Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2013) 21: 500 – 512
© TÜBİTAK
doi:10.3906/elk-1110-60

Design and evaluation of schemes for computing sum of squares in fixed point

Metin Mete ÖZBİLEN[1,*], **Miloš Dragutin ERCEGOVAC**[2]
[1]Department of Computer Engineering, Mersin University, Mersin 33343, Turkey
[2]Department of Computer Science, University of California, Los Angeles, CA 90095, USA

Received: 27.10.2011 • **Accepted:** 28.02.2012 • **Published Online:** 22.03.2013 • **Printed:** 22.04.2013

Abstract: Several schemes for computing the sum of squares for fixed point numbers are designed, synthesized, and evaluated in this study. The schemes use radix-2 folding, radix-4 folding, and radix-4 dual recoding approaches for squaring. The schemes have been modeled in hardware description language, simulated, and synthesized using Cadence SOC81 in 45 nm and 90 nm libraries and with Mentor Graphics Leonardo Spectrum for 180 nm. We show delay and area for 16-, 24-, and 32-bit operands. After hard-wire and software optimizations, it is seen that schemes with radix-4 give better results, especially in area.

Key words: Squaring, sum of squares, folding, radix-2, radix-4, argument recoding

1. Introduction

Squaring operations and sum of squares are frequently used in scientific calculations. They are used in geometric calculations in 2D and 3D environments found in graphics, video games, and multimedia animation. For example, the calculation of the distance between 2 objects requires the sum of squares. Squaring is also frequently used in signal processing and the sum of squares takes place in calculation of waveforms in digital signal processors [1]. Squaring techniques have been considered in the past and efforts to increase their performance are continuing. Some of the common methods are radix-2 and radix-2 folding, radix-4 folding, and radix-4 dual recoding [2, 3, 4].

In this paper we provide efficient schemes for computing the sum of 2 squares by integrating squaring with addition operations. Several techniques for squaring of fixed-point numbers are reviewed to determine the suitable schemes for calculating the sum of squares. Various addition techniques have been considered and adaptations are made to find a good fit to the sum of squares design. The squaring arrays and adder schemes are implemented in hardware description language (HDL) and their correctness is checked by ModelSim simulation software. The gate level delays are determined using Quartus software and synthesized with Cadence SOC81 software. An HDL generator library is developed in C/C++ language to speed up the design steps. With the help of this generator library, different instances of squaring and addition schemes for 16, 24, and 32 bits can be implemented quickly.

2. Squaring methods

We consider the following methods for calculating the square of a fixed-point number: (i) a standard radix-2 squaring with bit-array simplifications [2], (ii) a radix-4 folding scheme with Booth recoding [3], and (iii) a

[*]Correspondence: mmozbilen@mersin.edu.tr

radix-4 with dual coding [4]. Most of the squaring methods are based on multiplication operation and utilize partial squares similar to partial products in multiplication. These partial squares are generally in either radix-2 or radix-4 format, placed in bit arrays. The sum of partial square arrays is calculated with reduction methods such as the Wallace [5], Dadda [6], or Carry-Save array. The details of these methods are discussed in following section.

2.1. Radix-2 squarer

The bit-array for x^2 consists of a diagonal with entries $x_ix_i = x_i$ and regions above and below the diagonal. Because of the commutative property of the *and* operator, the sums of entries above and below the diagonal are equal. Therefore, the bit array containing the diagonal elements and the bit array either above or below the diagonal is shifted left to compute their sum. The transformed bit array and reduced array for a 6-bit squarer is shown in Figure 1.

11	10	9	8	7	6	5	4	3	2	1	0
						x_5x_0	x_4x_0	x_3x_0	x_2x_0	x_1x_0	x_0
					x_5x_1	x_4x_1	x_3x_1	x_2x_1	x_1	x_0x_1	
				x_5x_2	x_4x_2	x_3x_2	x_2	x_1x_2	x_0x_2		
			x_5x_3	x_4x_3	x_3	x_2x_3	x_1x_3	x_0x_3			
		x_5x_4	x_4	x_3x_4	x_2x_4	x_1x_4	x_0x_4				
	x_5	x_4x_5	x_3x_5	x_2x_5	x_1x_5	x_0x_5					
	x_5x_4	x_5x_3	x_5x_2	x_5x_1	x_5x_0	x_4x_0	x_3x_0	x_2x_0	x_1x_0		x_0
	x_5		x_4x_3	x_4x_2	x_4x_1	x_3x_1	x_2x_1		x_1		
			x_4		x_3x_2		x_2				
					x_3						

(a)

11	10	9	8	7	6	5	4	3	2	1	0
	x_5x_4	x_5x_3	x_5x_2	x_5x_1	x_5x_0	x_4x_0	x_3x_0	x_2x_0	x_1x_0		x_0
	x_5		x_4x_3	x_4x_2	x_4x_1	x_3x_1	x_2x_1		x_1		
			x_4	x_3x_2	x_3x_2'		x_2				

(b)

Figure 1. Bit array simplification in squaring of magnitudes ($n = 6$): a) bit array after using identities $x_ix_i = x_i$ and $x_ix_j + x_jx_i = 2x_ix_j$, b) further reduction in number of rows after using identity $x_ix_j + x_j = 2x_ix_j + x_ix_j'$.

2.2. Radix-4 folding

Let x be an 8-bit signed number to be squared. Its 2's complement representation is

$$x = -x_7 2^7 + x_6 2^7 + \cdots + x_1 2^1 + x_0 2^0 \tag{1}$$

and its radix-4 Booth representation is

$$X = -X_3 2^6 + X_2 2^4 + X_1 2^2 + X_0 2^0 \tag{2}$$

where the digit $X_i \in \{-2, -1, 0, 1, 2\}$ is defined as

$$X_i = -2x_{2i+1} + x_{2i} + x_{2i-1}. \tag{3}$$

The square of X can be calculated using Eq. (4) where both operands are Booth-encoded. Booth multiplication permits only one operand to be Booth-encoded. The details of how this problem is resolved are described in [3].

$$\begin{aligned} X^2 = {} & \left(X_3 2^6 + X_2 2^4 + X_1 2^2\right) \times X_0 2^0 + X_0 \times X_0 2^0 + \\ & \left(X_3 2^4 + X_2 2^2\right) \times X_1 2^4 + X_1 \times X_1 2^4 + \\ & \left(X_3 2^4\right) \times X_2 2^8 + X_2 \times X_2 2^8 + \\ & X_3 \times X_3 2^{12} \end{aligned} \tag{4}$$

Eq. 4 can be rearranged as

$$\begin{aligned} X^2 = {} & \left(P_0 2^3 + C_0\right) + \\ & \left(P_1 2^3 + C_1\right) 2^4 + \\ & \left(P_2 2^3 + C_2\right) 2^8 + \\ & C_3 2^{12} \end{aligned} \tag{5}$$

where

$$\begin{aligned} & C_i = X_i \times X_i,\ i = 1, \ldots, 4 \\ & P_i = \left(-X_7 2^{5-2i} + X_6 2^{4-2i} + \cdots + X_{2i+2} 2^0 + X_{2i+1} 2^0\right) X_i, \\ & i = 0, \ldots, 2. \end{aligned} \tag{6}$$

The square is then calculated by summing C_is and P_is. The C_i terms can have the value of $\{0, +1, +4\}$ The placement of these partial squares in the bit array is shown in Figure 2, where S_1s are the injecting constants with value '1' due to 2's complement of P_i terms. The sign extension technique used here is presented in [7].

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
						P_{06}	P_{05}	P_{04}	P_{03}	P_{02}	P_{01}	P_{00}	C_{02}		C_{00}
				P_{14}	P_{13}	P_{12}	P_{11}	P_{10}	C_{12}		C_{10}				
		P_{22}	P_{21}	P_{20}	C_{22}		C_{20}								
	C_{32}		C_{30}												
S_1	S_1		S_1		S_1	S_1									
S_1	C_{32}	P_{22}	P_{21}	P_{14}	P_{13}	P_{06}	P_{05}	P_{04}	P_{03}	P_{02}	P_{01}	P_{00}	C_{02}		C_{00}
	S_1		S_1	P_{20}	C_{22}	P_{12}	P_{11}	P_{10}	C_{12}		C_{10}	P_{00}	C_{02}		C_{00}
					S_1	S_1	C_{20}								

Figure 2. Bit array simplification in radix-4 Booth folding squaring.

3. Radix-4 dual recoding

In Booth-recoded multiplication, only one operand is coded into a radix-4 digit string. The second operand remains in binary. In squaring only one operand x is needed, which is already coded as X. The dual coding method [4] uses distinct asymmetric roles of a single operand. In this method, each coded digit is multiplied with its coded companion. Booth-folding and Booth radix-4 [3] recoding operations can be combined to reduce the partial square array. These methods utilize the reoccurrence of

$$\left(x'\right)^2 = x^2 - d\left(2x' + d\right), \tag{7}$$

where d is the low order Booth radix-4 digit of x. The Booth folding method is useful for the squaring of integers where low-order bits are needed [3].

The dual coding method utilizes Eq. (7) and it is determined by

$$x^2 = \left(x' + d\right)^2, \tag{8}$$

where d is the high-order radix-4 digits of x and x is the rounded-off tail.

In Eq. (7), d denotes the squarer leading digits, $(2x' + d)$ is the squarand, and $d\,(2x' + d)$ is the partial square [4]. The result, x^2, is obtained by adding partial squares.

The square of x is calculated using the following equation:

$$x^2 = sum_{i=1}^{\lfloor n+1/2 \rfloor} d_i q_i 16^{-i}, \tag{9}$$

where n is number of bits in x, d_i is the radix-4 Booth-coded digit of x, and

$$q_i = \begin{cases} x_{2i}.x_{2i+2}\ldots x_{n-1} & \text{for } x_{2i-1} = 0 \\ x'_{2i}.x'_{2i+2}\ldots x'_{n-2}x''_{n-1} & \text{for } x_{2i-1} = 1, \end{cases} \tag{10}$$

where x'_{2i} is 1's complement of x_{2i} and x''_{n-1} is 2's complement.

Once the partial squares are formed, they are placed in the squaring array as in Figure 3.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Q_{00}	Q_{01}	Q_{02}	Q_{03}	Q_{04}	Q_{05}	Q_{06}		Q_{20}	Q_{21}	Q_{22}	Q_{30}	Q_{31}	Q_{32}	Q_{33}	Q_{34}
				Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}		S_1					
								S_1							

Figure 3. Bit array simplification in radix-4 dual Booth recoder squaring.

4. Setup for sum of squares

We developed several designs to determine which squaring scheme was better for the calculation of the sum of squares. The designs are based on the squaring methods described in the previous section, reduction schemes, and final adder schemes.

Similar reduction and final adder schemes were applied to each squaring scheme for revealing the performance of the squaring operation. We looked at the effect of different argument precisions (16, 24, and 32) and investigated different types of carry-lookahead adders (CLAs) to obtain the best results. The following CLAs were implemented and analyzed: Brent–Kung [8], Sklansky [9], and Kogge–Stone [10]. The experimental results are shown in Table 1 and Table 2. Each is implemented with VHDL and their correctness was checked by simulation method. The details of each design are described below.

Table 1. Experimental results for carry-lookahead adders (delay).

Delay (ps)	16-bit	24-bit	32-bit	48-bit	64-bit
Brent–Kung	82	126	221	345	423
Sklansky	82	140	219	349	456
Kogge–Stone	115	203	290	498	706

Table 2. Experimental results for carry-lookahead adders (area).

Area (Gate)	16-bit	24-bit	32-bit	48-bit	64-bit
Brent–Kung	735	689	989	999	999
Sklansky	735	625	988	999	999
Kogge–Stone	287	321	342	370	401

4.1. Radix-2 folding squaring

The data flow and functional blocks are shown in Figure 4. The function of each block is as follows: *GEN-P*s generate partial squares. Partial squares are generated as in Figure 1. In reduction arrays (*RAs*), partial squares are reduced to sum and carry vectors using (3:2) counter, (2:2) counter, and (4:2) compressors. The reduction scheme is shown in Figure 5.

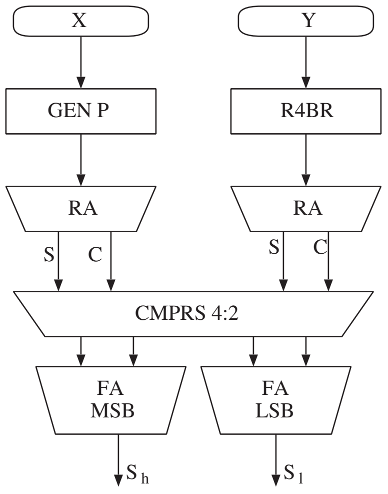

Figure 4. Sum of squares using radix-2 folding method.

CMPRS is formed from (4:2) compressors. They reduce results from RAs into a sum and a carry vector. *FA*s are the final adders, separated into halves of most significant bits (MSB) and least significant bits (LSB) for better performance.

The best results are obtained using separate Kogge–Stone adders for the MSB and LSB halves. The synthesis results are given in Table 3.

Table 3. Synthesis Results for radix-2 sum of squares.

	16-bit	24-bit	32-bit
w/FA 45 nm typical			
Delay (ps)	1453	1744	1824
Area (gate)	1084	2177	3445
w/FA 90 nm typical			
Delay (ps)	1453	1811	1929
Area (gate)	1054	2101	3246
w/FA 180 nm typical			
Delay (ps)	2530	2870	3590
Area (gate)	2897	5970	10941

4.2. Radix-4 folding squaring

The implemented design is shown in Figure 6. The explanation of functional blocks and data flow is as follows.

The *R4BR* is the radix-4 Booth encoder. It encodes input according to Eq. (3). *GEN-W* does recoding

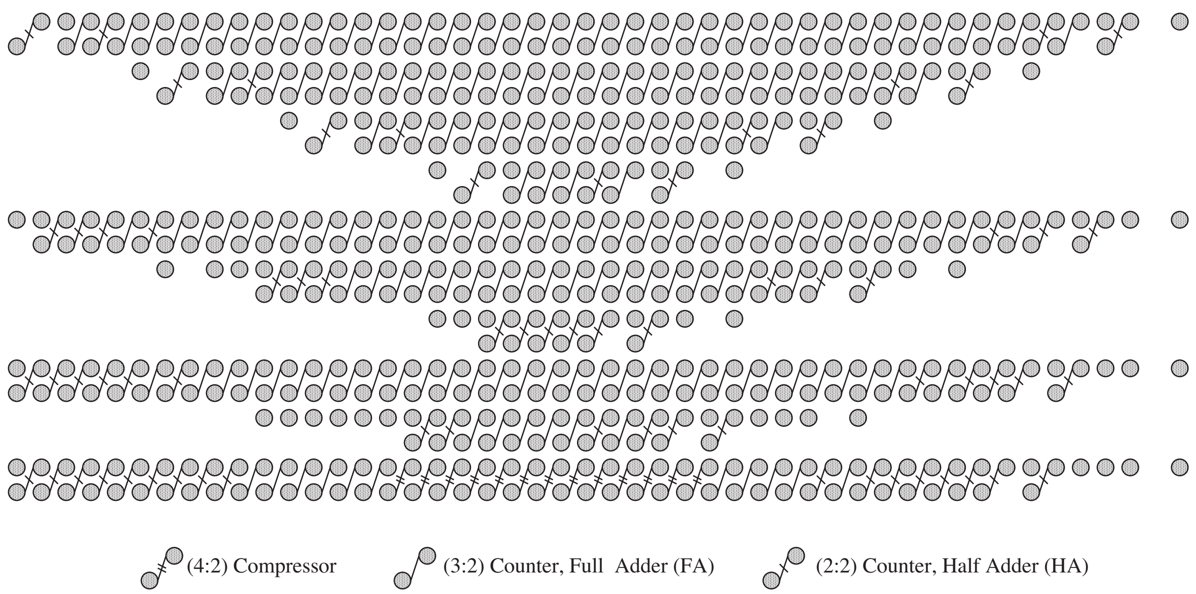
(4:2) Compressor
(3:2) Counter, Full Adder (FA)
(2:2) Counter, Half Adder (HA)

Figure 5. Radix-2 folding squarer reduction tree.

of inputs according to Eq. (11).

$$W_i = -b_7 2^{5-2i} + b_6 2^{4-2i} + ... + b_{2i+2} 2^0. \tag{11}$$

W_is are used in calculation of P_i in Eq. (6). *GEN-C* generates the C_is in Eq. (6). Alignment and complementing of partial products due to radix-4 Booth recoding is done in *SHF-CMP*. *GEN-P* builds the partial squaring tree to be summed. The details of the design are given in [3]. *RA* is the unit where the partial tree is reduced into a carry and a save vector. The reduction is carried out using (3:2) counters, (2:2) counters, and (4:2) compressors. The reduction scheme is shown in Figure 7. The reduction scheme is kept very similar to the radix-2 folding due to analogy and better comparison. *CMPRS* consists of (4:2) compressors. The results from squarers are flattened to a carry and a save vector. These carry and save vectors are summed in the final adders (*FAs*). Similar to the radix-2 squarer, it is divided into 2 portions to speed up calculation.

The design is synthesized with Cadence SOC81 for 45 nm and 90 nm libraries and 180 nm with Mentor Graphics Leonardo Spectrum. The synthesis results are given in Table 4.

Table 4. Synthesis results for radix-4 folding sum of squarer.

	16-bit	24-bit	32-bit
w/FA 45 nm typical			
Delay (ps)	1480	1823	1910
Area (gate)	1071	2361	3921
w/FA 90 nm typical			
Delay (ps)	1468	1754	1824
Area (gate)	1013	2160	3602
w/FA 180 nm typical			
Delay (ps)	2590	3460	3540
Area (gate)	2139	5371	8307

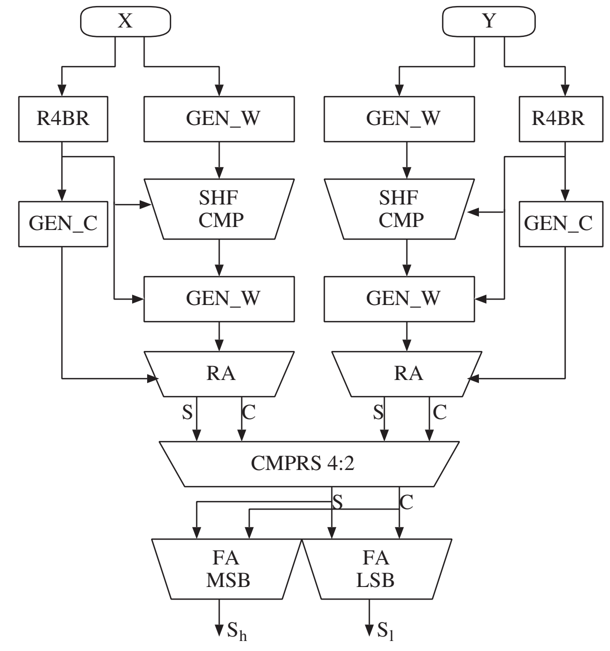

Figure 6. Sum of squares using radix-4 folding squarer method.

4.3. Radix-4 dual recoding

The last method implemented is shown in Figure 8. The explanation of each block function and data flow is as follows.

R4BR recodes input into radix-4 Booth recoding, similar to the radix-4 folding method. It is the implementation of Eq. (3). The truth table for this block is given in Table 5, where b_{2i-1}, b_{2i}, b_{2i+1} denotes adjacent input bits where $b_{-1} = 0$; the column labeled with 'B' denotes the corresponding Booth value and the next 3 columns denote the recoded Booth value. For 24-bit input the block generates twelve 3-bit output. *DE* means "dual encoder" and recodes the input for squaring. It implements Eq. (10). Partial squares are generated in *Wx210* blocks. The dual recoded inputs are multiplied here to generate partial squares. The outputs of the *R4BR* and *DE* are multiplied using the wired multiplication method. Wired multiplication is a slightly modified

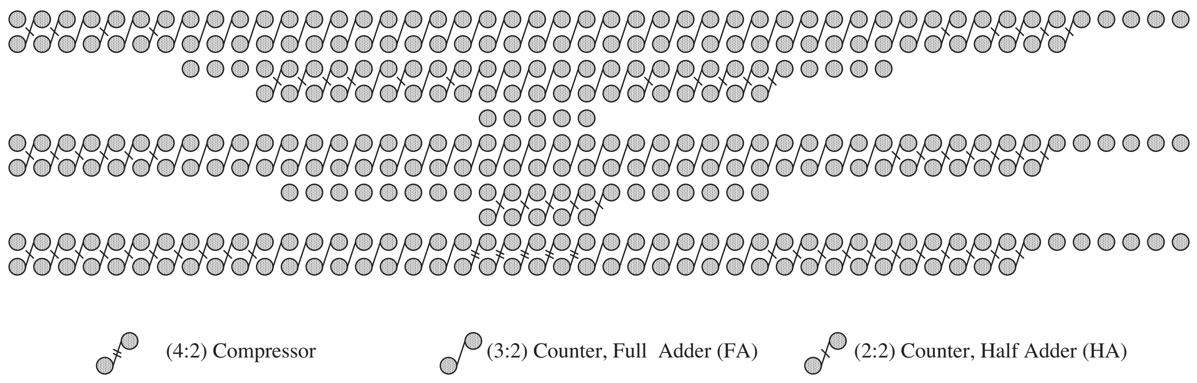
(4:2) Compressor
(3:2) Counter, Full Adder (FA)
(2:2) Counter, Half Adder (HA)

Figure 7. Radix-4 folding squarer reduction tree.

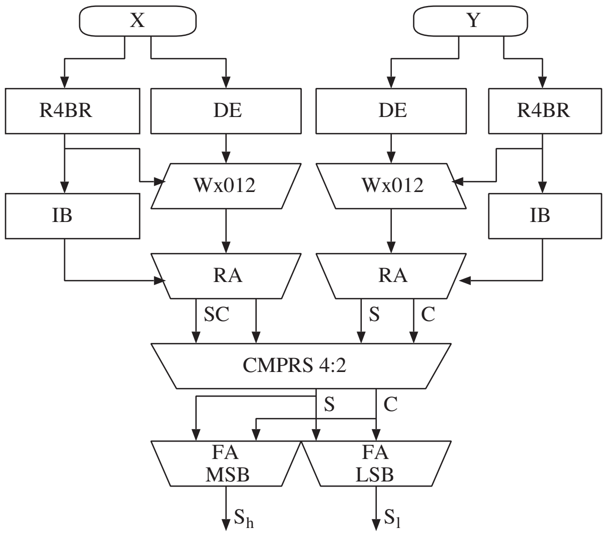

Figure 8. Sum of squares using radix-4 dual recoding squarer method.

multiplexer and is shown in Figure 9, where Q_i are dual encoded input bits with $Q_{-1} = 0$ and d_1 and d_2 are radix-4 coded inputs.

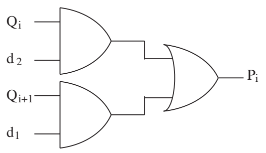

Figure 9. Wired multiplication cell.

IB is the inversion bit; it is only a buffer that holds n bits of *R4BR* block. The partial squares are reduced to a carry and a sum vector in *RA*. The reduction is performed similar to the radix-4 folding and radix-2 folding methods. The reduction scheme is shown in Figure 10. *FAs* are the final adders as in the radix-4 folding method and radix-2 folding method.

Table 5. Truth table for *R4BR* block.

b_{2i-1}	b_{2i}	b_{2i+1}	B	n	d_1	d_2
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	1	0	0	1
0	1	1	2	0	1	0
1	0	0	$\bar{2}$	1	1	0
1	0	1	$\bar{1}$	1	0	1
1	1	0	$\bar{1}$	1	0	1
1	1	1	0	0	0	0

The design is synthesized with Cadence SOC81 for 45 nm and 90 nm cell libraries and Mentor Graphics Leonardo for 180 nm. The results are given in Table 6.

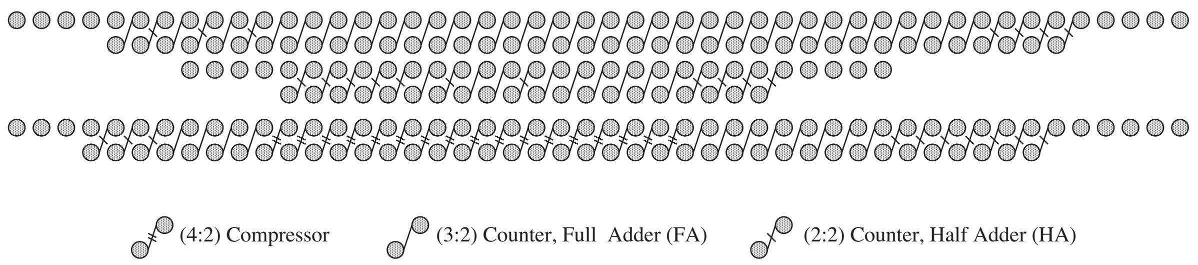
(4:2) Compressor
(3:2) Counter, Full Adder (FA)
(2:2) Counter, Half Adder (HA)

Figure 10. Radix-4 dual recoding squarer reduction tree.

Table 6. Synthesis results for radix-4 dual recoding sum of squarer.

	16-bit	24-bit	32-bit
w/FA 45 nm typical			
Delay (ps)	1550	1765	2065
Area (gate)	983	2412	4233
w/FA 90 nm typical			
Delay (ps)	1472	1725	2119
Area (gate)	954	1977	3355
w/FA 180 nm typical			
Delay (ps)	2440	2790	3380
Area (gate)	2154	4398	8085

5. Comparisons of methods

The methods are compared analytically and experimentally and their advantages and disadvantages are discussed in this section. The radix-2 folding method has an advantage in preparation time of the partial squares. The partial squares are prepared simultaneously; also, using the folding method reduces the number of partial squares. Although radix-4 folding has the advantage of the folding method, the radix-4 Booth recoding and preparation steps of terms take longer than in the radix-2 folding scheme. Partial squares become ready after the shifting and complementing process. Radix-4 dual encoding also has a disadvantage similar to radix-4 recoding. It also has a dual encoder, but both recodings take place simultaneously, unlike in radix-4 folding. Comparative results of the 24-bit sum of squares in 90 nm for units and whole designs with final adders are given in Table 7 and Table 8, respectively.

All of the methods in this study were based on fixed-point arithmetic; therefore, the methods can be easily modified to operate on floating point numbers. Methods can be adjusted to operate on the mantissa part

Table 7. Comparative results of units for 24-bit sum of squares in 90 nm.

Unit	Delay	Arrival	Unit	Delay	Arrival	Unit	Delay	Arrival
R4BR GEN-W	163	163	R4BR DE IB Wx012	320	320	GEN-P	48	48
GEN-C SHF-CMP GEN-P	97	260	RA	549	869	RA	854	902
RA	634	892						

Table 8. Comparative results of sum of squares in 90 nm. Delay (ns), area (gates), power (nW).

Name of Design	16-bit			24-bit			32-bit		
	Delay	Area	Power	Delay	Area	Power	Delay	Area	Power
Radix-2 Squarer	1453	1054	661,057	1811	2101	1,521,541	1929	3246	2,780,204
Radix-4 Folding	1468	1013	560,238	1754	2160	1,238,931	1824	3602	2,435,524
Radix-4 Dual Decoding	1472	954	635,276	1725	1977	1,314,889	2119	3355	2,787,366

of the floating point numbers. As a fixed point design, methods can be applicable to microcontrollers and digital signal processing processors in the calculation of geometric function such as the hypotenuse of triangles. When modified to floating point, they can be applicable as multimedia extensions from general propose processors to graphical processing units. Sums of squares are frequently used in many geometric and trigonometric operations, such as normalization of vectors.

Radix-2 folding has reduction arrays with twice as many rows as the radix-4 folding and radix-4 dual recoding methods. This seems to be a disadvantage, but the simultaneous behavior of counters used in reduction compensates most of this delay. The half-height reduction arrays of both radix-4 recoding methods are an advantage in both delay and area. Radix-4 dual recoding has the shortest reduction array.

The remaining parts of the designs are very similar to each other. The reduced arrays are flattened by (4:2) compressors and the flattened results are summed by CLAs in *FA*s. The comparable results for 24 bits are given in Table 7. The full comparison of methods with the FA is shown in Figure 11 and Figure 12.

For better objectivity and to see the effect of the final adder, designs were also compared without fnal adders. These results are show in Figure 13 and Figure 14.

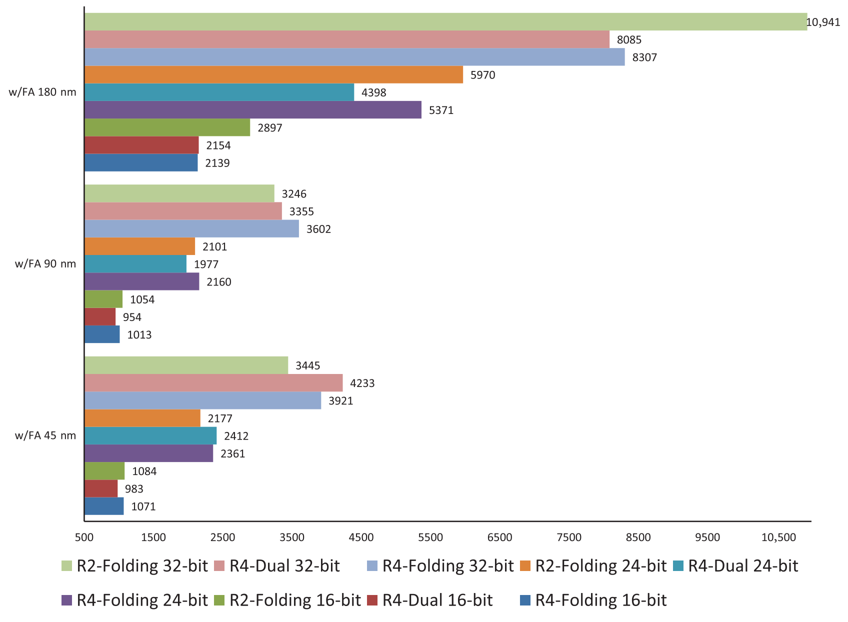

Figure 11. Comparison of areas for sum of squares designs.

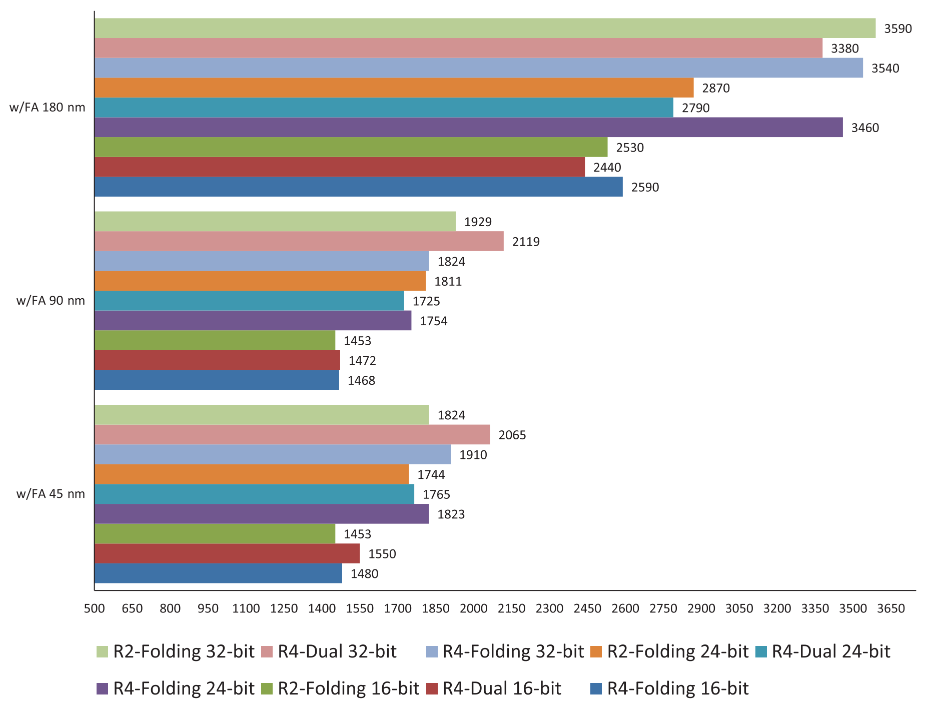

Figure 12. Comparison of delays for sum of squares designs.

6. Results and conclusion

In this paper we implemented and evaluated several squaring methods for fixed-point operands to be used in calculating the sum of 2 squares. The experimental results show that final adder has a major effect in both the

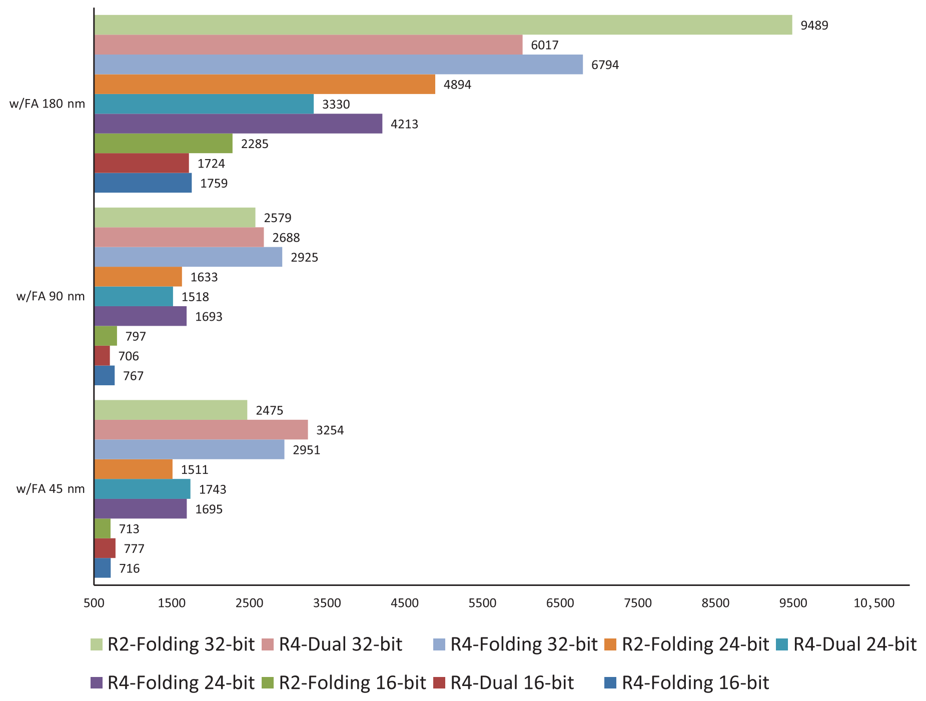

Figure 13. Comparison of areas for sum of squares designs without final adder.

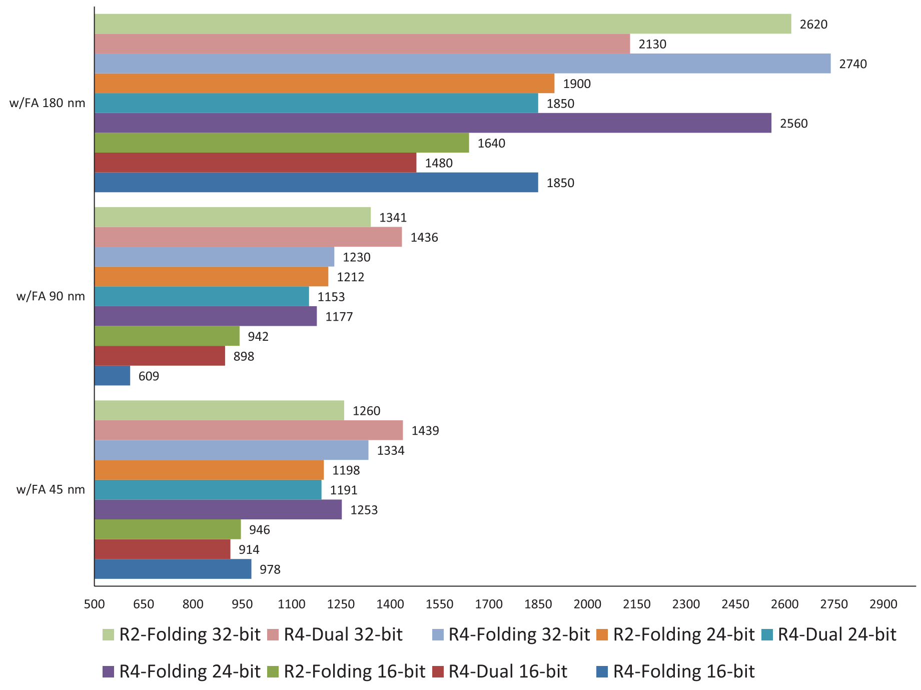

Figure 14. Comparison of delays for sum of squares designs without final adder.

delay and area of the approaches considered. When the designs are used in calculation of multistep processes like the square root of sum of squares, output in the form redundant of carry and sum vectors, produced by the reduction step, is sufficient. The results show that the radix-4 dual recoding setup gives the best values in both area and delay. These values also meet goals for square root calculation using redundant adders. Designs can be pipelined for further performance improvements. This study can be extended with a division or reciprocal square root design to provide a full geometric rotation operation or vectorormalization in graphics applications.

Acknowledgments

M. Özbilen would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for financial support during his stay in the Computer Science Department of the University of California, Los Angeles, September 2010 to August 2011.

References

[1] J. Pihl, E.J. Aas, “A multiplier and squarer generator for high performance DSP applications”, IEEE 39th Midwest Symposium on Circuits and Systems, Vol. 1, pp. 109–112, 1996.

[2] M.D. Ercegovac, T. Lang, Digital Arithmetic, San Francisco, Morgan Kaufmann, 2004.

[3] A.G.M. Strollo, D. De Caro, “Booth folding encoding for high performance squarer circuits”, IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 50, pp. 250–254, 2003.

[4] D.W. Matula, “Higher radix squaring operations employing left-to-right dual recoding”, 19th IEEE Symposium on Computer Arithmetic, pp. 39–47, 2009.

[5] C.S. Wallace, “A suggestion for a fast multiplier”, IEEE Transactions on Electronic Computers, Vol. EC-13, pp. 14–17, 1964.

[6] L. Dadda, “Some schemes for parallel multiplier”, Alta Frequenza, Vol. 34, pp. 349–356, 1965.

[7] J. Fadavi-Ardekani, “M*N Booth encoded multiplier generator using optimized Wallace trees”, IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors, pp. 114–117, 1992.

[8] R.P. Brent, H.T. Kung, “A regular layout for parallel adders”, IEEE Transactions on Computers, Vol. C-31, pp. 260–264, 1982.

[9] J. Sklansky, “Conditional-sum addition logic”, IRE Transactions on Electronic Computers, Vol. EC-9, pp. 226–231, 1960.

[10] P.M. Kogge, H.S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations”, IEEE Transactions on Computers, Vol. C-22, pp. 786–793, 1973.