group are bits from the original binary number. However, the inputs to the next level groups are conversion signals from the previous level. For instance, the inputs to the 4-bit group are the conversion signals generated from two 2-bit groups. Therefore, from the second level (4-bit grouping) on, the conversion signals are scanned in order to find the rightmost "1."

After determining the conversion signals, two's complementation is a mere complementation of the input binary according to the conversion signals. One possible implementation of our algorithm is shown in Figure 3(a). Figure 3(b) shows another version of the design using NAND, NOR, and inverter gates. Once we have the complete conversion signals, these signals are shifted left 1 bit and EXOR-ed with the input to create the two's complement of the input. In Figure 3(a), X_0 to X_7 represent the input and X'_0 to X'_7 represent its two's

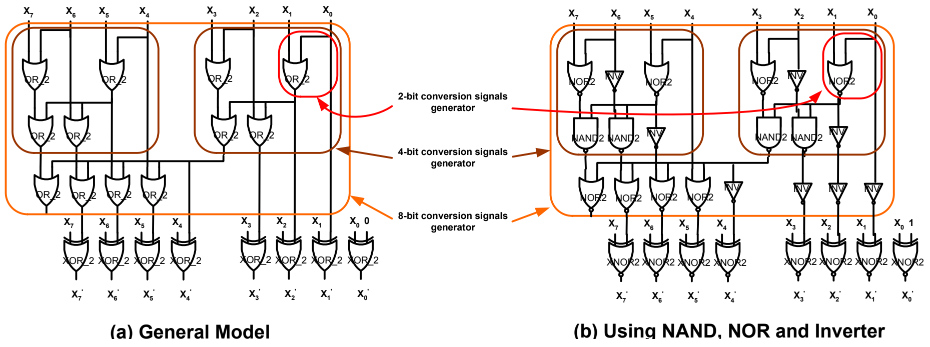

2-bit conversion signals generator
4-bit conversion signals generator
8-bit conversion signals generator
(a) General Model
(b) Using NAND, NOR and Inverter

Fig. 3. A Gate-Level Diagram of 8-bit Two's Complementation Logic Using Our Approach

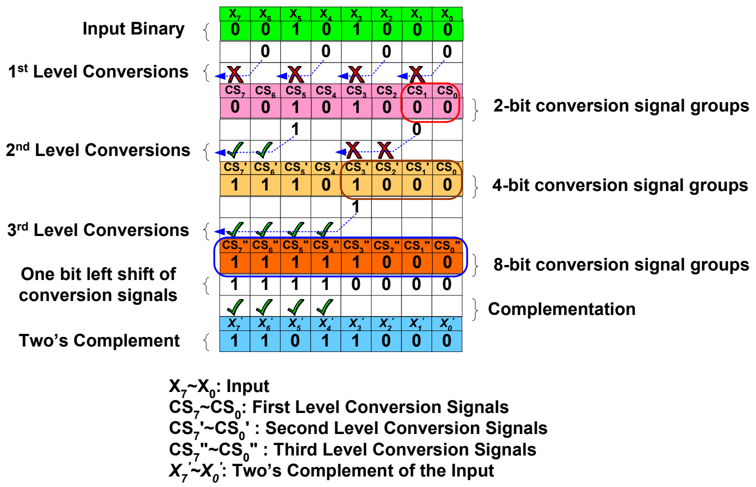

Input Binary
1st Level Conversions
2nd Level Conversions
3rd Level Conversions
One bit left shift of conversion signals
Two's Complement
2-bit conversion signal groups
4-bit conversion signal groups
8-bit conversion signal groups
Complementation
X7~X0: Input
CS7~CS0: First Level Conversion Signals
CS7'~CS0' : Second Level Conversion Signals
CS7"~CS0" : Third Level Conversion Signals
X7'~X0': Two's Complement of the Input

Fig. 4. 8-bit Example of Two's Complementation Using Our Approach

complement. One complete example of two's complementation of "00101000_2" is shown in Figure 4.

4 Performance Evaluation and Analysis

In order to measure the performance of the proposed algorithm and its implementation, we designed our algorithm using Verilog HDL (Hardware Description Language) and synthesized it using Synopsys synthesis tools [16]. Note that we used Artisan TSMC 0.13um 1.2-Volt standard-cell library [1] with "slow corner" operating conditions for our synthesis. We estimated the area, delay, and power of our designs and in order to measure the performance of the two's complementation of larger words, we expanded our proposed 8-bit two's complement logic in Figure 3(b) to larger sizes (such as 16-, 32-, 64-, and 128-bits). In order to compare the performance (against the conventional method using an adder), we implemented a two's complementation logic using a CLA (Carry-Lookahead Adder in [4]). (We used a high speed CLA for the conventional method instead of a ripple carry adder in order to be as fair as possible in our evaluation.)

Our synthesis results (Table 1) show that both methods result in linear growth in area and power as the input size increases. The delays for both methods show somewhat logarithmic characteristics. In our method, we observe that the added delay from one column (2^n-input) to the next column (2^{n+1}-input) is the one additional level of OR-gates, the associated wire delay, and the delay for driving twice the number of OR-gates (note that it would be a perfect logarithmic growth if there were no wire delays or delays due to the high fan-outs required in the last level of OR-gates). This can be made clearer as we observe

Table 1. Synthesis Reports

	Input (*n-bit*)	8-bit	16-bit	32-bit	64-bit	128-bit
Our Method	Delay (*ns*)	0.47	0.71	1.06	1.62	2.17
	Area (*um*^{2})	201	480	1111	2528	5661
	Power (*mW*)	0.096	0.18	0.35	0.69	1.37
Conventional (using CLA)	Delay (*ns*)	1.33	1.64	2.40	2.80	3.35
	Area (*um*^{2})	497	997	2122	4274	8614
	Power (*mW*)	0.51	1.04	2.35	4.74	9.96

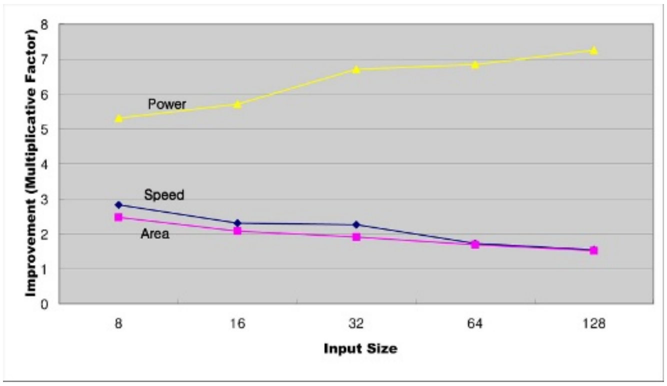


Fig. 5. Improvement (Speed, Area, and Power) of Our Two's Complementation Along with Input Size

the difference from one column to the other as we move right from one column to the other in the table.

When the methods are compared, our approach brings up to 2.8× (when $n = 8$) of performance improvement, up to 2.47× (when $n = 8$) of area savings, and up to 7.27× (when $n = 128$) of power saving when compared to the conventional method. We notice that as we increase the input size, the improvements from delay and area shrink (Figure 5). When $n = 128$, we achieve about 1.54× of performance improvement and about 1.52× of area saving. We believe this phenomenon is due to the fact that, as we increase the size of the operator, the fan-out of the last stage OR-gate is severely impacted which results in greater delays and area penalty. The power savings (in percentage) are about the same across the input sizes.

Related to our method, Hwang [7] and Hashemian [6] have shown similar approaches (finding the conversion signals). However, our method is logarithmic whereas Hwang's method is linear and Hashemian has focused on circuit optimization to improve the performance. Our approach is more general and shows better adaptability to any word size.

5 Module Generation

Our two's complementation algorithm can be easily modularized and expanded to cover binary numbers of any size. First, as shown in Figure 3(a), a 2-bit group can be modularized into a 2-bit conversion signals generator (using one OR-gate). Then, two 2-bit conversion signals generators and two more OR-gates form a 4-bit conversion signals generator. Again, two 4-bit conversion signals generators and four more OR-gates constitute an 8-bit conversion signal generator. In this fashion (two 2^n-bit conversion signals generators and 2^n more OR-gates connected to the left 2^n-bit conversion signals generator), we can continue for any 2^{n+1}-bit grouping.

6 Conclusions

This paper has introduced an innovative and efficient method to find the two's complement of a binary number. When using the proposed method, the two's complement of a binary number can be found in logarithmic time and can be used for cases where faster two's complementation is necessary such as fast multiplications as well as some specific parallel processing applications. At the same time, our approach brings a more regular structure which can be easily modularized and can be easily expandable to any word size. Our synthesis results show that our method achieves up to 2.8× of performance improvement and up to 7.27× of power savings compared to the conventional method.

Acknowledgements

This paper is based upon work supported in part by NSF grants CCR-0234444 and INT-0223647. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Artisan Components. *TSMC 0.13μm Process CL013LV 1.2-Volt SAGE-X^{T}M Standard Cell Library Databook*. Artisan Components, October 2001.
2. A. D. Booth. A Signed Binary Multiplication Technique. *Quarterly J. Mechanical and Applied Math.*, 4:236–240, 1951.
3. F. Elguibaly. A Fast Parallel Multiplier-Accumulator Using the Modified Booth Algorithm. *IEEE Transactions on Circuits and Systems*, 47(9):902–908, 2000.
4. M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2003.
5. D. Gajski. Principles of Digital Design. Prentice Hall, 1997.
6. R. Hashemian and C. P. Chen. A New Parallel Technique for Design of Decrement/Increment and Two's Complement Circuits. In *Proceedings of the 34^{t}h Midwest Symposium on Circuits and Systems*, volume 2, pages 887–890, 1991.

7. K. Hwang. *Computer Arithmetic Principles, Architecture and Design*. Wiley, New York, 1979.
8. J.-Y. Kang and J.-L. Gaudiot. A Fast and Well-Structured Multiplier. In *EUROMICRO Symposium on Digital System Design*, pages 508–515, August 2004.
9. J.-Y. Kang, S. Shah, S. Gupta, and J.-L. Gaudiot. An Ecient PIM (Processor-In-Memory) Architecture for Motion Estimation. In *IEEE $14^t h$ International Conference on Application-specic Systems, Architectures and Processors*, pages 273–283, June 2003.
10. J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and R. Bosch. The MPEG-4 Video Coding Standard - A VLSI Point of View. In *1998 IEEE Workshop on SIGNAL PROCESSING SYSTEMS (SiPS): Design and Implementation*, pages 43–52, Octover 1998.
11. M. Mano and C. Kime. *Logic and Computer Design Fundamentals*. Prentice Hall, 2000.
12. A. Marcovitz. *Introduction to Logic Design*. McGraw Hill, 2002.
13. O. L. Mac Sorley. High Speed Arithmetic in Binary Computers. *IRE Proc.*, 1961.
14. M. R. Santoro and M. Horowitz. SPIM: A Pipelined 64x64-bit Iterative Multiplier. *IEEE Transactions on Circuits and Systems*, 24(2):487–493, 1989.
15. M. Sun and K. Yang. A Flexible VLSI Architecture for Full-search Block-Matching Motion Vector Estimation. In *IEEE Int. Symp. on Circuits and Systems*, pages 179–182, May 1989.
16. Synopsys. Design Compiler User's Guide. *http://www.synopsys.com/*, 2004.
17. K. Yang, M. Sun, and L. Wu. A Family of VLSI Designs for Motion Compensation Block Matching Algorithm. *IEEE Transactions on Circuits and Systems*, 36(10):1317–1325, 1989.
18. W.-C. Yeh and C.-W. Jen. High-Speed Booth Encoded Parallel Multiplier Design. *IEEE Transactions on Computers*, 49(7):692–701, 2000.