

Low-power and high-speed shift-based multiplier for error tolerant applications



Sami Malek, Sarah Abdallah, Ali Chehab, Imad H. Elhadj, Ayman Kayssi*

Department of Electrical and Computer Engineering, American University of Beirut, Beirut, 1107 2020, Lebanon

ARTICLE INFO

Article history:

Received 3 October 2015

Revised 23 February 2017

Accepted 3 July 2017

Available online 6 July 2017

Keywords:

Multiplier

Error-tolerant applications

Low power

Error model

Shift-and-add

Pass transistor logic

Integrated circuits

ABSTRACT

We propose a new multiplier design that fulfills the need for low-power circuit blocks used in error-tolerant applications on energy-constrained devices. The design trades accuracy for higher speed, lower energy consumption, and lower transistor count. The average relative error of an N -bit multiplier is modeled as a function of N and saturates at a constant (around 17%) as the multiplier width increases. An 8-bit implementation simulated in HSPICE achieved almost 90% energy savings for a random sample of operands as compared to a conventional parallel multiplier. The design is flexible whereby simple variations to the circuit structure lead to a perfectly accurate multiplier. Tests performed on multimedia applications such as JPEG compression showed a promising outcome.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

With the advances in transistor technology and parallel computing, energy consumption has become the new major hurdle, especially for energy-limited devices such as smartphones and other portable devices. Today, the need for energy aware computing is more than ever. Many approaches aim to fulfill this need on the transistor, circuit, architecture, and device levels [1].

In this paper, we target error tolerant applications [2] and focus on designing a stochastic low-power multiplier block, given that the multiplier is extensively used in very important applications such as digital signal processing. Even though stochastic circuit blocks produce inaccurate results, they should not deteriorate the user experience or the main function of the application. We exploit in this paper the error tolerance of these applications to significantly improve the speed, performance, power, chip area, and energy consumption of the multiplier block. The proposed multiplier can for instance be a building block in the stochastic ALU of the general-purpose hybrid processor architecture discussed in [1].

We propose a new multiplication scheme that looks promising in achieving our aims. We compute its maximum error and model its average error in terms of the width of the multiplier. Many vari-

ations of our scheme are provided to show the flexibility of the design. Also, we propose two different designs for our scheme and then simulate them and validate their benefits. Finally, we test the scheme on a JPEG encoding algorithm to demonstrate its feasibility in error-tolerant multimedia applications.

The rest of the paper is organized as follows: Section II discusses related work. Section III discusses our proposed multiplier scheme along with its variations. Section IV presents the design and implementation of our multipliers. Simulation and testing results are presented in section V. We conclude and present potential future work in section VI.

2. Literature review

Mottaghi-Dastjerdi et al. [3] state that the generation of partial products, i.e. the repeated shifts and additions that converge to the accurate result, is the largest contributor to the total power consumed in shift-and-add multipliers. The authors propose a new low-power structure for shift-and-add multipliers that can be used for low-power applications that do not primarily depend on speed. It is different from the conventional multiplier where multiplying A by B includes directly feeding A to an adder, bypassing the adder when possible, using a ring counter instead of a binary counter, removing the partial product shift, and eliminating shifting the B register. Simulation results showed that the proposed architecture lowers the total switching activity to 76% and the power consumption to 30% compared to the conventional case.

* Corresponding author.

E-mail addresses: sam41@aub.edu.lb (S. Malek), saa78@aub.edu.lb (S. Abdallah), chehab@aub.edu.lb (A. Chehab), ie05@aub.edu.lb (I.H. Elhadj), ayman@aub.edu.lb (A. Kayssi).

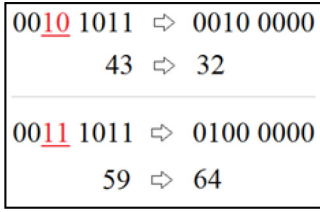


Fig. 1. Rounding process.

Muralidharan and Jagadeeswari [4] propose a new type of adder that achieves high performance while attaining low power consumption for digital signal processing applications. They target the addition operation by eliminating the carry propagation path, which accounts for the major portion of the energy consumption of the adder block in the conventional ripple carry adder in addition to a large contribution to the adder delay.

The authors of [5] propose an innovative multiplication scheme where the operands are split and only their most significant parts are multiplied accurately while the other parts are approximated by simple logic. The resulting errors are considered acceptable for circuit designers and users. This scheme proved to enhance the speed and to reduce the power consumption.

Our proposed scheme goes further by fully removing the adder block and thus promising much higher energy reductions, chip area savings, and speed improvements. Moreover, our design confines the average error to an acceptable value with a uniform distribution.

3. Multiplier Scheme: *Shiftply*

3.1. Proposed scheme

The main idea behind our proposed scheme, *Shiftply*, is to approximate the first operand of a multiplier to the nearest power of 2 and then multiply it by the second operand. This way, we end up with a simple shift instead of a full-fledged multiplication.

Rounding is quite simple in binary. For a binary number, the nearest power of 2 can be found by locating the most significant bit (MS1). For example, in an 8-bit system, the number 5 (0000 0101) has the MS1 at position 6 assuming little endian and starting indexing from 1. Then, we look at the next lower significant bit (in our example it is at position 7 and is a 0). If this bit is 0, we set all bits other than the MS1 to zero and we obtain the rounded number. If this bit is 1, we set all the bits other than the MS1 to zero and we shift the MS1 by 1 position to the left to obtain the rounded number. Hence number 5 gets rounded to 4. However, number 7 is rounded to 8 and not 4. Fig. 1 illustrates the process for the numbers 43 and 59.

One can also think of this scheme as a truncation of the conventional shift-and-add multiplier where we start the process from the MSB of the operand and we do not carry any addition operation [6]. For example, if we want to multiply number 7 by 6, we first round 6 to 8, then we shift 7 three times and get 56.

3.2. Average error model

Let the operand, which will be rounded to the nearest power of 2, be x and the other operand which will be shifted by y . The error of the multiplication of x by y is given by

$$e = \frac{|x \times y - \text{round}(x) \times y|}{x \times y}$$

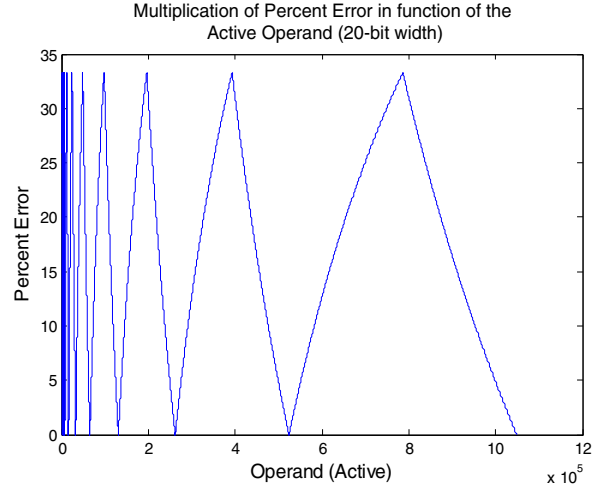


Fig. 2. Percent error e as a function of x for $N=20$.

Here, $\text{round}(x)$ is the function that would round x to the nearest power of 2. For $y > 0$, it follows that:

$$e = \frac{|x - \text{round}(x)|}{x}$$

Hence, the error can be expressed independently of y . The exact formula that rounds x to the nearest power of 2 can be expressed as:

$$\text{round}(x) = 2^{\lfloor \log_2(x) \rfloor + \lfloor 2^{\lfloor \log_2(x) \rfloor - \log_2(3)+1} \rfloor}$$

Fig. 2 shows a plot of the error e as a function of x for $N=20$, i.e. for x ranging from 0 to $(2^{20} - 1)$. We can see a maximum error of around 33%, later verified by actual simulations.

Since the error depends only on x , the total error from rounding x for a given N -bit multiplier is:

$$e_x = (2^N - 1) \frac{|x - \text{round}(x)|}{x}$$

Where $(2^N - 1)$ is the number of possible multiplication operations by x . Each x is rounded and used for all 2^N combinations of y except when $x=0$.

Knowing that x ranges from 0 to $(2^N - 1)$ for an N -bit multiplier and that the error is 0 for all y when $x=0$, the total error for an N -bit multiplier can be expressed as follows:

$$e_{total,N} = (2^N - 1) \sum_{i=1}^{2^N-1} \frac{|i - \text{round}(i)|}{i}$$

Accordingly, the average error for an N -bit multiplier using this scheme is:

$$e_{avg,N} = \frac{(2^N - 1)}{2^{2N}} \sum_{i=1}^{2^N-1} \frac{|i - \text{round}(i)|}{i}$$

For large N , we can approximate the average error by:

$$e_{avg,N} = \frac{1}{2^N} \sum_{i=1}^{2^N-1} \frac{|i - \text{round}(i)|}{i}$$

The plot in Fig. 3 shows that the error model saturates at a 17% average error. Note that simulation results correlate with this model.

3.3. Choice of operand

The following terms are defined:

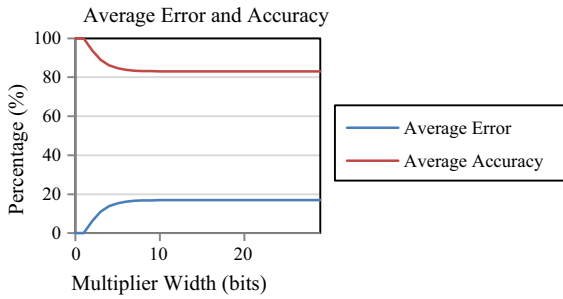


Fig. 3. Average error & accuracy behavior vs. multiplier width.

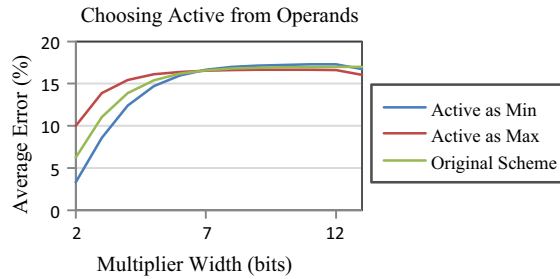


Fig. 4. The choice of Active's effect on average error.

- *Active*: Operand rounded to nearest power of 2
- *Passive*: Operand shifted according to the *Active*'s MS1
- *Shiftplier*: A multiplier which uses the *Shiftply* scheme

Whenever the *Active* is a power of 2, we get 100% accurate results regardless of the *Passive*. A quick comparative study showed that for an 8-bit *Shiftplier*, choosing the larger operand, the smaller operand, or the same operand as the *Active* would give the same average error as shown in the graph of Fig. 4. As a result, the latter scheme was chosen, thus saving the usage of a comparator.

3.4. Scheme variations

3.4.1. Shiftply-Bold

This first variation aims to further reduce the hardware size and energy consumption of the shifter-based implementation of *Shiftply* discussed later in section IV at the cost of increasing the average error. It implements the same logic as in the *Shiftply* scheme: The most significant "10" or "11" pattern of the *Active* is located, which then gets rounded to the nearest power of 2. However, while shifting the *Passive*, we fill it sequentially with the *Active* bits (rather than 0's) starting from the MSB. An example is illustrated in Fig. 5. First, the reverse of the *Active* is concatenated to the LSB side of the *Passive* as shown in the figure. Then, the whole block is shifted towards its LSB while keeping track of the previous LSB value. Shifting stops when either the pair (LSB2, LSB) is "11" or the pair (LSB, Previous LSB) is "01", indicating that the result is ready.

To further illustrate this example, assume the *Passive* and the *Active* are represented by $P_8P_7P_6P_5P_4P_3P_2P_1$ and $A_8A_7A_6A_5A_4A_3A_2A_1$, respectively. The result in the example of Fig. 5 is $A_7A_8P_8P_7P_6P_5P_4P_3P_2P_1A_1A_2A_3A_4A_5A_6$ (*Shiftply-Bold*) instead of $00P_8P_7P_6P_5P_4P_3P_2P_1000000$ (*Shiftply*).

This scheme differs from the *Shiftply* scheme by the *Active* impurities added to the result. In a shifter-based design, this allows a reduction of an N -bit shift register with parallel load. This scheme improves the accuracy in some cases, and severely worsens it in other cases, amounting to a drop in the average accuracy. Moreover, *Active* operands with the most significant pattern "11" result in a significant increase in the error. To mitigate this, we can think of treating the MS "11" case as the case of MS "10". Moreover, in

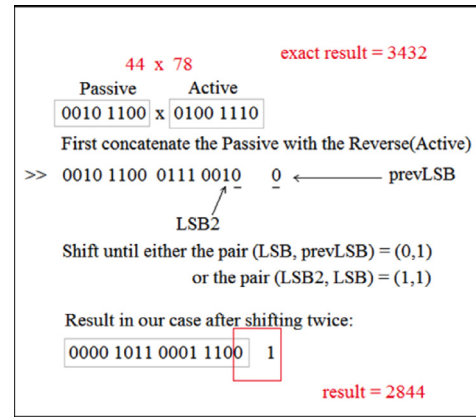


Fig. 5. Shiftply-Bold example.

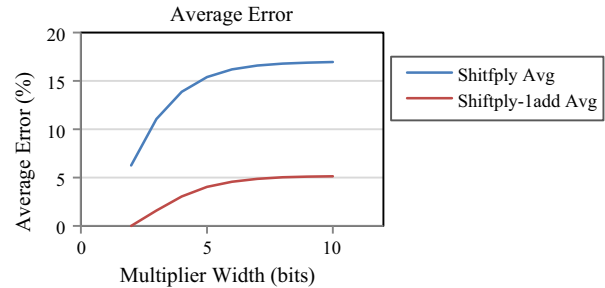


Fig. 6. Comparison of average Error.

this scheme, the way we choose the *Active* has a more significant effect on the result. A larger *Passive* is better. To study these factors, new variations to the scheme were simulated based on:

- **Bold-ns**: This variation doesn't differentiate between a most significant "10" and a "11".
- **Bold-nd**: MS "10" and "11" are treated differently; when *Active* has a MS "11", the *Passive* is shifted once more.
- **Bold-cs**: The larger operand is chosen as the *Passive* which incurs the cost of an additional comparator. This variation also doesn't differentiate between the MS "10" and "11" cases.
- **Bold-cd**: The larger operand is chosen as the *Passive*, which requires an additional comparator. This variation however differentiates between the MS "10" and "11" cases.

3.4.2. Shiftply-1add

In an attempt to boost the accuracy of the original *Shiftply* scheme, the following variation is proposed: The *Active*'s two most significant bits are located, and the *Active* is rounded to two numbers that are consecutive powers of 2. For example: The *Active* binary number 1011 (11 in decimal) is rounded to 1000 (8 in decimal) and 100 (4 decimal) and the result is computed as:

$$Result = 1000 \times Passive + 100 \times Passive$$

Note that the first number is not actually rounded, but rather extracted. After finding the second bit, we use exactly the rounding procedure (10 and 11) used in the original *Shiftply* scheme. This can be thought of as the first addition of the conventional shift-and-add algorithm, but in reverse starting from the MS side.

Obviously, *Shiftply-1add* provides better accuracy than *Shiftply*; however, it increases energy and delay due to additional hardware. Simulation results are shown in Figs. 6–8, which provide a comparison between the maximum error, error distributions, and average error of the *Shiftply-1add* and *Shiftply* schemes, respectively. These results show a remarkable improvement in accuracy for *Shiftply-1add*.

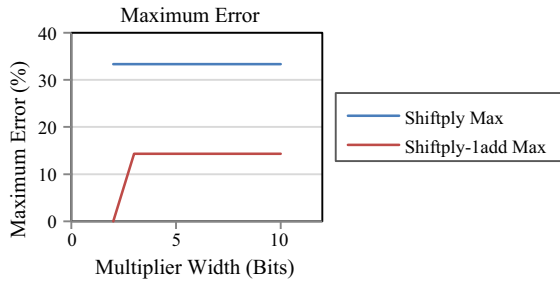


Fig. 7. Comparison of maximum error.

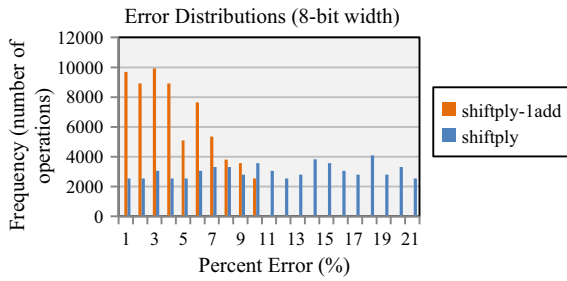


Fig. 8. Comparison of error distributions.

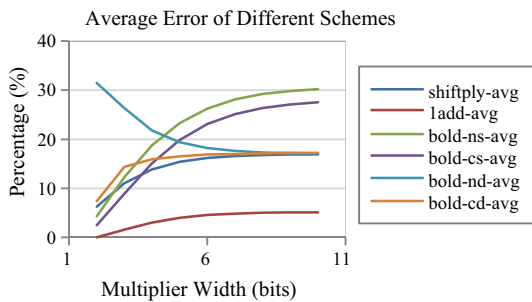


Fig. 9. Average error of various schemes.

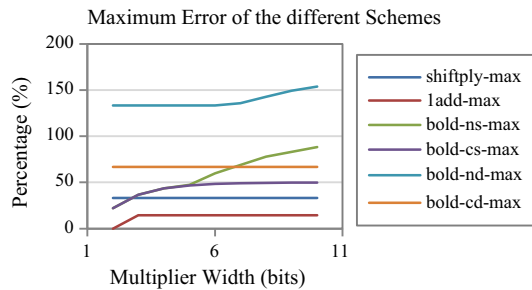


Fig. 10. Maximum error of the various schemes.

This scheme is especially interesting since it can be tuned by increasing the number of shift-and-add operations to balance a desired set of constraints such as energy, error, accuracy, delay and so on.

3.5. Error comparison

The accuracy of the different schemes previously discussed is compared in Figs. 9 and 10, which show the average error and maximum error of the *Shiftply* scheme and its five variations. As observed, the *Shiftply-ladd* scheme provides the best accuracy, having the lowest error.

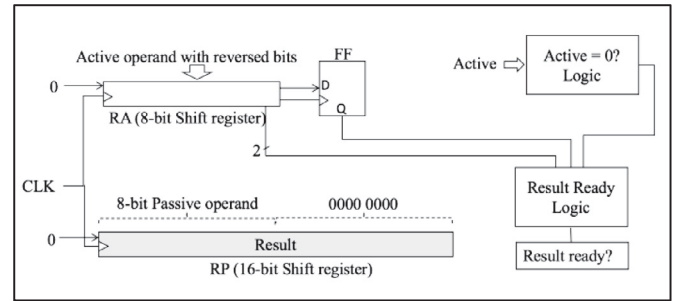


Fig. 11. High-level “Shiftply” design schematic.

Table 1
Average shifts per operation.

Width	Towards MSB	Towards LSB	Optimal
2	0.75	0.75	0.25
3	1.00	1.63	0.63
4	1.19	2.56	0.81
5	1.31	3.53	1.03
6	1.39	4.52	1.14
7	1.44	5.51	1.26
8	1.46	6.50	1.32

4. 8-Bit multiplier design and implementation

Despite the improved results of other schemes in terms of accuracy, the *Shiftply* scheme was adopted since it requires less hardware and less energy consumption. The design and implementation of an 8-bit multiplier adopting the *Shiftply* scheme is presented next.

4.1. 8-Bit sequential (shift-based) design

4.1.1. Shift-Register-based Implementation

One way to implement the *Shiftply* scheme is by using shift registers. A high-level view of the design is shown in Fig. 11. First, the *Active* is parallel-loaded into the top 8-bit shift register (RA) with its bits reversed, and it is simultaneously compared to zero as shown on top right. Register RA is sequentially fed with 0 and its sequential output goes to a flip-flop (FF), which is initialized to 0. The second block is a 16-bit shift register (RP) where initially the *Passive* is loaded into the upper 8 bits with the lower 8 bits set to zero. The sequential input of register RP is 0. If the *Active* turned out to be zero, the upper 8 bits of RP are zeroed and a send signal is set. Otherwise, on each cycle, the *Active* in RA is shifted towards its MSB while the *Passive* is shifted towards the LSB side of the register RP. A simple logic detects when the RP register contains the correct result and should be dispatched.

Simulations confirmed that in the search for the most significant ‘1’, shifting the *Active* towards its MSB direction rather than shifting it towards its LSB direction is more efficient. This is shown in Table 1 where the average number of shifts towards MSB and LSB to find the MS1 are presented.

For an 8-bit *Shiftplier*, an average of 1.465 shifts per operation is needed when the *Active* is being shifted towards its MSB as compared to 6.504 shifts per operation when being shifted towards its LSB. Ideally, some logic will be needed to decide on each case whether it is more efficient to shift the *Active* towards its MSB or LSB. This can be done by comparing the upper half of the *Active* operand to 0. If it is all 0, it is more efficient to shift it towards its LSB; otherwise, towards its MSB. However, since the improvement compared to the MSB shifts is negligible (on average 0.145 less shifts per operation), it is better to always shift towards the

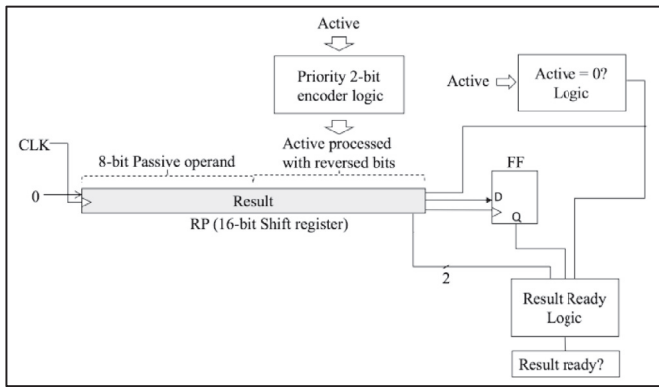


Fig. 12. Design variation.

Table 2

Boolean equations for active equals zero detection logic.

Circuit a	$X = A'B'$	(1)
	$X' = (A'B')' = A+B$	(2)
Circuit b	$W = D'C'X = D'C'A'B'$	(3)
	$W' = A+B+C+D$	(4)
Circuit c	$Z = F'E'$	(5)
	$Z' = (F'E')' = F+E$	(6)
Circuit d	$(Active = 0?) = G'H'ZW = A'B'C'D'E'F'G'H'$	(7)

MSB. In this design, fewer shifts imply fewer clock cycles, delay, and thus energy consumption.

These results stem from the fact that in a system of N bits, there are much more numbers with at least one of the upper half bits set to 1 than numbers with all upper bits being zeroes. For an 8-bit number, the number of combinations with all upper bits being zeroes is $2^4 - 1 = 15$ whereas the number of 8-bit combinations with at least one of its upper half bits set to 1 is 240. As a result, it is more efficient to shift 15 numbers more than 4 times and 496 numbers less than 4 times than doing the opposite.

4.1.2. Possible Design Variations

One can use only one $2N$ -bit register and some additional logic such as a priority 2-bit encoder that would zero out all the less significant bits of the *Active* after the most significant "10" or "11". This design variation is presented in Fig. 12.

4.1.3. Pass-Transistor Logic Implementation

Since we are targeting low-energy circuits, the design was implemented using pass transistor logic. The different circuit blocks of the implementation are presented in what follows.

4.1.3.1. Active equals zero detection logic. Fig. 13 shows the detection logic for "Active equals to zero". Note that A, B, C, D, E, F, G, and H are the 8 bits of the *Active* operand. The detection logic circuit is decomposed into 4 sub-circuits to simplify the drawing. Each sub-circuit is described in a boolean equation form in Table 2. The top sub-circuit (a) is composed of an AND gate (output X) followed by a NOT gate (output X'). Actually, when A is high, $X=0$ regardless of the value of B. When A is low, X' is high, and $X=B'$. Accordingly X is high when both A' and B' are high (Eq. (1)). Additionally, when X is low, the PMOS gate is closed and the output X' becomes high (Vdd). On the other hand, when X is high, the PMOS gate is open and X' is low. Consequently, circuit (a) is equivalent to an OR gate in boolean Logic (see Eq. (2)). In circuit (b), the output D'C' of the first AND gate, is ANDed with X at the second AND gate, resulting in the output W (Eq. (3)). The two AND gates are followed by a NOT gate (output W' in Eq. (4)). Subcircuit (c) is similar to subcircuit (a). It is composed of an AND gate

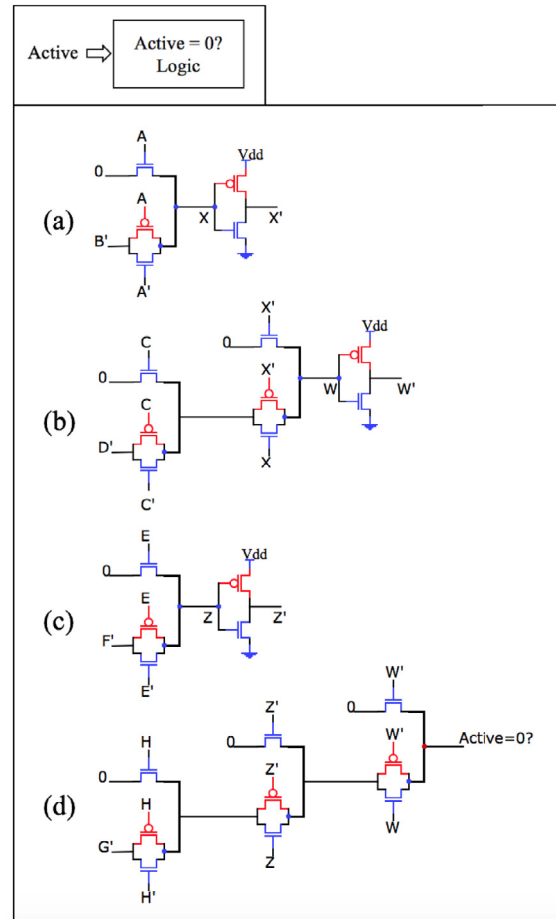


Fig. 13. Active equals '0' detection logic.

(output Z in Eq. (5)) followed by a NOT gate. Thus circuit c is also equivalent to an OR gate (output Z' in Eq. (6)). Finally, circuit (d) is composed of three consecutive AND gates, where the output of each gate is inputted into the following one. G' is ANDed with H', the result is ANDed with Z, and finally ANDed with W. The resulting (Active=0?) output is equivalent to ANDing the complements of the 8-bits of the active operand (Eq. (7)). Consequently, the active bit (the output bit of circuit d) is set when all the complement of the input bits are equal to 1, or equivalently when all the bits of the active operand are 0.

4.1.3.2. Result-ready logic. Fig. 14 shows the result-ready logic. It detects when the RP register contains the result. It is based on the logic used for approximating a number to the nearest power of 2 in section III.A.

In fact, at the start of the multiplication, the RA register contains the active operand in reverse order. The bit 'b' in Fig. 14 is the MSB of the *Active* operand, and bit 'a' is the next bit after the MSB, whereas bit 'c' hold the value of the previous MSB as the active gets shifted right. 'c' is initialized at 0. The passive is inputted into the upper 8-bits of the RP register, with the lower 8 bits set to 0. Consequently, at the start of the multiplication the result register contains the value of the passive multiplied by 2^8 .

On each cycle, bits 'b', 'a' and 'c' are verified, and the RP register is shifted to the right once, to reduce the power of 2 that it is being multiplied with. The solution is ready in two cases: when the MSB is 1 and the bit right after it is 1. In this case the passive is not shifted and result is directly dispatched (*the active is approximated to the higher power of 2 when we encounter a 11 pattern*). Or

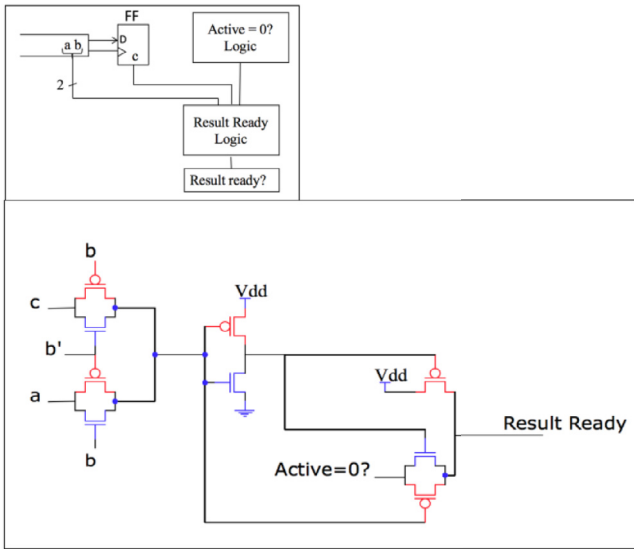


Fig. 14. Result-ready logic.

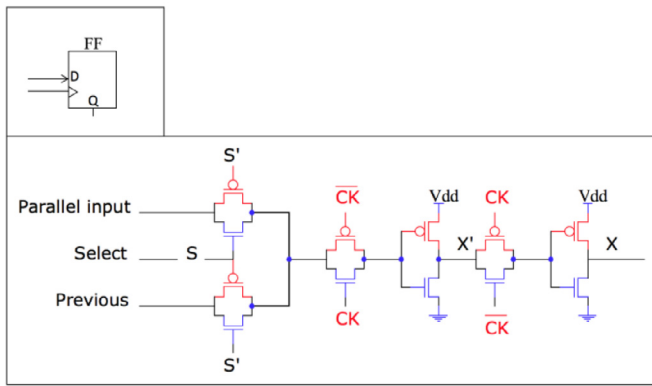


Fig. 15. Flip-flop unit.

when the MSB is 1 followed by a zero. In this case, an extra shift is needed on the passive to reduce the order of 2 that it is being multiplied with by 1 (*the active is approximated to the lower power of 2 when we encounter a 10 pattern*). Consequently, the result is ready in the 16-bit register when $a=1$ and $b=1$ (the leftmost lower transmission gate in Fig. 14 is closed and passing a value of 1), or when $c=1$ and $b=0$ (the leftmost upper transmission gate in Fig. 14 is closed and passing a value of 1). The result ready bit is also set when the $\langle \text{active}=0 \rangle$ logic returns 1 (the rightmost transmission gate always passes the value of $\langle \text{active}=0 \rangle$ to the output). This occurs when the active operand is equal to 0. Eq. (8) describes the logic in Boolean.

$$\text{Result Ready} = (ab + bc') + \langle \text{Active} = 0? \rangle \quad (8)$$

4.1.3.3. Flip-flop unit logic. The flip-flop logic is shown in Fig. 15. An N -bit register is constructed out of N concatenated flip-flop units. The total transistor count of this implementation is around 333. The *Shiftply-Bold* scheme would require around 240 transistors and the *Shiftply-1add* scheme would require additionally a 16-bit adder as well as more glue logic. This implementation was simulated and verified using *LogiSim*.

4.2. 8-Bit combinational design

It is also possible to implement the *Shiftply* scheme using a combinational circuit without the need for a clock. The truth ta-

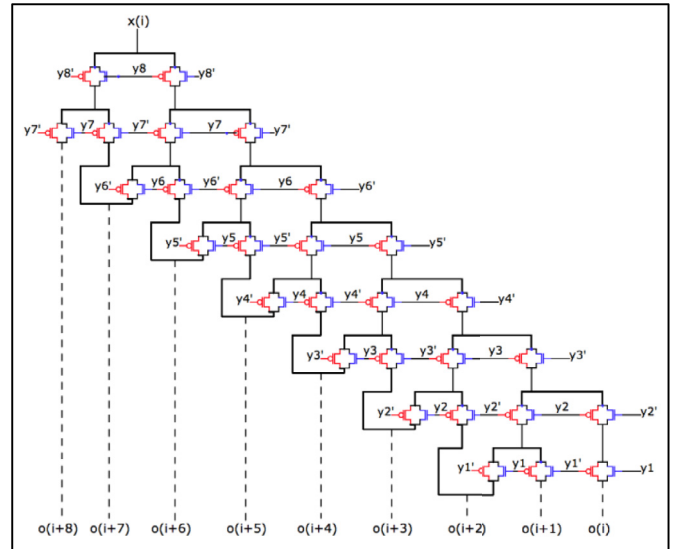


Fig. 16. First block.

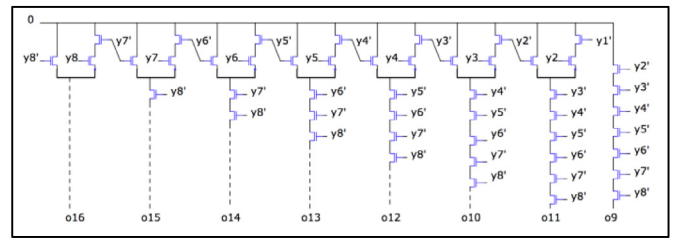


Fig. 17. Second block.

ble for the combinational *Shiftply* is shown in Table 3. The *Passive* is represented by $x[8-1]$, the *Active* by $y[8-1]$, and the result is represented by $o[16-1]$.

8-bit Combinational *Shiftply* Implementation: The design is based on forcing an output on the $o[16-1]$ bits given the sequence of the bits of the active operand bits $y[8-1]$, according to the truth table in Fig. 16. The design was divided into three blocks.

- 1) The first block directs each *Passive* bit $x[i]$ to the appropriate result bit $o[j]$. This block consists of 8 copies of the circuit shown in Fig. 16 where i varies from 8 to 1. The circuit is derived from Table 3. For example, for $i=1$, $x[1]$ is directed to bit $o[i+8]=o[9]$ when $y7=1$ and $y8=1$. In the truth table, we find $x[1]$ in position $o[9]$ when $y8$ and $y7$ are 1 and all other bits are don't cares (first row). Consequently, in the circuit, output $o[i+8]$ is connected to input $x[i]$ through 2 transmission gates in series driven by $y7$ and $y8$. Which means that $x[i+8]$ will only contain the value of $x[i]$ when $y7$ and $y8$ are both equal to 1. Also, output $o[i+2]=o[3]$ contains the value $x[1]$ when transmission gates driven by $y8', y7', y6', y5', y4', y3'$ and $y2'$ are passing, or when transmission gates driven by $y8', y7', y6', y5', y4', y3', y2, y1$ are passing. The first case corresponds to the y vector value of 0000010x (row 12 in truth table), and the second to value 00000011 (row 13 in truth table). The same analysis is used to direct the rest of the x bits to the corresponding output bits.
- 2) The second block (Fig. 17) directs zeros to the upper half of the output, $o[16-9]$. The circuit is also derived from Table 3. For example, $o[16]$ is set to 0 whenever $y8=0$, or $y8=1$ and $y7=0$. This is verified in the truth table. In the circuit, 0 is connected to $o[16]$ only when the transistor driven by $y8'$ is on, or when both transistors driven by $y7'$ and $y8$ are on. The same analysis

Table 3
8-bit Shiftply truth table.

y8	y7	y6	y5	y4	y3	y2	y1	o16	o15	o14	o13	o12	o11	o10	o9	o8	o7	o6	o5	o4	o3	o2	o1
1	1	x	x	x	x	x	x	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0	0	0	0	0
1	0	x	x	x	x	x	x	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0	0	0	0
0	1	1	x	x	x	x	x	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0	0	0	0
0	1	0	x	x	x	x	x	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0	0	0
0	0	1	1	x	x	x	x	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0	0	0
0	0	1	0	x	x	x	x	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0	0
0	0	0	1	1	x	x	x	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0	0
0	0	0	1	0	x	x	x	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0	0
0	0	0	0	1	1	x	x	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0	0
0	0	0	0	1	1	x	x	0	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0
0	0	0	0	1	0	x	x	0	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0
0	0	0	0	0	1	1	0	x	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2	x1
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3	x2
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7	x6	x5	x4	x3
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7	x6	x5	x4
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7	x6	x5
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7	x6
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8	x7
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	x8
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4
Hspice simulation results.

	Power (μ W)	Delay (ns)	Energy (fJ)	Trans. Count
Shift-based "Shiftply"	112.20	1.18	127.24	336
Combinational "Shiftply"	51.26	0.27	51.26	635
Conventional Multiplier	312.06	0.60	468.10	1104

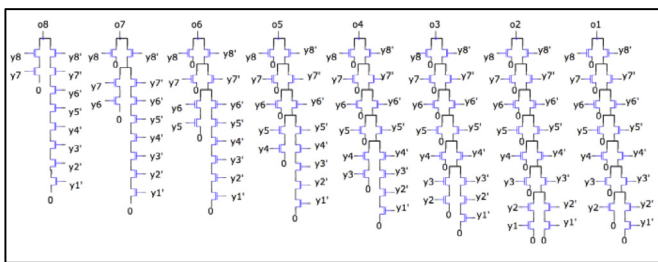


Fig. 18. Third block.

is followed to set bits $o[16-9]$ to 0. Additionally, $o[9]$ is connected to 0 when $y2, y3, y4, y5, y6, y7$ and $y8$ are all 0s. This corresponds to rows 15 and 16 in the truth table. The same analysis is followed to direct 0s to the appropriate output bits.

- The third block (Fig. 18) directs zeros to the lower half of the output, using the same approach as in the second block. For example, $o[8]$ is set to 0 when the transistor driven by $y7$ and $y8$ are on, i.e. $y7$ and $y8$ are both 1 (row 1 in truth table), or when transistors driven by $y8', y7', y6', y5', y4', y3', y2', y1'$ are on, i.e. $y8$ through $y1$ are all 0s (row 16 in truth table).

The total transistor count for this implementation is 635.

5. Simulation and testing results

To study the performance of the proposed designs, we perform HSPICE simulations for a multiplier with $N=8$. All circuits were implemented using the 90nm BSIM4 model provided by Hspice [7], with $L_{min}=0.12\mu m$ and $V_{DD}=1V$. The simulation results are shown in Table 4. We set the clock period of the shift-based implementation to 0.3 ns. Simulations were performed on the same 1000 sets of random inputs.

We can see that the combinational *Shiftplier* outperformed the conventional (exact, accurate) shift-and-add multiplier on all aspects except accuracy.

Table 5 shows the performance of the multipliers relative to the exact conventional parallel multiplier. Fig. 19 illustrates the im-

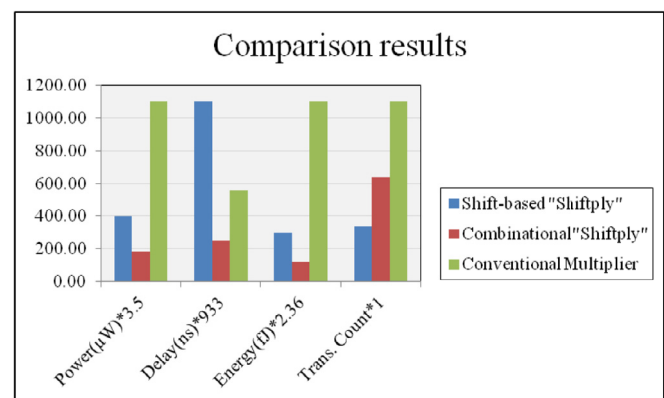


Fig. 19. Advantageous results of the *Shiftply* scheme.

provements of the *Shiftply* scheme with respect to the exact parallel multiplier.

The shift-based sequential implementation takes the least transistor count and can reach very high speeds with high clock frequencies of 3.33 GHz. For an N -bit multiplier, roughly, $3N$ -bit shift registers are needed or $3 \times N \times 16$ transistors in PTL. It has a worst-case operation count of N shifts, which translates into a worst-case delay of N cycles neglecting operand loading.

Compared to the sequential design, the combinational implementation proved to have lower energy consumption for an 8-bit multiplier, much less delay, but a higher transistor count (almost double), which increases greatly as N increases.

Finally, we tested the *Shiftply* scheme on a higher-level JPEG compression algorithm, an error tolerant application whose characteristics were studied in [8]. In this latter, the Discrete Cosine Transformation (DCT) block of the application is identified as the most energy consuming block. Consequently, voltage scaling techniques are performed on the adders to reduce their power consumption. Results showed that energy improvements of around 30% are accomplished while still obtaining satisfactory user experience. In our design, we target the multiplication operations of the DCT blocks instead of the additions, by replacing the conventional

Table 5
Performance relative to the exact conventional parallel multiplier.

	Power Saving (%)	Speedup (%)	Energy Saving (%)	Area Saving (%)
Shift-based "Shiftply"	64.0	–96.9	72.8	69.6
Combinational "Shiftply"	83.6	54.9	89.0	42.5



Fig. 20. JPEG encoding using *Shiftply*.



Fig. 21. JPEG encoding using exact conventional multiplier.

Table 6
Measured error between conventional JPEG image And Shiftply-based JPEG Image.

	Conventional JPEG image	Shiftply-based JPEG image
PSNR	37.84 dB	28.68 dB
MSE	10.77	88.82

multiplication with the *Shiftply* model. Our results are shown in Figs. 20 and 21.

Fig. 22 represents the difference between the two images. It is evident that there is negligible quality loss to the image using the proposed scheme. Table 6 shows the PSNR and the MSE measured between the original image and each of the two compressed images. When compared to the original image, the compressed images exhibit close PSNR, which means that the error introduced with the *Shiftply* scheme did not greatly increase the noise. The MSE measure increased when using the *Shiftply* scheme when compared to the MSE obtained with the conventional JPEG compression. However, this is expected because the errors introduced when using *Shiftply* are amplified by MSE because it squares them. Despite of the increased errors, the quality loss is tolerated by the human eye. This demonstrates the feasibility of the *Shiftply* scheme in error tolerant multimedia applications.

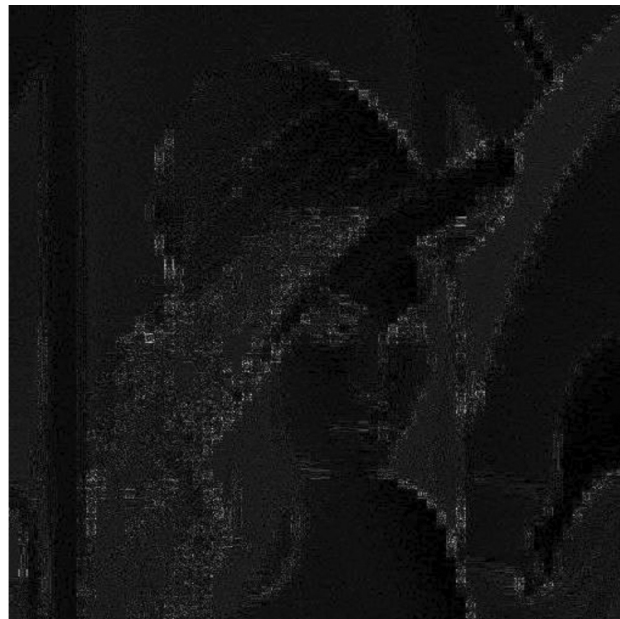


Fig. 22. Difference image – scale 1:1.5.

6. Conclusion and future work

In conclusion, we proposed several stochastic multiplier designs that show considerable energy and power reductions with respect to the exact parallel multiplier, and demonstrated feasibility in an error-tolerant application - JPEG encoding. The average delay per operation, although aggravated by the sequential implementation, is directly dependent on clock frequency. Moreover, the sequential design contributed to immense chip area savings and is interesting to study its forecasted performance enhancement in serial-in/serial-out sequential multipliers. On the other hand, the combinational design implementation outperformed the conventional multiplier on all aspects and mainly with respect to energy consumption with an 89% average energy savings per operation. Finally, results from the error model of the proposed scheme can be used to further study its applicability in other error-tolerant applications. It is also interesting to study the increased energy and chip area savings with increased multiplier width, especially that, in our scheme, the average and maximum error are constants for large widths.

Acknowledgment

This research was funded by Intel project on Middle-East Energy Research (MER).

References

- [1] S. Abdallah, C. Ali, T.H. Elhaji, A. Kayssi, Stochastic hardware architectures: a survey, in: International Conference on Energy Aware Computing, 2012, pp. 91–96.

- [2] D.D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, F. Chong, Characterization of error-tolerant applications when protecting control data, in: Proc. IEEE International Symposium on Workload Characterization, San Jose, California, USA, 2006.
- [3] M. Mottaghi-Dastjerdi, A. Afzali-Kusha, M. Pedram, BZ-FAD: a low-power low-area multiplier based on shift-and-add architecture, *IEEE Trans. Very Large Scale Integr. Circuits* 17 (2) (2009) 302–306.
- [4] V. Muralidharan, Dr.M. Jagadeeswari, An enhanced carry elimination adder for low power VLSI applications, *Int. J. Eng. Res. Appl. (IJERA)* 2 (2) (2012) 1477–1482 issue ISSN: 2248-9622 www.ijera.com Vol. 2, Issue 2, Mar-Apr 2012.
- [5] KhaingYin Kyaw, WangLing Goh, KiatSeng Yeo, Low-power high-speed multiplier for error-tolerant application, in: IEEE International Conference on Electron Devices and Solid-State Circuits, Hong Kong, 2010, pp. 15–17.
- [6] C.N. Marimuthu, P. Thangaraj, Aswathy Ramesan, Low power shift and add multiplier design, *Int. J. Comput. Sci. Inf. Technol.* 2 (2010) Number 3, June.
- [7] Synopsys, "HSPICE reference manual: MOSFET models", version D-2010.12, Dec. 2010.
- [8] F. Saab, C. Ali, I.H. Elhajj, A. Kayssi, Energy efficient JPEG using stochastic processing, in: International Conference on Energy Aware Computing, 2012, pp. 97–102.