ULTRA LOW POWER MODULO $2^n + 1$ MULTIPLIER USING GDI

Pavankumar Reddy S
Dept Of Ece
Srm Univesity

Mrs.N.SARASWATHI
Assistant Prof. (S.G)
Dept Of Ece
Srm Univesity

Gnanavargin Rokkala
Assistant Professor
Dept Of Ece
Aditya Engineering College

Abstract:
Modulo $2^n + 1$ multiplier is one of the critical components in applications in the area of digital signal processing, data encryption and residue arithmetic that demand high-speed and low-power operation. Ultra low power and low area modulo $2^n + 1$ multiplier is designed using the GDI technology. modulo $2^n + 1$ multiplier has three major functional modules including partial products generation module, partial products reduction module and final stage addition module. The partial products reduction module is completely redesigned using the novel compressors and the final addition module is implemented using a new less complex sparse tree based inverted end-around-carry adder. The resulting modulo $2^n + 1$ multiplier is implemented in GDI cell technology and compared both qualitatively and quantitatively with the existing hardware implementations.

Keywords: Modulo multipliers, Residue Number System (RNS), Compressors, Sparse Tree Adder, GDI technology

I. Introduction

Modulo arithmetic has been widely used in various applications such as digital signal processing where the residue arithmetic is used for digital filter design [1, 2]. Also, the number of wireless and internet communication nodes has grown rapidly. The confidentiality and the security of the data transmitted over these channels have becoming increasingly important. Cryptographic algorithms like International Data Encryption Algorithm (IDEA) [5, 6, 7, 8] are frequently used for secured transmission of data.

In IDEA there are three major components that's decide the overall power area and performance. They are modulo 2^n addition, bitwise-xor and modulo $2^n + 1$ multiplier. Modulo 2^n addition and bitwise-xor will take less time and easy to implement improving the area and power efficiency of the modulo $2^n + 1$ multiplication operation leads to significant decrease in area and power consumption of the IDEA cipher.

Here we introducing new low power design technique called GDI (Gate-Diffusion-Input) technology Instead of CMOS technology. The modulo multiplier has three major blocks partial product generation, partial product reduction, and final stage addition. The partial product reduction block use the GDI EXOR gates this the area of the partial product block which will reduced to 65% of the area. In the partial product generation blocks we will use GDI based AND and OR gates.

II. COMPRESSORS

A. GDI Vs CMOS:

A new low power design technique that solves most of the problems known as Gate-Diffusion-Input (GDI) is proposed [17]. This technique allows reducing power consumption, propagation delay, and area of digital circuits. The GDI method is based on the simple cell shown in Figure.1. A basic GDI cell contains four terminals – G (common gate input of nMOS and pMOS transistors), P (the outer diffusion node of pMOS transistor), N (the outer diffusion node of nMOS transistor), and D (common diffusion node of both transistors)

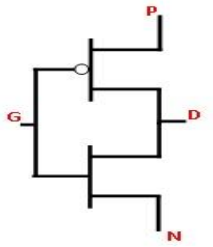


Fig.1 basic gdi cell

B. Description of compressors:

A (p,2) compressor[8,9] with p inputs $X_1, X_2, \ldots X_p$ and two output bits Sum and Carry along with carry input bits and carry output bits is governed by the equation:

$$\sum_{i=1}^{p} X_i + \sum_{i=1}^{p} (c_{in})_i = \text{sum} + 2\left(\text{carry} + \sum_{i=1}^{p} (c_{out})_i\right)$$

For example, a (5:2) compressor takes five inputs and two carry inputs and generates a Sum and Carry bit along with two carryout bits. Diagrams of 5:2 and 7:2 compressors are shown in Fig. 2 & 3. Are designed using the GDI multiplexers.

The Boolean equation of carry generator block is given by

$$O = (X_1 + X_2)X_3 + X_1X_2$$

TABLE:1 different operations using GDI

N	P	G	D	Function
0	B	A	A'B	F1
B	1	A	A'+B	F2
1	B	A	A+B	OR
B	0	A	AB	AND
B	A	S	S'A+SB	MUX
0	1	A	A'	NOT

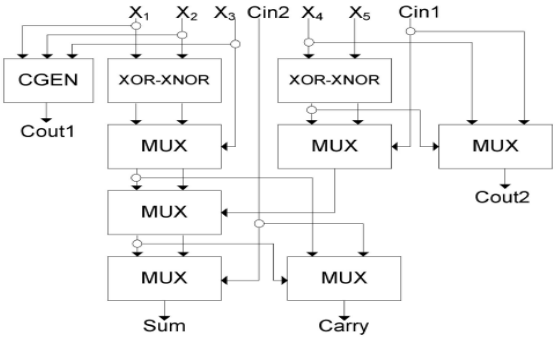


FIG 2 5:2 compressor

Iii. Algorithm For Implementation Of Modulo 2*n*+1 Multiplier

The algorithm for computation of X·Y mod $2^n + 1$ is described below. From the architectural characteristic comparisons, the algorithm presented in is considered as the best existing algorithm for the computation of X·Y mod $2^n + 1$ in the literature. Hence, this algorithm is used for the proposed implementation of the modulo multiplier. According to the algorithm it takes two n+1 bit unsigned numbers as inputs and gives one n+1 bit output. The proposed implementation can be adapted to IDEA cipher, in which the mod $2^n + 1$ multiplication module takes two n-bit inputs and gives one n-bit output, by assigning the most significant bits of the inputs zeros and neglecting the most significant bit of the output Let $|A|$ B denote the residue of A modulo B. Let X and Y be two inputs represented as X=$X_n X_{n-1} \ldots X_0$ and Y=$y_n y_{n-1} \ldots y_0$ where the most significant bits x_n and y_n are '1' only when the inputs are 2^n and 2^n respectively. X·Y mod $2^n + 1$ can be represented as follows:

$$P = |X.Y|_{2^n+1} = \left| \sum_{i=0}^{n} x_i 2^i . \sum_{j=0}^{n} y_i 2^i \right|_{2^n+1} = \left| . \sum_{i=0}^{n} \left(\sum_{j=0}^{n} p_{i,j} 2^{i+j} \right) \right|_{2^n+1}$$

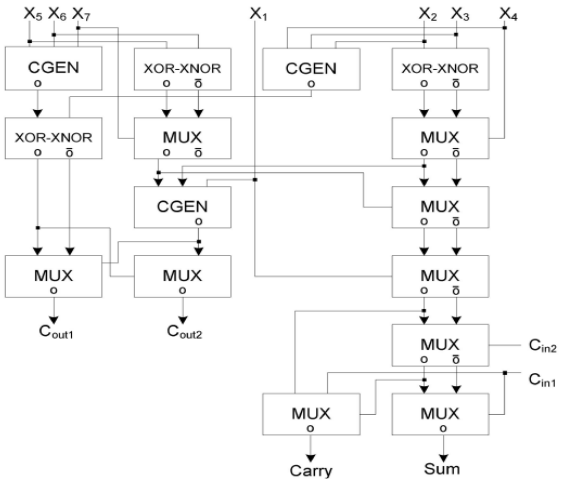


FIG 3 7:2 compressors

The n×n partial product matrix in Fig. 6 is derived from the initial partial product matrix in Fig. 4, based on several observations. First observation is, the initial partial product matrix can be divided into four groups A, B, C and D in which the terms in only one group can be different from '0'. Groups A, B, D and C are different from '0', if inputs (X,Y) are in the form of (0Z,0Z),(1Z,0Z),(0Z,1Z) and (10. . .0,10. . .0) respectively (here 'Z' is a 16-bit vector). Hence the four groups can be integrated into a single group by performing logical OR operation (denoted by v) instead of adding the bits arithmetically. Logical OR operation is performed on the terms of the groups B, D and A in the columns with weight 2^n up to $2^{2n} - 2$ and on the two terms of the groups B and D with weight $2^{2n} - 1$ (the ORed terms of the groups B and D are represented by q_i, where $q_i = p_{n,i} \, V p_{i,n}$). Since $|2^{2n-1}|_{2^n+1} = 2^{n-1}$+1, the term with weight $2^{2n} + 1$, p_{n-1}, can be substituted by two terms q$n-1$ in the columns with weight 2^{n-1} and 1, respectively, and ORed with any term of the group A there. Moreover, since $|2^{2n}|_{2^n+1}$=1, the term $p_{n,n}$ can be ORed with p0,0. The modified partial product matrix is shown in Fig. 5. Second observation is repositioning of the partial product terms in the modified partial product matrix, with weight greater than 2^{n-1} based on the following equation:

$$|s2^t|_{2^n+1} = |-s2^{|t|_n}|_{2^n+1} = |(2^n + 1 - s)2^{|t|_n}|_{2^n+1} = |\bar{s}2^{|t|_n} + 2^n 2^{|t|_n}|_{2^n+1} \ldots (1)$$

Equation (1) shows that the repositioning of each bit results in a correction factor of $2^n 2^{|t|_n}$. In the first partial product vector, there is only one such bit and in the second partial product vector 2 bits need to be repositioned and so on. Hence the correction factor for the entire partial product matrix would be:

$$\mathbf{COR1} = |2^n(2^n - n - 1)|_{2^n+1} \ldots (2)$$

The n×n partial product matrix along with the equation (2) results in n+1 operands. These partial product terms can be reduced into two final summands Sum array and Carry array using a Carry Save Adder (CSA) array. Suppose the carry out bit at $\mathbf{i^{th}}$ stage of CSA is $\mathbf{c_i}$ with weight 2n, this carryout can be reduced into:

$$|c_i 2^n|_{2^n+1} = |-c_i|_{2^n+1} = |2^n + \bar{c_i}|_{2^n+1}$$

Therefore the carry output bits at the most significant bit position of each stage can be used as carry input bits of the next stage. In an n-1 stage CSA array in areproduced n-1 such carry out bits. Hence there will be a second correction factor. And the overall correction factor using this algorithm is:

$$\mathbf{COR2} = |2^n(n-1)|_{2^n+1} \ldots\ldots (3)$$

The final correction factor will be the sum of COR1 and COR2. The constant '3' in equation (5) will be the final partial product.

$$\mathbf{COR} = \mathbf{COR1} + \mathbf{COR2}$$

$$= |2^n(n-1) + 2^n(2^n - n - 1)|_{2^n+1} \ldots\ldots (4)$$

$$= |2^n(2^n - 2)|_{2^n+1} = 3 \ldots\ldots \textbf{(5)}$$

Iv. Proposed Implementation Of The Mod 2*n*+1 Multiplier

The proposed implementation of the modulo multiplier consists of three modules. First module is to generate partial products, second module is to reduce the partial products to two final operands and the last module is to add the Sum and Carry operands from partial products reduction to get the final result.

A. Partial products generation

From the above *n* × *n* partial product matrix (shown in Fig. 6), it is possible to observe that the partial product generation requires AND, OR and NOT gates. The most complex function of partial product generation module is $\overline{p_{n-1,n-1} V q_{n-2}}$ where $p_{i,j} = a_i b_j$ and $q_i = p_{n,i} V p_{i,n}$.

B. Partial products reduction

The partial product reduction unit is the most important module which mainly determines the critical path delay and the overall performance of the multiplier. Hence this module needs to be designed so as to get minimum area and consume less power.

In the partial products reduction module, the *n*×*n* partial product matrix and the constant 3(i.e., correction factor) need to be added to produce the final sum and carry vectors. Zimmerman demonstrated that (*Sum*+*Carry*+1) modulo $\mathbf{2^n + 1}$ (final stage addition module) can also be calculated by (*Sum* + *Carry*) modulo $\mathbf{2^n}$ using inverted End-Around-Carry (EAC) adders, where *Sum* and *Carry* are *n*-bit vectors generated by the partial products reduction module. Modulo $\mathbf{2^n}$ inverted EAC adders have a regular structure that can be easily laid out for efficient VLSI implementation. Hence, instead of directly adding the correction factor of 3 to the *n*×*n* partial product matrix in the partial products reduction module, an intermediate correction factor of 2 has to added to the *n*×*n* partial product matrix to save a constant 1 for the final stage addition module. Since there is a saved constant 1 available in the final stage addition module, modulo $\mathbf{2^n}$ inverted EAC adder can be used instead of the complex modulo $\mathbf{2^n + 1}$ adder.

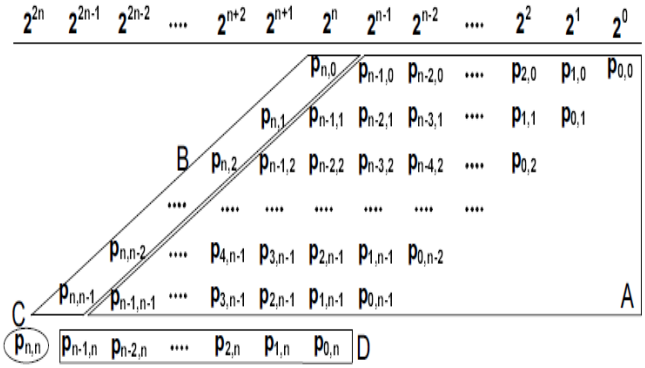


Fig-4: normal partial products

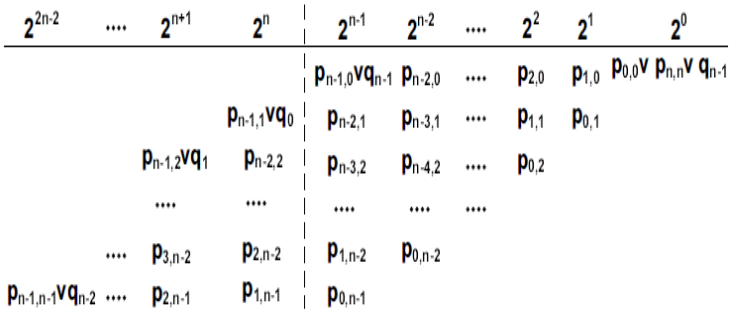


Fig 5: modified partial product matrix

Table 1: final partial product matrix

2^{n-1}	2^{n-2}	2^{n-3}	2^2	2^1	2^0
$pp_0 = p_{n-1,0} V q_{n-1}$	$p_{n-2,0}$	$p_{n-3,0}$	$p_{2,0}$	$p_{1,0}$	$p_{0,0} V q_{n-1} V p_{n,n}$
$pp_1 = p_{n-2,1}$	$p_{n-3,1}$	$p_{n-4,1}$	$p_{1,1}$	$p_{0,1}$	$\overline{p_{n-1,1} V q_0}$
$pp_2 = p_{n-3,1}$	$p_{n-4,2}$	$p_{n-5,2}$	$p_{0,2}$	$\overline{p_{n-1,2} V q_1}$	$\overline{p_{n-2,2}}$
....	....	....	....	....	....
$pp_{n-2} = p_{1,n-2}$	$p_{0,n-2}$	$\overline{p_{n-1,n-2} V q_{n-3}}$	$\overline{p_{4,n-2}}$	$\overline{p_{3,n-2}}$	$\overline{p_{2,n-2}}$
$pp_{n-1} = p_{0,n-1}$	$\overline{p_{n-1,n-1} V q_{n-2}}$	$\overline{p_{n-2,n-1}}$	$\overline{p_{3,n-1}}$	$\overline{p_{2,n-1}}$	$\overline{p_{1,n-1}}$

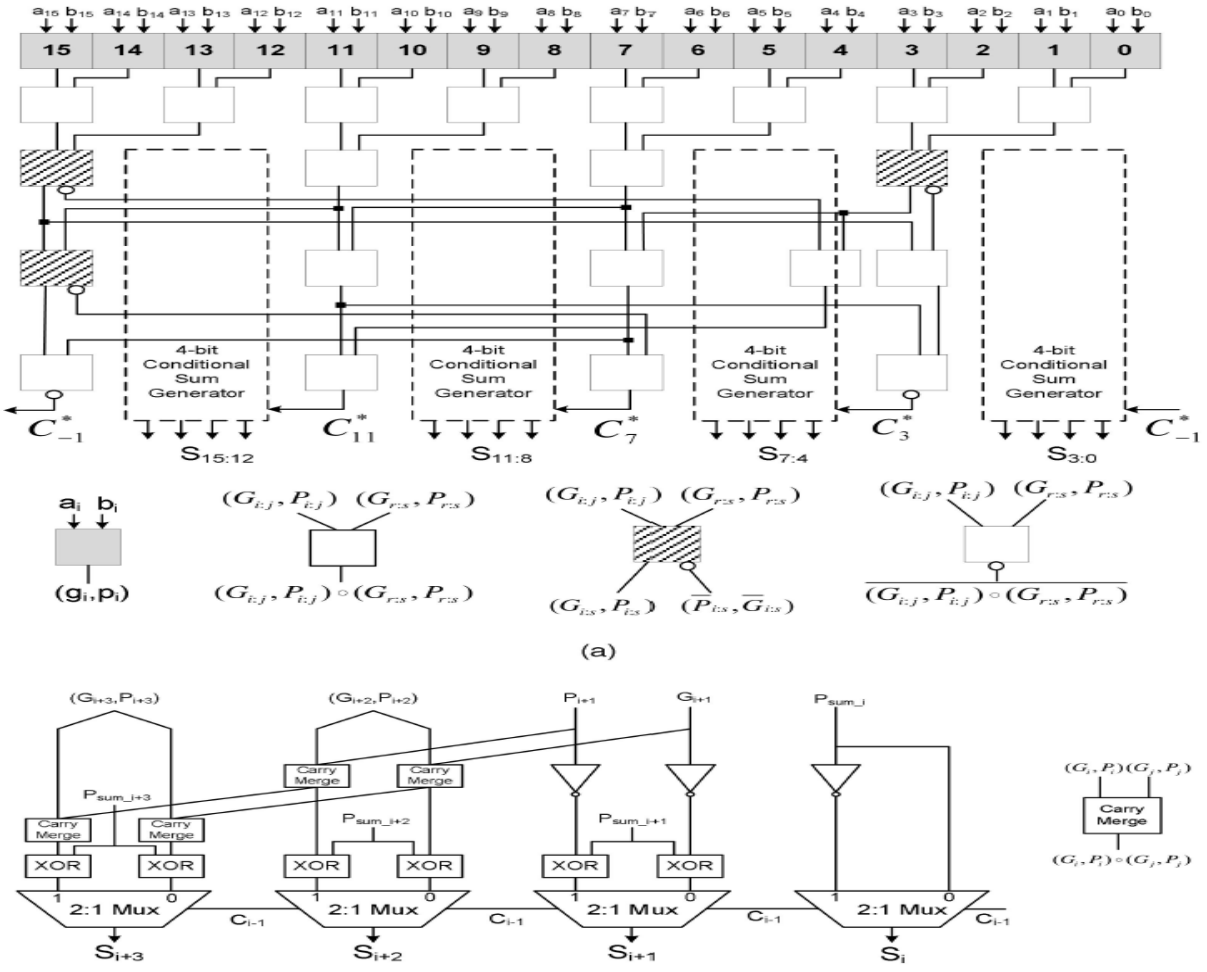


FIG 7 : (a) 16-bit Sparse-tree-based Inverted EAC adder and (b) 4-bit conditional sum generator.

Along with the constant 2, the *n* partial products should be added to produce the final *n*-bit Sum and Carry vectors. In a single stage of the Carry Save Addition, series of *n* full adders take 3 input operands and produce two *n*-bit output vectors. To add *n*+1 input operands, *n*-1 Carry Save Addition stages are required in the partial products reduction module. As the first partial product is the constant 2, in the first stage of the *n*-1 CSA stages, half adders can be used instead of full adders except for the for the second bit. The *n*-1 stage CSA can be implemented using *n* full adders in each column of the *n*-1 stages. In this regular implementation, series of full adders in the CSA adder columns can be replaced by the proposed GDI EXOR-based compressors that take the same number of inputs, which leads less power implementation of the multiplier. For example, for a modulo $2^8 + 1$ multiplier the existing CSA design uses 7 full adder stages in a single column to reduce the 9 partial products, the same reduction can be done by more efficient designs based on the proposed compressors.

In the proposed compressor-based architecture, use of suggested compressors not only reduces power consumption but also the area of the circuit. For example, the full adder implementation requires fifteen full adders in series in any column for a modulo 2^{16+1} multiplier. However, these fifteen full adders can be replaced by two 7:2 compressors, one 5:2 compressor and two 3:2 compressors .

V. Final stage addition

In binary addition operation, the critical path is determined by the carry computation module. Among various formulations to design carry computation module, parallel prefix formulation [18] is delay effective and has regular structure suitable for efficient hardware implementation. The binary addition of two numbers using a parallel prefix network is done as follows: $A = a_{n-1}a_{n-2} \ldots\ldots a_1 a_0$ and $B = b_{n-1}b_{n-2} \ldots\ldots b_1 b_0$ be two weighted input operands to the network. The generate bit (g_i) and the propagate bit (p_i) are defined as $g_i = a_i \; AND \; b_i \; and \; p_i = a_i \; OR \; b_i$ and these generate bits can be associated using the prefix operator as follows:

$(g_i, p_i) \bullet (g_{i-1}, p_{i-1}) =$
$(g_i + p_i * g_{i-1}, p_i * p_{i-1})$ $(g_{i:i-1}, p_{i:i-1})$

Where + is logical OR operation and * is logical AND operation. The carryout's (C_i) for all the bit positions can obtained from the group generate $(C_i^* = G_i^*)$ where

$(G_i, P_i) = (g_i, p_i) \bullet (g_{i-1}, p_{i-1}) \bullet \ldots\ldots (g_1, p_1) \bullet (g_0, p_0)$.

The function of End around Carry (EAC) adder is to feed back the carryout of the addition and add it to the least significant bit of the sum vector. Similarly, in inverted End Around Carry adders, the carryout is inverted and fed back to the least significant bit of the sum vector. The parallel prefix-network-based Inverted EAC [19] adder achieves the addition of the input operands by recirculating the generate and the propagate bits at each existing level in $log_2 n$ stages. Let $C_i^* (G_i^*)$ be the carry at bit position i in the inverted EAC, this can be related to G_i as follows:

$$(G_i^*, P_i^*) = \begin{cases} \overline{(g_{n-1}, p_{n-1})} \; for \; i = -1 \\ (g_i, p_i) \bullet \overline{(g_{n-1:i+1}, p_{n-1:i+1})} \\ for \; n-1 \geq i \geq 0 \end{cases}$$

……(5)

In the above equation $\overline{(G_i^*, P_i^*)} = (\bar{G}, P)$ where

$(G_i, P_i) = (g_i, p_i) \bullet (g_{i-1}, p_{i-1}) \bullet \ldots\ldots (g_1, p_1) \bullet (g_0, p_0)$

and

$(G_i, P_i) = (g_{n-1}, p_{n-1}) \bullet (g_{n-2}, p_{n-2})$

$\bullet \ldots\ldots (g_{i+2}, p_{i+2}) \bullet (g_{i+1}, p_{i+1})$

In some cases, it is not possible to compute (G_i^*, P_i^*) In $log_2 n$ stages, then in these cases, the equations in (5) are transformed into the equivalent ones as shown in (7) by using the following property [19]: suppose that $(G^x, P^x) = (g, p) \bullet \overline{(G, P)}$ and $(G^y, P^y) = \overline{((\bar{g}, \bar{p}) \bullet (G, P))}$

$G^x = g + p\bar{G} = \overline{\bar{g} + \bar{p}G}$…..(6) Therefore $G^x = G^y$ and in (2), P^y is computed as p.P. To implement the parallel prefix computation efficiently, these transformations have to be applied j number of times recursively on $(G_i, P_i) \bullet \overline{(G_{n-1:i+1}, P_{n-1:i+1})}$ using the fallowing relation:

$$n - 1 - i + j = \begin{cases} n, if \; i > \frac{n}{2} - 1 \\ \frac{n}{2}, if \; i \leq \frac{n}{2} - 1 \end{cases} \quad \ldots\ldots(7)$$

\

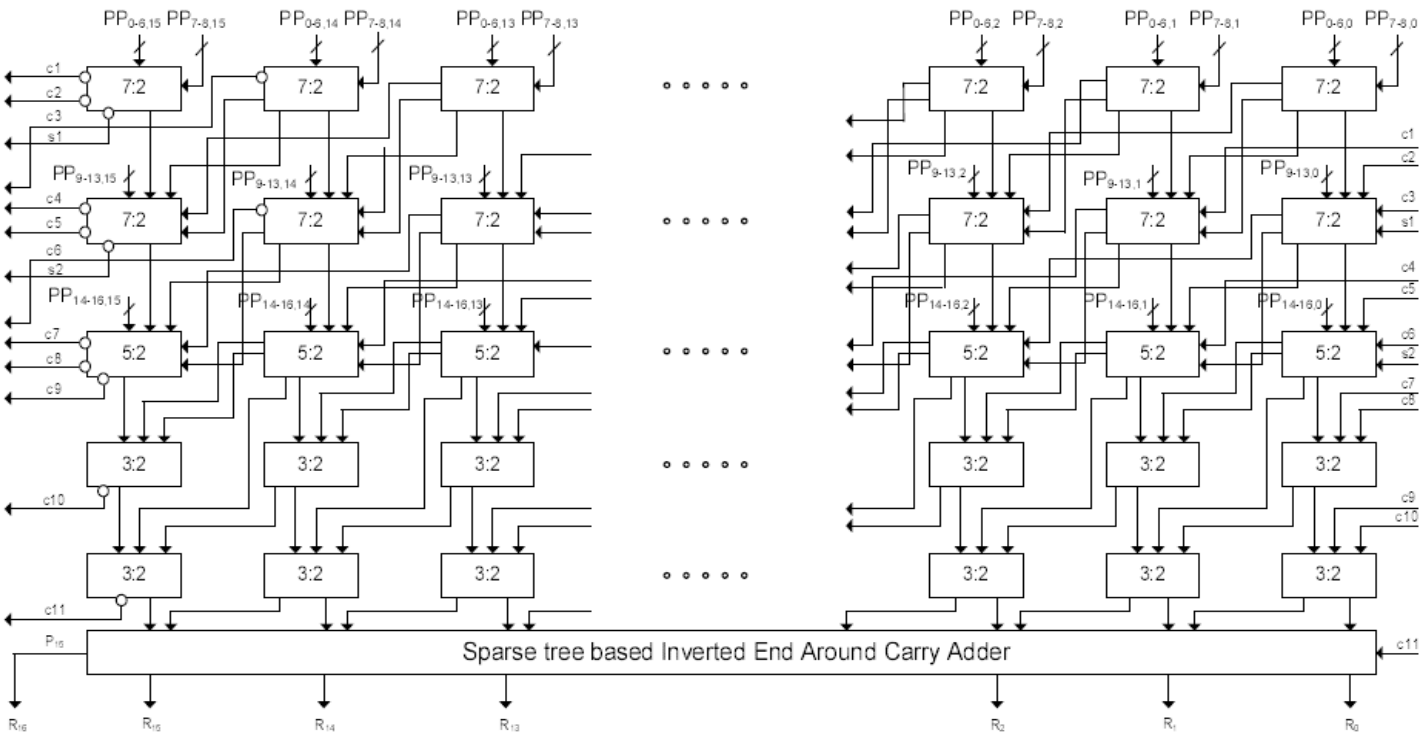


Fig 8 : proposed implimentation of modulo $2^{16}+1$ multiplier using compressors with GDI technology

The new carryout's can be computed using the following equation:

$$(G_i^*, P_i^*) = \begin{cases} \overline{(g_{n-1}, p_{n-1})} & for\ n = - \\ \overline{(P_i, G_i)} \bullet (g_{n-1:i+1}, p_{n-1:i+1}) & for\ n-1 \geq i \geq 0 \end{cases}$$

……(8)

Hence, the transformations used above to achieve the parallel prefix computation in $log_2 n$ stages result in more number of carry merge cells and thereby adding more number of interstage wires. Parallel prefix adders suffer from excessive interstage wiring complexity and large number of cells, and these factors make parallel prefix based adders inefficient choices for VLSI implementations. Therefore, a novel sparse-tree-based EAC and inverted EAC adders are used as the primitive blocks in this work.

In sparse-tree-based inverted EAC adders, instead of calculating the carry term Gi for each and every bit position, every Kth (K=4,8 . .) carry is computed. The value of K is chosen based on the sparseness of the tree, generally for 16 and 32-bit adders, K is chosen as four. The higher value of K results in higher value of noncritical path delay compared to critical path delay of O($log_2 n$) which should not be the case. The proposed implementation of the sparse-tree-based Inverted End Around Carry Adder (IEAC) is explained below clearly for 16-bit operands. For a 16-bit sparse IEAC with sparseness factor (i.e., K) equal to four, the carries are computed for bit positions -1,2,3 and 11. Here, bit position -1 corresponds to the inverted carryout $(\overline{(G_{15}, P_{15})})$ of the bit position 15. The carryout equation for the 16-bit sparse tree IEAC are as fallows

$$C_{-1}^* = \overline{(G_{15}, P_{15})}$$
$$= \overline{(g_{15}, p_{15}) \bullet (g_{14}, p_{14}) \bullet \ldots . (g_1, p_1) \bullet (g_0, p_0)}$$
$$C_3^* = (G_3, P_3) \bullet \overline{(G_{15:4}, P_{15:4})}$$
$$= (g_3, p_3) \bullet .. \bullet (g_0, p_0) \bullet \overline{(g_{15}, p_{15}) \bullet \ldots . \bullet (g_4, p_4)}$$
$$C_7^* = (G_7, P_7) \bullet \overline{(G_{15:8}, P_{15:8})}$$
$$= (g_7, p_7) \bullet \ldots \bullet (g_0, p_0) \bullet \overline{(g_{15}, p_{15}) \bullet \ldots . \bullet (g_8, p_8)}$$
$$C_{11}^* = (G_{11}, P_{11}) \bullet \overline{(G_{15:12}, P_{15:12})}$$

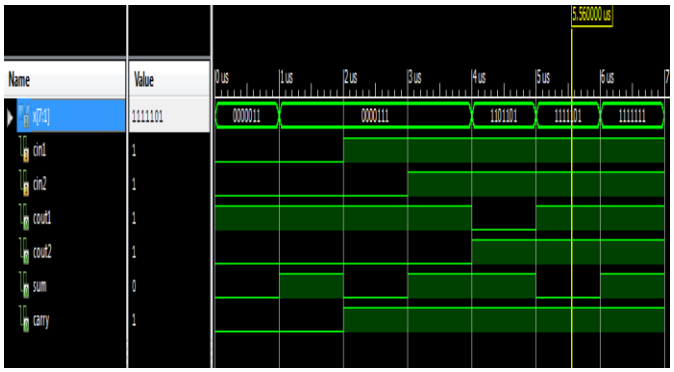
$$= (g_{11}, p_{11}) \bullet \ldots \bullet (g_0, p_0) \bullet \overline{(g_{15}, p_{15}) \bullet \ldots . \bullet (g_{12}, p_{12})}$$

Fig. 7 shows the finalized 16-bit sparse tree Inverted EAC adder. From Fig. 7, we can observe that all the carryouts are computed in $log_2 n$ stages with less number of carry merge cells and reduced interstage wiring intensity. The implementation of the sparse-tree-based EAC is similar to IEAC shown in Fig. 7, except the carry is not inverted.

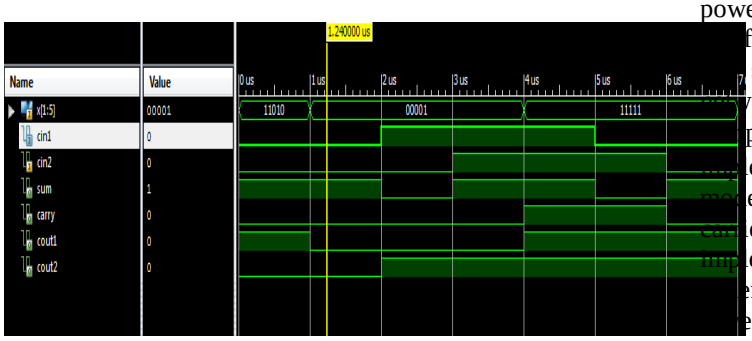
The Conditional Sum Generator (CSG) shown in Fig. 2C is implemented using ripple carry adder logic, and two separate rails are run to calculate the carries $C^*_{i+1}, C^*_{i+2}, C^*_{i+3}$ *and* C^*_{i+4} assuming the input carry C^*_i as 0 and 1. Four 2:1 multiplexers using the carry C^*_i from sparse tree network as one-in-four select line generate the final sum vector. The conditional sum generator is shown in Fig. 7b. The final sum is generated in log2n stages in IEAC sparse tree adder with less number of cells and less interstage wiring. Hence, this approach results in low power and smaller area while providing better performance.

VI. SIMULATION RESULTS:

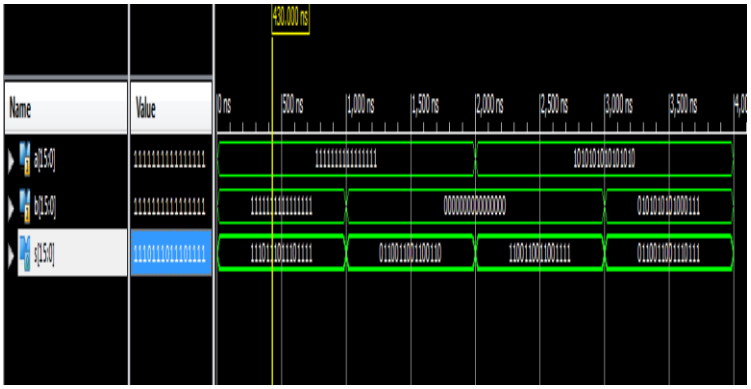
7:2 compressor:



5:2 compressor:



Spare tree adder:



Final output :

V. CONCLUSIONS:

An ultra low power implementation of the modulo $2^n + 1$ multiplier is presented in this paper. The proposed novel implementation takes advantage of newly designed low power compressors using GDI technology and sparse tree based Inverted End-Around-Carry (EAC) adders. The proposed design of the modulo $2^n + 1$ multiplier uses compressors in the partial products reduction stage, the use of the GDI compressors in place of CMOS compressors resulted in considerable improvements in terms of area and power. An efficient sparse tree based inverted EAC adder in final stage addition, which has less wiring complexity sparse carry merge cells compared to parallel prefix ork based implementations. The proposed multiplier is pared with the most efficient modulo $2^n + 1$ multiplier ementations available in the literature. The unit gate el analysis and EDA-based parametric simulation are ed out on the proposed implementation and the existing ementations to clearly demonstrate and verify potential fits from the proposed design. The proposed multiplier rified to outperform the existing implementations with respect to three major design criteria (i.e., area and power).

REFERENCES

1. Zimmermann, R., Curiger, A., Bonnenberg, H., Kaeslin, H., Felber,N., and Fichtner, W. ”A 177 Mb/s VLSI implementation of the international data encryption algorithm”, IEEE J. Solid-State Circuits, 1994, 29, (3), pp. 303-307
2. Zimmerman, R., ”Efficient VLSI implementation of modulo (2*n* ± 1) addition and multiplication” IEEE trans. Comput., 2002, 51, pp. 1389- 1399.
3. Sousa, L., and Chaves, R., ”A universal architecture for designing efficient modulo 2*n* + 1 multipliers”, IEEE Trans. Circuits Syst. I., 2005, 52, pp. 1166-1178.
4. Efstathiou, C., Vergos, H.T., Dimitrakopoulos, G., and Nikolos, D., ”Efficient diminished-1 modulo 2*n* + 1 multipliers”, IEEE Trans. Comput., 2005, 54, pp. 491-496.
5. Vergos, H.T.; Efstathiou, C., ”Design of efficient modulo 2*n* + 1 multipliers”, IET Comput. Digit. Tech., 2007, 1, (1), pp. 49-57.
6. R. Zimmermann and W. Fichtner., ”Low power logic styles: CMOS versus pass-transistor logic” IEEE J. Solid- State Circuits, vol. 32, pp. 1079-1090, July 1997.
7. Veeramachaneni, S.; Avinash, L.; Rajashekhar Reddy M; Srinivas, M.B., ”Efficient Modulo (2*k* ± 1) Binary to Residue Converters System on- Chip for Real-Time Applications” The 6th International Workshop on Dec. 2006 pp.195 - 200.
8. C-H Chang, J Gu, MZhang ”Ultra low-voltage lowpower CMOS 4-2 and 5-2 compressors for fast arithmetic circuits ” IEEE J. Circuits and Systems I, Volume: 51, Issue: 10 pp: 1985- 1997,2004
9. Rouholamini, M.; Kavehie, O.; Mirbaha, A. P.; Jasbi, S.J.; Navi, K., ”A New Design for 7:2 Compressors” Computer Systems and Applications, 2007. AICCSA ’07. IEEE/ACS International Conference on 13-16 May 2007 Page(s):474 – 478
10. Mathew, S.; Anders, M.; Krishnamurthy, R.K.; Borkar, S.; ”A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core” In IEEE Journal of Solid-State Circuits, Volume 38, Issue 5, May 2003 Page(s):689 - 695.
11. Zhongde Wang, Graham A. Jullien, William C. Miller., ”An efficient tree architecture for modulo 2*n* +1 multiplication” VLSI Signal Processing 14(3): 241-248 (1996).
12. P. Kogge and H. S. Stone., ”A parallel algorithm for the efficient solution of a general class of recurrence equations” IEEE Trans. Comput., vol. C-22, pp. 786-793, Aug 1973.
13. Sklavos N and Koufopavlou O, ”Asynchronous Low Power VLSI Implementation of the InternationalData Encryption Algorithm” proc. of 8th IEEE International Conference on Electronics, Circuits 346 and Systems (ICECS’01) (Malta 2-5 September 2001) Vol. III pp 1425-1428
14. A. Curiger et. al., ”VINCI: VLSI Implementation of the New SecretkeyBlock Cipher IDEA”, Proc. of the Custom Integrated Circuits Conference, San Diego, USA, May 1993
15. Yan Sun, Dongyu Zheng, Minxuan Zhang, and Shaoqing Li ”High Performance Low-Power Sparse-Tree Binary Adders” 8*th* International Conference on Solid state and Integrated Circuit Technology, ICSICT 2006.
16. Haridimos T. Vergos, Costas Efstathiou, Dimitris Nikolos: ”Diminished- One Modulo 2n+1 Adder Design” IEEE Trans. Computers 51(12): 1389-1399 (2002)
17. A. Morgenshtein, A. Fish, I. A. Wagner. Gate Diffusion Input (GDI) – A Novel Power Efficient Method for Digital Circuits: A Design Methodology. 14th ASIC/SOC Conference, Washington D.C., USA, September 2001.
18. P. Kogge and H.S. Stone, “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations,” IEEE Trans. Computers, vol. 22, no. 8, pp. 786-793, Aug. 1973.
19. H.T. Vergos, C. Efstathiou, and D. Nikolos, “Diminished-One Modulo 2n+1 Adder Design,” IEEE Trans. Computers, vol. 51, no. 12, pp. 1389-1399, Dec. 2002.