TELKOMNIKA, Vol. 11, No. 10, October 2013, pp. 5774 ~ 5781
ISSN: 2302-4046

Efficient Implementation of Decimal Floating Point Adder in FPGA

Yang Huijing*[1], Yu Fan[2], Han Dandan[3]
[1]School of Software, Harbin University of Science and Technology, Harbin 150040, Heilongjing, China
[2]Financial Department, Heilongjiang University, Harbin 150080, Heilongjing, China
[3]School of Software, Harbin University of Science and Technology, Harbin 150040, Heilongjing, China
*Corresponding author, e-mail: 49645521@163.com

Abstract

Decimal floating Point adder is one of the most frequent operations used by many financial, business and user-oriented applications but current implementations in FPGAs are very inefficient in terms of both area and latency when compared to binary floating point adder. This paper has shown an efficient implementation of a new parallel decimal floating point module on a reconfigurable platform, which is both area as well as performance optimal. The decimal floating-point Adder was further pipelined into five stages to increase the maximum frequency of operation. The synthesis results for a Stratix IV device indicate that our implementations have 25.1% reduction of the latency and 1.1% reduction of area compared to an existing alter-core adder design, presenting area and delay figures close to those of optimal binary adder trees.

Keywords*: Floating Point, Field Programmable Gate Array, Parallel Adder, Pipeline*

1. Introduction

Floating point adder is widely used in large set of scientific and signal processing computation. However, binary floating-point operations are not suitable for financial and commercial computations. its binary counterpart has an innate defect in aforementioned applications [1]. Decimal numbers in these applications are usually required to be represented exactly, and arithmetic operations often need to mirror manual decimal calculations, which perform decimal rounding. but most of the decimal floating point numbers cannot be exactly represented by the binary weighted series in a finite precision, and the error can be accumulated after calculations. Although Decimal floating point operations can be done through the software program, but its speed is 100~1000 tims slower than the speed of binary floating-point arithmetic [2]. Addition is the basic but the most important function among the decimal arithmetic operations. In sequential multiplication and digit recurrence division, the partial products for every iteration are accumulated by the adders. Moreover, in parallel multiplication and functional division, the partial products are reduced by the adders arranged in wallace tree structure. Since an improvement in addition can benefit to many other decimal operations, many methods and algorithms were applied to boost the performance of the decimal adder [3] [4].

Decimal arithmetic is complex to implement in hardware because of the larger range of decimal digits ([0, 9]) and the inefficiency of binary codes to represent decimal values, so that decimal floating point adder could not achieve better performance and smaller area than comparable binary floating point adder. Over the years, several designs for floating point decimal adder have been proposed for ASIC and FPGA platforms. FPGA implementations are generally based on techniques originally developed for VLSI architectures. The special builtin characteristics of FPGA architectures make it difficult to use many well-known methods to speedup computations (for example, carry-save and signed-digit arithmetics) [5]. Therefore, beyond adapting existing techniques we explore new decimal floating point adder algorithms more suitable for FPGAs [6].

This paper presents the algorithm, architecture and FPGA implementation of a novel unit to perform fast decimal floating point add. We have designed a decimal parallel floating point adder which results in an area-efficient implementation on Stratix IV chip of altera FPGA. The structure of the paper is as follows. In Section II we Discuss some relation work. In Section

Received April 23, 2013; Revised July 2, 2013; Accepted July 15, 2013

III, A new decimal floating point adder which performs addition is proposed, and introduces the resultant combinational and pipelined architectures. In Section IV presents the synthesis results of an implementation on a Stratix IV FPGA and comparison of the result with altera core design. Finally, the conclusions are summarized in Section V.

2. Related Works

Scientific and Engineering applications work on real numbers. Real numbers can be represented in computers in fixed-point representations where the fractional point has a fixed position in the number. This allows for using the same integer units to perform real number computations. However the range of real numbers in fixed-point representation is very small. Another alternative is to represent real numbers in floating-point representation. so floating-point arithmetic is very important . Addition is the basic but the most important function among the arithmetic operations.

2.1. Formats of Floating-point Numbers

The IEEE 754-1985 is the first IEEE standard for binary floating-point computations. The standard was later revised in 2008 (IEEE 754-2008) to incorporate decimal floating-point computations as well. The 754-1985 standard defines formats for representing floating-point numbers and special values (infinities, and NaNs) together with a set of floating-point operations that operate on these values. A floating-point number consists of three fields as shown in Figure 1: The sign bit, the fraction, and the exponent. Since the significand is normalized, the most significant bit (MSB) must be "1", hence it is not explicitly stored and it is called a "hidden 1". Only the fraction is explicitly represented [7][8].

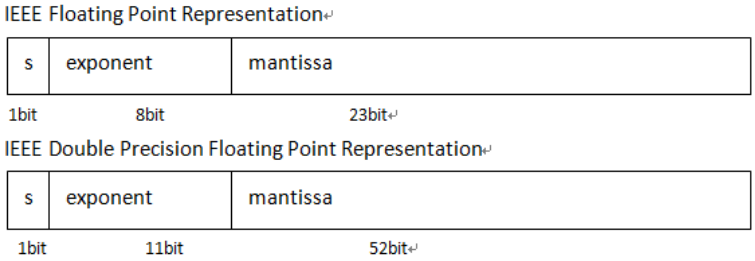

Figure 1. Significant and Exponent Representation in Single and Double Precision

The IEEE 754-1985 standard was revised in 2008 when the IEEE 754-2008 replaced it. It includes the entire original IEEE 754-1985 standard in addition to decimal floating-point computations. The standard defines arithmetic formats for binary and decimal floating-point data as shown in Table 1. It also defines interchange formats (encodings) for the floating-point data [9].

Table 1. The IEEE 754-2008 arithmetic formats

Name	Common Name	Base	Digits	Max. exponent	Min. exponent
bin16	Half precision	2	11	15	-14
bin32	Single precision	2	24	127	-126
bin64	Double precision	2	53	1023	-1022
bin128	Quadruple precision	2	113	16383	-16382
decl32	10	7	96	-95	dec32
decl64	10	16	384	-383	dec64
dec128	10	34	6144	-6143	decl128

2.2. Binary Floating-Point Adder

In general, floating point arithmetic implementations involve processing separately the sign, exponent and mantissa parts, and then combining them after rounding and normalization [9]. The hardware implementation of this arithmetic for floating point numbers is a complicated operation due to the normalization requirements. An implementation of double precision floating point adder has been shown here[10].

The steps for computing addition of two floating point numbers proceeds as follows:

1. Compare exponents and mantissa of both numbers. Decide large exponent & mantissa and small exponent & mantissa.
2. Right shift the mantissa associated with the smaller exponent, by the difference of exponents.
3. Add both mantissa if signs are same else subtract smaller mantissa from large one.
4. Do the rounding of the result after mantissa addition.
5. If the subtraction results in loss of most significant bit (MSB), then the result must be normalized. To do this, the most significant non-zero entry in the result mantissa must be shifted until it reaches the front. This is accomplished by a “Leading one detector (LOD)” followed by a shift.
6. Do normalization and adjust large exponent accordingly.
7. Final result includes sign of larger number, normalized exponent and mantissa.

2.3. Decimal Digit Adder

Financial and business applications use decimal based arithmetic to perform arithmetic operations. These applications require accuracy.We have mentioned that floating-point arithmetic introduces a roundoff error. This is because of the finiteness of the floating-point numbers representable in a given floating-point system. The accumulation of these roundoff errors can result in a totally different numbers than the number expected.Early solution to the above problem was to implement decimal floating-point arithmetic in software. However, to increase the performance of decimal floating-point arithmetic, decimal floating-point units were recently implemented in hardware[11].

conventional decimal adder and it is shown in Figure 2. For each decimal digit, it has two 4-bit binary adders and carry detection logic between the adders. The first level adders produce the binary addition results. If the result is greater than 9, a carry output is produced and the result of first level 4-bit adder is corrected by adding 6. Furthermore, the carry output is used as a carry input for the next digit. The main disadvantage of the conventional decimal adder is its low speed because all the first level 4-bit adders must wait for a number of 4-bit additions to get the right carry input [12].

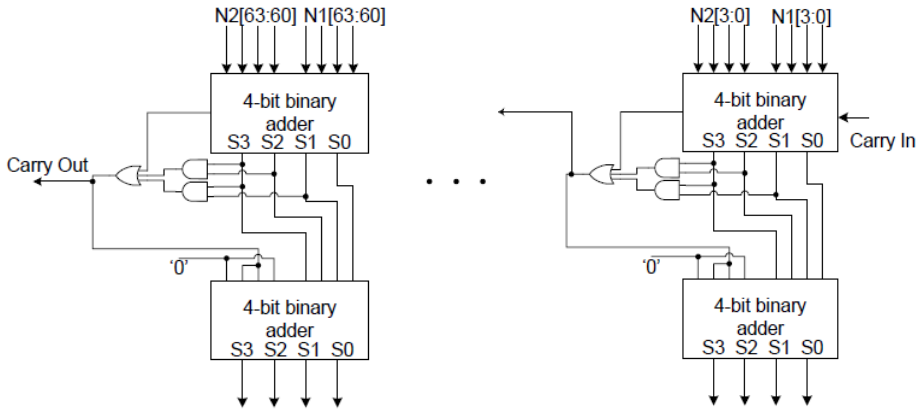

Figure 2. Conventional BCD Adder

3. Design and Implementation

This section presents our decimal floating-point adder, which uses a parallel method for decimal significand alignment and a Kogge-Stone parallel prefix network for significand addition

and subtraction. The decimal floatingpoint adder supports all the rounding modes and appropriate exceptions specified in IEEE 754.Figure 3 shows a high-level block diagram of our proposed decimal floating-point adder.

The 'Forward Format Conversion Unit' takes the two IEEE-encoded operands, A and B, and the operation, and produces the sign bits, SA1 and SB1, BCD significands, CA1 and CB1, biased exponents, EA1 and EB1, and effective operation, EOP (not shown in the figure). The 'Operand Alignment Calculation and Swapping Unit' (OACSU) takes these values and computes the result's temporary exponent, ER1, the right shift amount, RSA, and the left shift amount, LSA. It also swaps the significands if EB1 > EA1 . The two significant after swapping are denoted as CAS and CBS. Next, two 'Decimal Barrel Shifters' take these results and perform operand alignment on CAS and CBS. The two shifted significands, CA2 and CB2, are then corrected in the 'Precorrection Unit'. Based on the EOP signal and the prevailing rounding mode, the 'Pre-correction Unit' prepares the BCD operands for addition or subtraction and inserts a value needed for injection-based rounding. The corrected significands, CA3 and CB3, are then fed into the Kogge-Stone (K-S) network, which produces an uncorrected result, UCR, a digit-carry vector, C1, and flag vectors, F1 and F2. After this, the 'Post-correction Unit' converts UCR back into the BCD encoding to produce CR1. If needed, the 'Shift and Round Unit' shifts and rounds CR1 to produce the result's significands, CR2, and adjusts the temporary exponent, ER1, to produce the result's exponent,ER2. Simultaneously, the 'Sign Unit' and the 'Overflow Unit' compute the result's sign bit, SR1, and the overflow signal. The result's values, CR2, ER2, and SR1, are combined to generate the IEEE-encoded result in the 'Backward Format Conversion Unit'. This result and the original input operands are examined in the 'Post-processing Unit' to determine if a special result is needed, which happens if either one or both of the operands are Not-a-Number (NaN) or infinity. Further details on each of these units are provided below.

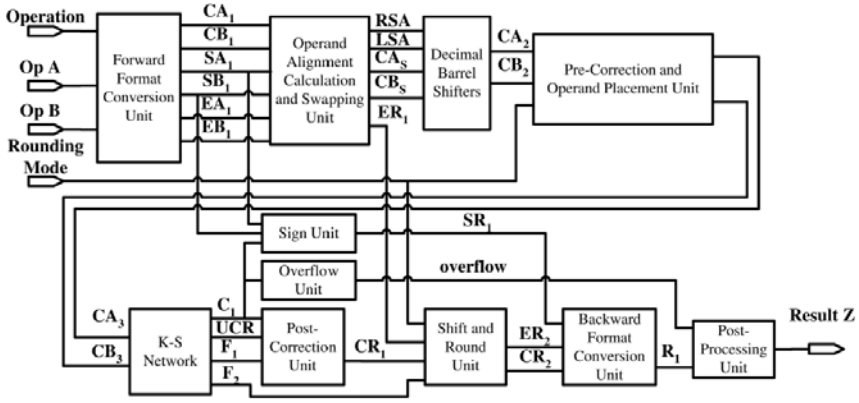

Figure 3. Block Diagram of the Proposed Decimal Floating-Point Adder

The core of our decimal floating-point adder operates on BCD significands. Therefore, converters are first employed to extract the DPD-encoded significands, binary exponents, and sign bits from both IEEE-encoded operands. Once unpacked, the two resulting significands are swapped if EB1 > EA1 and the temporary result exponent, ER1, is determined. The two significands after swapping are denoted as CAS and CBS where the subscript "S" refers to Swapped. The number of leading zeros in the significand with the larger exponent, CAS, is denoted as LAS. In parallel with swapping the operands, the effective operation (EOP) is determined by the Boolean equation EOP = SA1□SB1□Operation, where EOP and Operation are zero for addition and one for subtraction.

Decimal operand alignment is more complex than its binary counterpart because decimal numbers are not normalized. This leads to both left and right shifts to ensure the rounding location is in a fixed digit position. To correctly adjust both operands to have the same exponent, the follow computations are performed:

LSA = min(|EA1 − EB1|, LAS)

RSA = min(max(|EA1 − EB1| − LAS, 0), 19)
ER1 = EAS − LSA

The above equations produce a left shift amount, LSA, which indicates by how many digits CAS should be left shifted. LSA is equal to the absolute value of the exponent difference, |EA1 − EB1|, but is limited to LAS digits so that the left-shifted significand, CA2, does not have more than 16 digits. The RSA value indicates by how many digits CBS should be right shifted in order to guarantee that both numbers have the same exponent, ER1, after significand alignment. RSA is limited to 19 digits, since the right shifted significand, CB2, contains 16 digits plus guard and round digits and a sticky bit, The temporary result exponent, ER1 is simply the larger exponent, EAS, after it has been adjusted to compensate for the left shift amount, LSA.

After computing the left and right shift amounts, two decimal barrel shifters, which shift by multiples of four bits, perform the operand alignment. The significands after alignment are denoted as CA2 = left shift(CAS, LSA) and CB2 = right shift(CBS,RSA). As noted previously, CA2 is 16 digits, and CB2 is 16 digits plus a guard digit, G, a round digit, R and a sticky bit, S, as shown in Figure 4.

Once shifted, an injection value based on the sign bit and prevailing rounding mode is inserted into the Round and Sticky digit positions of CA2 to form CA_2, which is a 19-digit BCD number. The injection value is determined by equations similar to those developed for binary floating-point addition and is used to facilitate correct rounding.

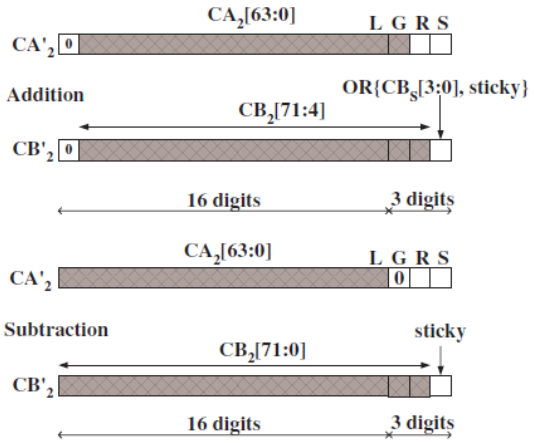

Figure 4. Operand Placement for Decimal Addition

Because both operands are corrected, a binary Kogge-Stone (K-S) network can be used to generate the proper carry into each digit. Figure 5 illustrates how the original K-S network is extended to detect trailing nines. The traditional injection based rounding method uses conditional adders to compute the uncorrected sum and the uncorrected sum plus one and then uses the MSDs of these values and the carry into the LSD of the uncorrected sum to select the proper sum. To reduce area, our adder instead uses the flagged-prefix method to compute the uncorrected sum and the uncorrected sum plus one.

The temporary result generated from the Kogge-Stone network requires a post-correction unit to convert the uncorrected result, UCR back to BCD to produce CR1. The rules for performing this correction are defined in Figure 6.

Overflow occurs when the addition or subtraction of two operands exceeds the maximum representable value in the destination format. Typically, the adder needs to check the carry from the MSD after incrementing the corrected result to see if an overflow occurs. With the injection-based rounding method, however, since the injection correction value does not generate another carry from the MSD, the overflow signal can be generated by examining the final exponent from the operand alignment unit, ER1, and the MSD of the corrected result, CR1.

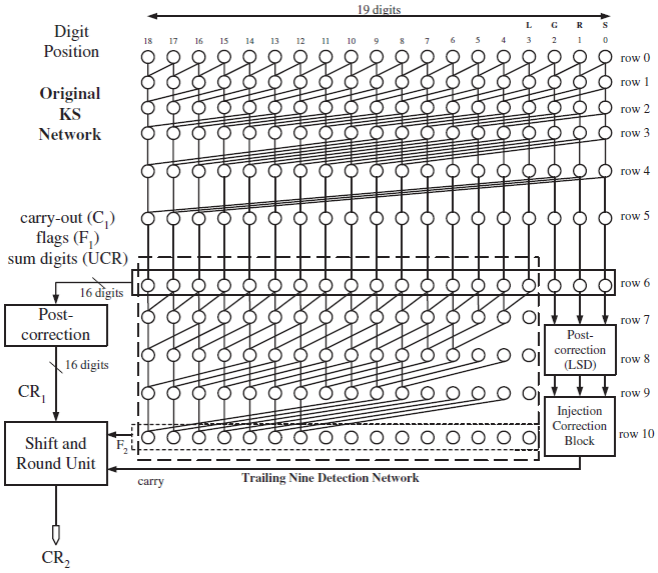

Figure 5. Conceptual View of the Flag-based Logic and the Kogge-Stone Network.

Rule 1: Rule enforced when performing an effective addition operation:

Add 1010 (correction of -6) to $(UCR)_i$ *in digit* i *for which* $(C_1)_{i+1}$ *is 0*

Rule 2: Rule enforced when performing an effective subtraction operation:

If (MSB of $C_1 \equiv 1$*) //the sign of the result is positive*

1. Invert bits in *UCR* for which the corresponding bit in F_1 is one. This increments *UCR*.
2. Add '1010' to the above result in digit i for which $(C_1)_{i+1} \oplus (F_1)_{3i} \equiv 0$

Else // the sign of the result is negative

1. Invert all sum bits
2. Add '1010' to the above result in digit i for which $(C_1)_{i+1} \equiv 1$

Figure 6. The rules for performing this correction

The 'Overflow Unit', also generates a signal to determine if the final result should be infinity or the maximum representable value of the destination format, based on the rounding mode and the sign of the result. Using this signal and the overflow flag, the final result can be modified, if needed, in the 'Post Processing Unit'.

We have fully pipelined the combinational architecture as follows: each level of the adder tree is placed in a pipeline stage. Besides, each carry-ripple adder is pipelined in chunks of k bits at most. The total number of pipelined stages is equal to [(4p+l)/k]+[log2(m)]. A significant amount of registers is required for input synchronization. To reduce the hardware cost, these synchronization registers are placed in the first pipeline level of the tree and packed together as 16-bit shift register LUTs.

4. Experiments and Results Comparison

The three decimal floating point adders have been descripted in the verilog HDL and implemented on The hardware EP4SGX230 wich is Stratix IV FPGA. Stratix FPGAs have HardCopy ASIC equivalent devices. HardCopy ASICs provide a path to low-cost volume production with low risk through FPGA prototyping of your design. Stratix series FPGAs are also

ideal for the prototyping and verification of standard-cell ASICs. Table 2 compares Critical path Latency and the total LUT's of the three designs.

From Figure 7, the proposed DFP adder has about 19.2 percent less delay and 11.53 percent less LUT's than the design presented in [2], and about 9.76 percent less delay and 8.85 percent less LUT's than the design presented in [3].

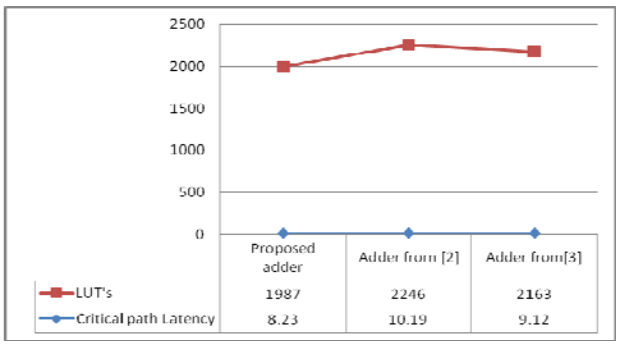

Figure 7. The three adders performance comparison

Futher more, we compare the proposed module with the altera core adder. Implementation results are shown on Figure 8.The proposed design has been implemented for various latencies. The data for altera Core adder has also been shown for various available latencies, to have better idea of proposed design. The proposed design is taking approximately same hardware (in terms of number of LUT's and FF's count) as of altera module, but have better performance speed with similar latencies. The proposed design is achieving a speed of 358 MHz than 286 MHz for altera core for a latency of 12, which shows a significant performance improvement in the proposed design.

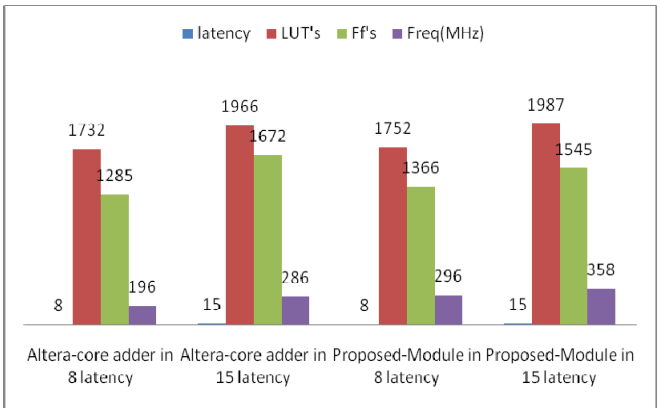

Figure 8. The decimal floating point adder on Stratix IV

5. Conclusion

This paper has shown an efficient implementation of a new parallel decimal floating point module on FPGA, We described in detail several novel components in the designs. we have provided a detailed analysis on our synthesis results and a comparison between a altera-core adder design and the others two decimal floating point adders. Implemention results show that the proposed adder design has 25.1% less latency and 1.2% less LUT's than the altera-core design. We can also find the proposed adder design has the better performance than the other two decimal floating point adders.

Acknowledgement

This work was supported by F201232.

References

[1] Asger Munk Nielsen, David W. Matula, Chung Nan Lyu, Guy Even. An IEEE compliant floating-point adder that conforms with the pipeline packet-forwarding paradigm. *IEEE Transactions on Computers*. 2000; 49: 33-47.
[2] J Moskal, E Oruklu, and J Saniie. *Design and Synthesis of a Carry-Free Signed-Digit Decimal Adder*. Proceedings of the IEEE Symposium on Circuits and Systems. 2007; 1089-1092.
[3] K Yehia, HAH Fahmy, and M Hassan. *A Redundant Decimal Floating-Point Adder*. Proceedings of Asilomar Conference on Signals, Systems & Computers. 2010; 1144-1147.
[4] Amir Kaivani and Ghassem Jaberipur. Fully redundant decimal addition and subtraction using stored-unibit encoding. *Integration, the VLSI journal*. 2010; 43(1): 34-41.
[5] EM Schwarz, JS Kapernick, and MF Cowlishaw. Decimal floating-point support on the IBM z10 processor. *IBM Journal of Research and Development*. 2009; 53: 231-239.
[6] J Thompson, N Karra, and MJ Schulte. *A 64-bit decimal floating-point adder*. Proceedings of the IEEE Computer Society Annual Symposium on VLSI. 2004; 297-298.
[7] Liang-Kai Wang, MJ Schulte, JD Thompson, and N Jairam. Hardware Designs for Decimal Floating-Point Addition and Related Operation. *IEEE Transactions on Computers*. 2009; 58: 322-335.
[8] G Even and PM Seidel. A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication. *IEEE Trans. Computers*. 2000; 49(7): 638-650.
[9] A Vazquez, E Antelo and P Montuschi. Improved Design of High-Performance Parallel Decimal Multipliers. *IEEE Transactions on Computers*, 2010; 59(5): 679-693.
[10] Ghassem Jaberipur and Saeid Gorgin. *A Nonspeculative Maximally Redundant Signed Digit Adder*. Proceedings of The 13th international CSI Computer Conference. 2008; 235-242.
[11] Sonia Gonzalez-Navarroa, Javier Hormigoa, Michael J. Schulteb. A study of decimal left shifters for binary numbers. *Information and Computation*. 2012; 216: 47-56.
[12] Saeid Gorgin, Ghassem Jaberipur. A fully redundant decimal adder and its application in parallel decimal multipliers. *Microelectronics Journal*. 2009; 40(10): 1471-1481