1 Basic Arithmetic Circuits

Oscar Gustafsson and Lars Wanhammar

Division of Computer Engineering, Linköping University, Linköping, Sweden

1.1 Introduction

General-purpose DSP processors, application-specific processors [1], and algorithm-specific processors are used to implement different types of DSP systems or subsystems. General-purpose DSP processors are programmable and therefore, provide maximum flexibility and reusability. They are typically used in applications involving complex and irregular algorithms while application-specific processors provide lower unit cost and higher performance for a specific application, particularly when the volume of production is high. The highest performance and lowest unit cost is obtained by using algorithm-specific processors. The drawback is the restricted or even lack of flexibility, and very often the nonrecurring engineering (NRE) cost could be very high.

The throughput requirement in most real-time DSP applications is generally fixed, and there is no advantage of an implementation with throughput than that design to minimize the chip area, and power consumption. Now in a CMOS implementation with higher throughput than required, it is possible to reduce the power consumption by lowering the supply voltage and operating the system at lower frequency [2].

1.2 Addition and Subtraction

The operation of adding two or more numbers is the most fundamental arithmetic operation, since most other operations in one or another way are based

Arithmetic Circuits for DSP Applications, First Edition. Edited by Pramod Kumar Meher and Thanos Stouraitis.
© 2017 by The Institute of Electrical and Electronics Engineers, Inc. Published 2017 by John Wiley & Sons, Inc.

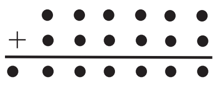


Figure 1.1 Addition of two binary numbers.

on addition. The operands of concern here are either two's-complement or unsigned representation.

Most DSP applications use fractional arithmetic instead of integer arithmetic [3]. The sum of two w-bit numbers is a $(w + 1)$-bit number while the product of two w-bit binary numbers is a $2w$-bit number. In many cases and always in recursive algorithms the resulting number needs to be quantized to a w-bit number. Hence, the question is which bits of the result are to be retained. In fractional arithmetic, the input operands as well as the result are interpreted as being in the range [0, 1], that is,

$$x = \sum_{i=1}^{w} x_i 2^{-i} \tag{1.1}$$

for unsigned numbers and in the range [−1, 1], that is,

$$x = -x_0 \sum_{i=1}^{w} x_i 2^{-i} \tag{1.2}$$

for signed numbers in two's-complement representation. For convenience, we let w denote the number of fractional bits and one additional bit is used for a signed representation.

We use the graphic representation shown in Figure 1.1 to represent the operands and the sum bits with the most significant bit to the left.

1.2.1 Ripple-Carry Addition

Ripple-carry addition is illustrated in Figure 1.2. A ripple-carry adder performs addition of two numbers; adds the bits of the same significance and the carry-bit from the previous stage sequentially using a full adder (FA), and propagates the carry-bit to the next stage. Obviously, the addition takes w addition cycles, where duration of each clock cycle is the time required by an FA to complete the addition of three bits. This type of adder can add both unsigned and two's-complement numbers.

The major drawback with the ripple-carry adder is that the worst-case delay is proportional to the word length. Also, typically, the ripple-carry adder produces many glitches since the full adder cells need to wait for the correct carry input. This situation is improved if the delay for the carry bit is smaller than that of the sum bit [4].

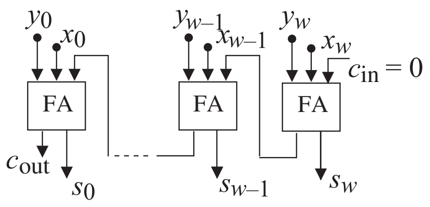


Figure 1.2 Ripple-carry adder.

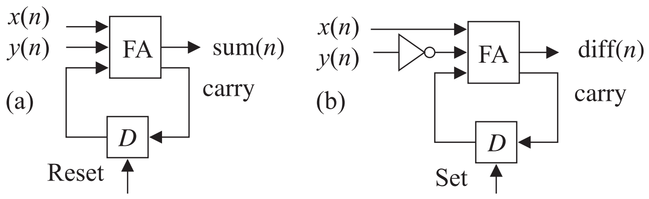


Figure 1.3 Bit-serial (a) adder and (b) subtractor.

Alternatively, all pairs of bits of the same significance can be added simultaneously and then the carries are added using some efficient scheme. Many additional schemes have been proposed. For more details, we refer to, for example, Reference [5].

It is also possible to perform addition in constant time using redundant number systems such as signed-digit or carry-save representations [6]. An alternative is to use residue number systems (RNS), which split the carry-chain into several shorter ones [7].

1.2.2 Bit-Serial Addition and Subtraction

Bit-serial addition and subtraction of numbers in a two's-complement representation can be performed with the circuits shown in Figure 1.3. A pair of input bits of operands X and Y are fed to the circuit in the least-significant-bit-first order, and the carry generated during at any given cycle is returned as input of the circuit during the next clock cycle. Since the carries are saved from one bit position to the next, the circuits of Figures. 1.3a and b are called carry-save adder and carry-save subtractor, respectively. At the start of the addition, the D flip-flop is reset (set) for the adder (subtractor), respectively.

Figure 1.4 shows two latency models for bit-serial adders. The leftmost is suitable for static CMOS and the rightmost for dynamic (clocked) logic styles. The time to add two w-bit numbers is w or $w + 1$ clock cycles. However, the throughput rate of computation would not be affected by this additional latency of one clock cycle since the two successive sum values computed in an interval is w cycles.

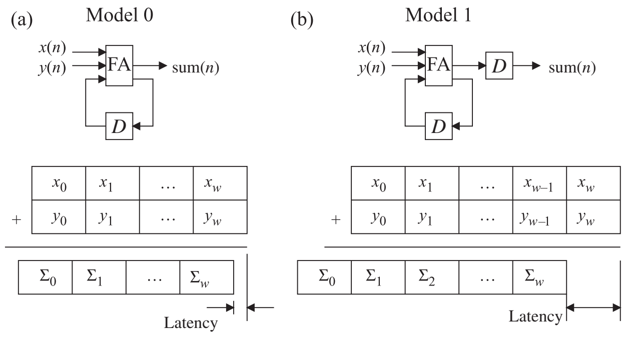

(a)
Model 0
x(n)
y(n)
FA
sum(n)
D
x_0 x_1 … x_w
+ y_0 y_1 … y_w
Σ_0 Σ_1 … Σ_w
Latency
(b)
Model 1
x(n)
y(n)
FA
D
sum(n)
D
x_0 x_1 … x_{w-1} x_w
+ y_0 y_1 … y_{w-1} y_w
Σ_0 Σ_1 Σ_2 … Σ_w
Latency

Figure 1.4 Latency models for a bit-serial adder.

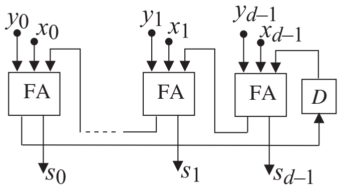


Figure 1.5 Digit-serial adder with digit-size *d* obtained from unfolding a bit-serial adder.

1.2.3 Digit-Serial Addition and Subtraction

In case of digit-serial processing [8–11] a group of bits (called a digit) of input words are processed at a time. From speed and power consumption points of view, it is advantageous to process several bits at a time. The number of bits processed in a clock cycle is referred to as the digit size.

Figure 1.5 shows a digital-serial adder, where d is the digit size. The D-flip-flop transfers the output carry. In case of subtraction of a two's-complement number, the negative value is instead added by inverting the bits and setting the carry flip-flop during the addition of the least significant digit.

Most of the principles used for bit-serial arithmetic can easily be extended to digit-serial arithmetic.

The bit-serial and digit-serial arithmetic circuits require less chip area and therefore their equivalent switched capacitance and leakage current are relatively low compared with word-level circuits [3,12].

1.3 Multiplication

Multiplication of two numbers can be realized into the following two main steps:

1. Partial product generation where partial product is the result of multiplication of a bit of the multiplier with the multiplicand.
2. Accumulation of the partial products.

In the following subsections, we discuss various techniques to simplify and speed-up the summation of the partial products.

1.3.1 Partial Product Generation

In unsigned binary number representation, bit-wise multiplications can be written as

$$Z = XY = \sum_{i=1}^{w} x_i 2^{-i} \sum_{j=1}^{w} y_j 2^{-j} = \sum_{i=1}^{w} \sum_{j=1}^{w} x_i y_j 2^{-i-j} \tag{1.3}$$

This leads to a partial product array as shown in Figure 1.6. The partial product generation can be readily realized using AND gates.

For two's-complement representation, the result is very similar to that in Figure 1.6, except that the sign-bit causes some of the bits to have negative weight. This can be seen from

$$\begin{aligned} Z &= XY \\ &= \left(-x_0 + \sum_{i=1}^{w} x_i 2^{-i}\right)\left(-y_0 + \sum_{j=1}^{w} y_j 2^{-j}\right) \\ &= x_0 y_0 - x_0 \sum_{j=1}^{w} y_j 2^{-j} - y_0 \sum_{i=1}^{w} x_i 2^{-i} + \sum_{i=1}^{w} \sum_{j=1}^{w} x_i y_j 2^{-i-j} \end{aligned} \tag{1.4}$$

The corresponding partial product matrix is shown in Figure 1.7.

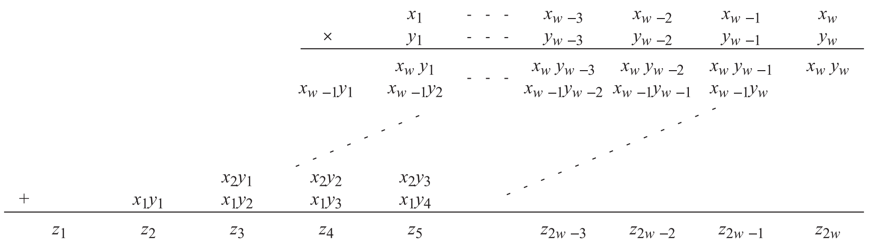
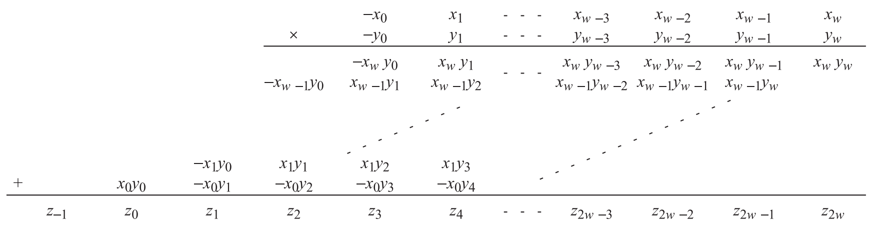

x_1 - - - x_{w-3} x_{w-2} x_{w-1} x_w
$\times$ y_1 - - - y_{w-3} y_{w-2} y_{w-1} y_w
$x_w y_1$ - - - $x_w y_{w-3}$ $x_w y_{w-2}$ $x_w y_{w-1}$ $x_w y_w$
$x_{w-1} y_1$ $x_{w-1} y_2$ $x_{w-1} y_{w-2}$ $x_{w-1} y_{w-1}$ $x_{w-1} y_w$
$x_2 y_1$ $x_2 y_2$ $x_2 y_3$
$+$ $x_1 y_1$ $x_1 y_2$ $x_1 y_3$ $x_1 y_4$
z_1 z_2 z_3 z_4 z_5 z_{2w-3} z_{2w-2} z_{2w-1} z_{2w}

Figure 1.6 Partial products for unsigned binary numbers.



−x0 x1 - - - xw−3 xw−2 xw−1 xw
× −y0 y1 - - - yw−3 yw−2 yw−1 yw
−xw y0 xw y1 - - - xw yw−3 xw yw−2 xw yw−1 xw yw
−xw−1y0 xw−1y1 xw−1y2 xw−1yw−2 xw−1yw−1 xw−1yw
−x1y0 x1y1 x1y2 x1y3
+ x0y0 −x0y1 −x0y2 −x0y3 −x0y4
z−1 z0 z1 z2 z3 z4 - - - z2w−3 z2w−2 z2w−1 z2w

Figure 1.7 Partial products for two's-complement numbers.

1.3.2 Avoiding Sign-Extension (the Baugh and Wooley Method)

Addition of any two numbers in two's-complement representation requires that the word lengths are equal. Hence, it is necessary to sign-extend the MSB of partial products in Figure 1.7 to obtain the same word length for all rows.

To avoid this the computation resulting from sign-extension, it is possible to perform the summation from top to bottom and perform sign-extension of the partial results to match the next row to be added. However, if we want to add the partial products in an arbitrary order using a multioperand adder, the following technique, proposed by Baugh and Wooley [13], can be used.

Note that for a negative partial product, we have $-p = \overline{p} - 1$. Hence, we can replace all partial products with negative weight with an inverted version. Then, we need to subtract a constant value from the result. Since there will be several constants, one from each negative partial product, we can sum these up and form a single compensation vector to be added. When this is applied we get the partial product array as shown in Figure 1.8 as

$$\begin{aligned}
Z &= x_0y_0 + \sum_{j=1}^{w} \overline{x_0y_j}2^{-j} - \sum_{j=1}^{w} 2^{-j} + \sum_{i=1}^{w} \overline{y_0x_i}2^{-i} - \sum_{i=1}^{w} 2^{-i} \\
&\quad + \sum_{i=1}^{w}\sum_{j=1}^{w} x_iy_j2^{-i-j} \\
&= x_0y_0 + \sum_{j=1}^{w} \overline{x_0y_j}2^{-j} + \sum_{i=1}^{w} \overline{y_0x_i}2^{-i} + \sum_{i=1}^{w}\sum_{j=1}^{w} x_iy_j2^{-i-j} \\
&\quad - \left(2 - 2^{-w+1}\right) \qquad (1.5)
\end{aligned}$$

1.3.3 Reducing the Number of Partial Products

There are several methods for reducing the number of partial products. A technique to reduce the number of partial products using small ROM-based

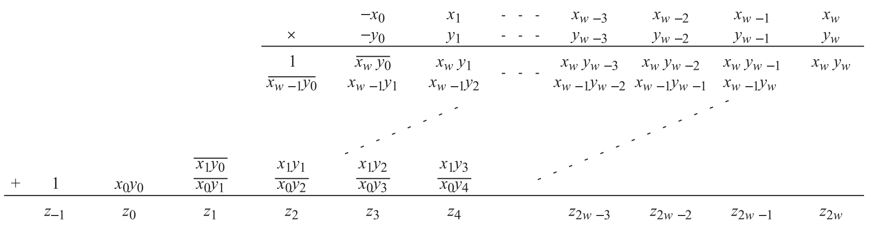


Figure 1.8 Partial products for two's-complement numbers without sign-extension.

Table 1.1 Rules for the radix-4 modified Booth encoding.

$x_{2k}x_{2k+1}x_{2k+2}$	r_k	$d_{2k}d_{2k+1}$
000	0	00
001	1	01
010	1	01
011	2	10
100	−2	$\bar{1}0$
101	−1	$0\bar{1}$
110	−1	$0\bar{1}$
111	0	00

multipliers is found in Reference [14]. Other methods are based on number representation or encoding of one of the operands. It is possible to reduce the number of nonzero positions by using a signed-digit representation, for example, canonical signed digit (a CSD) representation to obtain a minimum number of nonzeros. However, the drawback is that the conversion from two's-complement to CSD involves carry-propagation [15]. Furthermore, the worst case is that half of the positions are nonzero, and, hence, one would still need to design the multiplier to deal with this case.

Instead, it is possible to derive a signed-digit representation that is not necessarily minimal, but has at most half of the positions being non-zero. This is referred to modified Booth encoding [16,17] and is often described as being a radix-4 signed-digit representation where the recoded digits $r_k \in \{-2, -1, 0, 1, 2\}$.

The logic rules for performing the modified Booth encoding are based on the idea of finding strings of ones and replace them as $011...11 = 100...0\bar{1}$, illustrated in Table 1.1. From this, one can see that there is at most one nonzero digit in each pair of digits $\left(d_{2k}d_{2k+1}\right)$.

Now, to perform the multiplication, we must be able to possibly negate and multiply the operand with 0, 1, or 2. As discussed earlier, the negation is typically

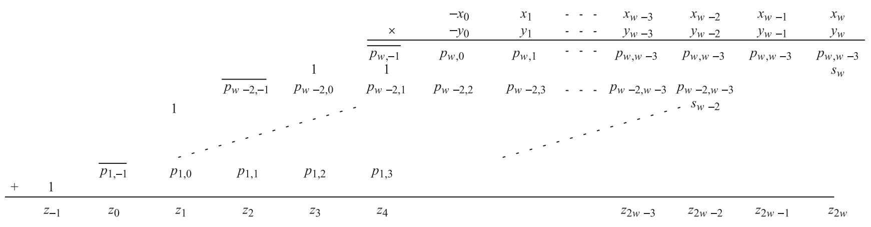

$-x_0$ x_1 x_{w-3} x_{w-2} x_{w-1} x_w
× $-y_0$ y_1 y_{w-3} y_{w-2} y_{w-1} y_w
$\overline{p_{w,-1}}$ $p_{w,0}$ $p_{w,1}$ $p_{w,w-3}$ $p_{w,w-3}$ $p_{w,w-3}$ $p_{w,w-3}$
s_w
1 1
$\overline{p_{w-2,-1}}$ $p_{w-2,0}$ $p_{w-2,1}$ $p_{w-2,2}$ $p_{w-2,3}$ $p_{w-2,w-3}$ $p_{w-2,w-3}$
s_{w-2}
1
$\overline{p_{1,-1}}$ $p_{1,0}$ $p_{1,1}$ $p_{1,2}$ $p_{1,3}$
+ 1
z_{-1} z_0 z_1 z_2 z_3 z_4 z_{2w-3} z_{2w-2} z_{2w-1} z_{2w}

Figure 1.9 Partial products for radix-4-modified Booth encoding.

performed by inverting the bits and add a one in the column corresponding to the LSB position. The partial product array for a multiplier using the modified Booth encoding is shown in Figure 1.9. From this, it can be seen that each row now is one bit longer and that the least significant position contains two bits, where the additional bit is used for negation.

1.3.4 Reducing the Number of Columns

The result of multiplication is usually quantized to be represented with fewer bits. To reduce the complexity of the multiplication of such cases it has been proposed to perform the quantization at the partial product generation stage and partial product summation stage [18]. This is commonly referred to as fixed-width multiplication referring to the fact that (most of) the partial product rows have the same width. Simply truncating the partial products will result in a rather large error. Several methods have therefore been proposed to reduce the error by introducing compensating terms [19,20].

1.3.5 Accumulation Structures

The problem of summing up the partial products can be solved in the following three general ways:

1. Sequential accumulation, where a row or a column of partial products are accumulated in each cycle.
2. Array accumulation, which gives a regular structure.
3. Tree accumulation, which gives the smallest logic depth but in general an irregular structure.

1.3.5.1 Add-and-Shift Accumulation

In multipliers based on add-and-shift, the partial products are successively accumulated in w cycles for w-bit operands. During each cycle a new partial product is added with the current sum of partial products divided by two. In most number systems dividing by two can be implemented by a shift of the bits.

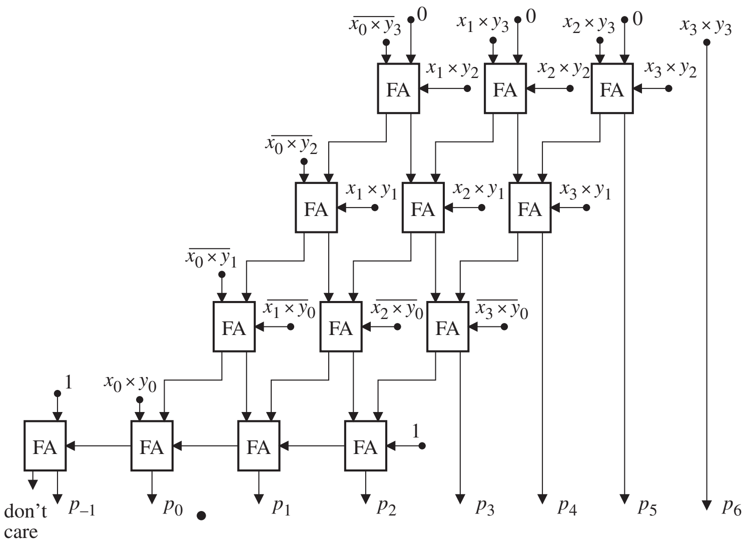


Figure 1.10 Baugh–Wooley array multiplier.

The multiplication time becomes large if two's-complement representation and ripple-carry adders are used. Redundant number representation is therefore used to alleviate the carry propagation problem. We will defer the discussion of add-and-shift multipliers based on bit-serial and digit-serial realizations to Section 1.3.6.

1.3.5.2 Array Accumulation

Array multipliers use an array of almost identical cells for generation and accumulation of the partial products. Figure 1.10 shows a realization of the Baugh–Wooley array multiplier [13], leading to a multiplication time proportional to $2w$.

The array multiplier is a highly regular structure resulting in short wire lengths between the logic circuits, which is important for high-speed design in nanometer processes where wiring delay gives a significant contribution to the overall delay. However, in a process where cell delay dominates wire delay, the logic depth of the design is more important than regularity. In the array multiplier the logic depth is $\mathcal{O}(w)$, where w is the input word length. In the adder tree multiplier, which is discussed in Section 1.3.5.3, the depth is $\mathcal{O}(\log(w))$. Even for short word lengths, this leads to a significant shorter delay.

1.3.5.3 Tree Accumulation

The array structure provides a regular structure, but at the same time the delay grows linearly with the word length. All the partial product generation methods in Figures 1.6–1.9 provide a number of partial products that can be accumulated in arbitrary order.

The first approach is to use as many full adders as possible to reduce as many partial products as possible. Then, we add as many half adders as possible to minimize the number of levels and try to shorten the word length for the vector-merging adder. This kind of approach is followed in the Wallace tree proposed in Reference [21]. The main drawback of this approach is the excessive use of half adders. Dadda [22] instead proposed that full adders and half adders should only be used if required to obtain a number of partial products equal to a value in the Dadda series. The value of position *n* in the Dadda series is the maximum number of partial products that can be reduced using *n* levels of full adders. The Dadda series starts $\{3, 4, 6, 9, 13, 19, \dots\}$. The benefit of this is that the number of half adders is significantly reduced while still obtaining a minimum number of levels.

However, the length of the vector-merging adder increases in case of Dadda reduction trees. A compromise between these two approaches is the 'reduced area' heuristic [23], which is similar to the Wallace tree, since as many full adders as possible are introduced in each level. Half adders are on the other hand only introduced if they are required to reach a number of partial products corresponding to the Dadda series, or if exactly two partial products have the same least significant weight. In this way a minimum number of stages is obtained, while at the same time both the length of the vector-merging adder and the number of half adders are kept small.

To illustrate the operation of the reduction tree approaches we use dot diagrams, where each dot corresponds to a bit of the partial product to be added. Bits with the same weight are placed in the same column and bits in adjacent columns are either of one position higher or one position lower weight, with higher weights toward left. The bits are manipulated by using either full or half adders. The operation of these are illustrated in Figure 1.11.

The reduction schemes are illustrated for an unsigned 6×6-bit multiplication in Figure 1.12. The complexity results are summarized in Table 1.2. It should be noted that the positioning of the results in the next level is resulting partial products done based on ease of illustration. From a functional point of view, this step is arbitrary, but it is possible to optimize the timing by carefully utilizing different delays of the sum and carry outputs of the adder cells [24]. Furthermore, it is possible to reduce the power consumption by optimizing the interconnect ordering [25].

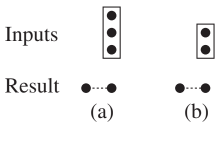


Figure 1.11 Operation on bits in a dot diagram with (a) full adder and (b) half adder.

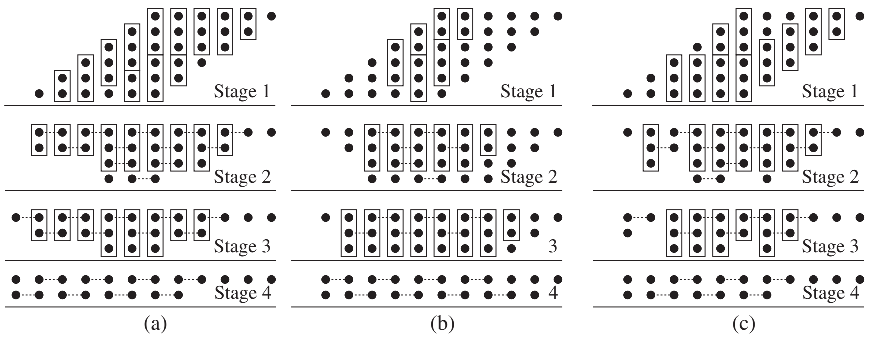


Figure 1.12 Reduction trees for a 6 × 6-bit multiplier: (a) Wallace, (b) Dadda, and (c) Reduced area.

Table 1.2 Complexity of the three reduction trees in Figure 1.12.

Tree structure	Full adders	Half adders	VMA length
Wallace	16	13	8
Dadda	15	5	10
Reduced area	18	5	7

The reduction trees in Figure 1.12 do not provide any regularity. This means that the routing is complicated and may become the limiting factor in an implementation. Reduction structures that provide a more regular routing, but still a small number of stages, include the overturned-stairs reduction tree [26] and the HPM tree [27].

1.3.5.4 Vector-Merging Adder

The role of the vector-merging adder is to add the outputs of the reduction tree. In general, any carry-propagation adder can be used. However, the signals corresponding to different bits of input to the vector-merging adders are typically available after different delays after the arrival of input to the multiplier. It is possible to derive carry-propagation adders that utilize this different signal arrival times to optimize the adder delay [28].

1.3.6 Serial/Parallel Multiplication

Serial/parallel multipliers are based on the add-and-shift principle where the multiplicand, x, arrive serially while the multiplier, a is available in bit-parallel format. Several forms of serial/parallel multipliers have been proposed [29].

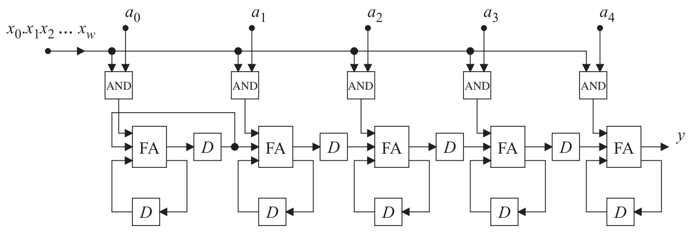


Figure 1.13 Serial/parallel multiplier based on carry-save adders. The box containing "AND" refers to AND gate throughout this chapter.

They differ mainly by the order in which bit-products are generated and added and in the way subtraction is handled.

The generation of partial products are handled by the AND gates shown in Figure 1.13. The first partial products correspond to the first row of partial products in Figure 1.6. Thus, in the first time slot, we add the partial products, $a \times x_w$ to the initially cleared accumulator. Next, the D flip-flops are clocked and the sum-bits from the FAs are shifted by one bit to the right; each carry-bit is saved to be used in the next clock cycle: the sign-bit is copied; and the LSB of the product is produced as output bit. These operations correspond to dividing the accumulator contents by 2. Note that the value in the accumulator is in redundant form and that carry propagation is not needed.

In the following clock cycle, the next bit of x is used to form the next partial product and added to the value in the accumulator, and the value in the accumulator is again divided by 2. This process continues for w clock cycles, until the sign bit x_0 is reached, whereupon a subtraction must be done instead of an addition.

During the first w clock cycles, the least significant part of the product is computed and the most significant is stored in the D flip-flops. In the next $w_c - 1$ clock cycles, we apply zeros to the input so that the most significant part of the product is shifted out of the multiplier. Hence, the multiplication requires $w + w_c - 1$ clock cycles. Two successive multiplications must therefore be separated by $w + w_c - 1$ clock cycles. Note that the accumulation of the partial products is performed using a redundant representation, which do not require carry propagation. The redundant value stored in the accumulator in carry save format is converted to a nonredundant representation in the last FA.

An alternative and better solution is to copy the sign-bit in the first multiplier stage as shown in Figure 1.14. The first stage, corresponding to the sign-bit in the coefficient, is replaced by a subtractor. In fact, only an array of half-adders is needed since one of the input bits to the 1-bit adders is zero.

The subtraction of the bit-products required for the sign-bit in the serial/parallel multiplier can be avoided by extending the input by $w_c - 1$ copies of the sign-bit. After w clock cycles the most significant part of the product is

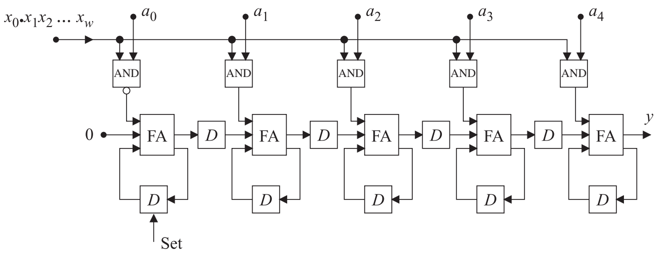
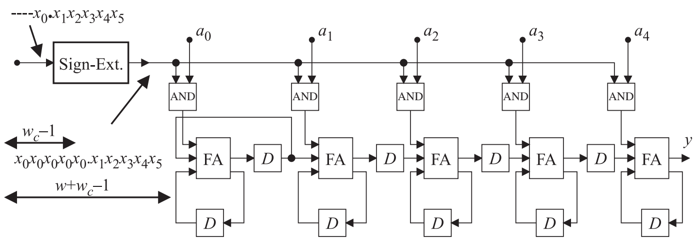


Figure 1.14 Modified serial/parallel multiplier.



----$x_0 . x_1 x_2 x_3 x_4 x_5$
Sign-Ext.
w_c-1
$x_0 x_0 x_0 x_0 x_0 . x_1 x_2 x_3 x_4 x_5$
$w+w_c-1$
a_0
a_1
a_2
a_3
a_4
AND
FA
D
y

Figure 1.15 Serial/parallel multiplier with sign extension-circuit.

stored in the D flip-flops. In the next w_c clock cycles the sign bit of x is applied to the multiplier's input. This is accomplished by the sign extension-circuit shown in Figure 1.15. The sign extension-circuit consists of a latch that transmits all bits up to the sign-bit and thereafter latches the sign-bit. For simplicity, we assume that $w = 5$ bits and $w_c = 5$ bits.

The product is

$$y = a \times x = a \times \left(-x_0 + \sum_{k=1}^{5} x_k 2^{-k} \right) \tag{1.6}$$

but the multiplier computes

$$\begin{aligned} y &= a \left(x_0 2^4 + x_0 2^3 + x_0 2^2 + x_0 2^1 + x_0 2^0 + \sum_{k=1}^{5} x_k 2^{-k} \right) \\ &= a \left(x_0 2^4 + x_0 2^3 + x_0 2^2 + x_0 2^1 + x_0 2^1 - x_0 2^0 + \sum_{k=1}^{5} x_k 2^{-k} \right) \\ &= a x_0 \left(2^4 + 2^3 + 2^2 + 2^1 + 2^1 \right) + a \times x \\ &= a x_0 2^5 + a \times x \end{aligned} \tag{1.7}$$

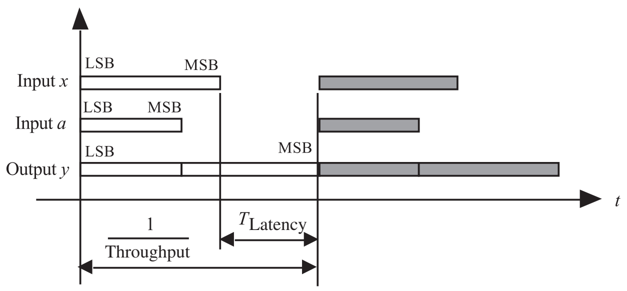


Figure 1.16 Latency and throughput for a multiplication $y = a \times x$ using bit-serial arithmetic with least significant bit (LSB) first.

The first term above contributes an error in the desired product. However, there will not be any error in the $w + w_c - 1$ least-significant bits since the error term only contributes to the bit positions with higher significance.

A bit-serial multiplication takes at least $w + w_c - 1$ clock cycles. However, two successive multiplications can partially be overlapped to increase the throughput [3]. These serial/parallel multipliers using this technique can be designed to perform one multiplication every $\max\{w, w_c\}$ clock cycles. Latency and throughput for a multiplication using bit-serial arithmetic with least significant bit (LSB) first is illustrated in Figure 1.16.

A major advantage of bit-serial over bit-parallel arithmetic is that it significantly reduces chip area. This is done in two ways. First, it eliminates the need of wide buses and simplifies the wire routing. Second, by using small processing elements, the chip itself becomes smaller and requires shorter wiring. A small chip can support higher clock frequencies and is therefore faster.

Bit-serial multiplication can be done either by processing the least significant or the most significant bit first. The former is the most common since the latter is more complicated and requires the use of so-called redundant arithmetic.

The latency of a multiplication is equal to the number of fractional bits in the coefficient. For example, a multiplication with a coefficient $w_c = (1.0011)_{2C}$ will have a latency corresponding to four clock cycles. A bit-serial addition or subtraction has in principle zero latency while a multiplication by an integer may have zero or negative latency. But, the latency in a recursive loop is always positive, since the operations must be performed by causal PEs. In practice, the latency may be somewhat longer, depending on the type of logic that is used to realize the arithmetic operations, as will be discussed shortly.

Here, we define two latency models for bit-serial arithmetic. Two latency models for a bit-serial adder are shown in Figure 1.4. In model 0, which can be used to model a static CMOS logic style without pipelining of the gates, the latency is equal to the gate delay at a full adder. In model 1, which can be used to model a dynamic CMOS logic style, or a static CMOS logic style

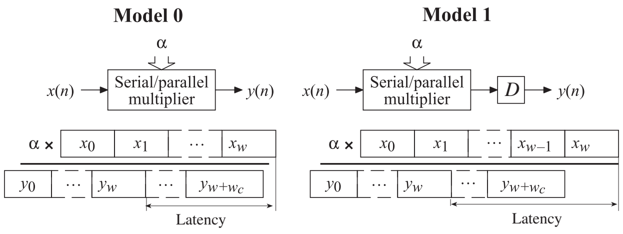


Figure 1.17 Latency models for a serial/parallel multiplier.

with pipelining at the gate level, the full adder, followed by a D flip-flop, causes the latency to become one clock cycle. Model 1 generally results in faster bit-serial implementations, due to the shorter logic paths between the flip-flops in successive operations [30].

For low-complexity implementation of a serial/parallel multiplier, bit-serial adders may be used. The corresponding latency models for a serial/parallel multiplier are shown in Figure 1.17. Denoting the number of fractional bits of the coefficient w_c, the latencies become w_c for latency model 0, and $w_c + 1$ for latency model 1.

A digit-serial multiplier, which accumulate several bits in each stage, can be obtained either via unfolding of a bit-serial multiplier or via folding of a bit-parallel multiplier [10]. The execution time is $\lceil w/d \rceil$ clock cycles for a digit-serial adder. For a serial/parallel multiplier, it is $\lceil w/d \rceil + \lceil w_\alpha/d \rceil$ clock cycles, where w and w_α are the data and coefficient word lengths, respectively.

1.4 Sum-of-Products Circuits

Multimedia and communication applications involve real-time audio and video/image processing which very often require sum-of-products (SOP) computation. The SOP of two N-point vectors is given by

$$y = \sum_{i=1}^{N} a_i x_i \tag{1.8}$$

SOP is a common operations in most DSP applications, for example, IIR and FIR filters and correlation. Sum-of-product is also referred to as inner products of two vectors. Most digital signal processors are optimized to perform multiply-accumulate operations.

When N is large or varies, it is not a constant, the SOP given by Eq. (1.8) is computed sequentially using a multiply-accumulate circuit (MAC) by repeating the MAC operation shown in Figure 1.18 N times.

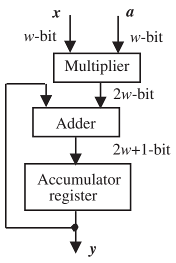


Figure 1.18 Multiplier-accumulator.

Typically, the multiplier is realized by modified radix-4 Booth's algorithm (MBA) [16,31], a carry-save adder (CSA) tree to add the partial products, and a final adder, similar to the multipliers discussed in a previous section.

The delay of a Wallace tree is proportional to $\mathcal{O}(\log(N))$, where N is the number of inputs. However, the throughput is limited by the long critical path for multiplication. (4:2) or (7:3) compressors may therefore be used to reduce the number of outputs in each step. Typically, the accumulations are combined with the adder tree that compresses partial products, for example, it is possible to use two separate $N/2$-bit adders instead of one N-bit adder to accumulate the N-bit MAC result.

It is instructive to study the multiplier-accumulator in more detail. Typically, in the first stage some form of Booth encoding is used to reduce the number of partial products. The adder and multiplier are usually realized by carry-select or carry-save adders, as throughput is of utmost importance. In addition, pipelining is often used to increase the throughput further.

The adder must be operated $N \times 1$ times, which will be expensive if we use a conventional techniques with carry propagation. However, using carry-save adders (CSA), we eliminate the need for carry propagation, except of the last addition. In the intermediate steps we use a redundant number representation with separate sum and carry words. A CSA is a 3:2 compressor, that is, a full adder. Hence, the intermediate results are represented by a sum and carry word.

In the final step, however, we most often need to use a nonredundant representation for the further processing of the SOP. This step involves a vector-merging adder, which may be realized using any of the previously discussed fast adders.

The multiplier typically consists of a partial-product generation stage and an adder-tree. Often Booth encoding is used to reduce the number of partial products. A Wallace carry-save adder-tree is typically used for long word lengths, while overturned-stairs may be better for shorter word length due to the more regular wire routing.

The resulting realization is shown in Figure 1.19, where the dashed box indicates that parts that use a redundant carry-free representation. Note that the output bits (sum and carry words) of the accumulator register can be

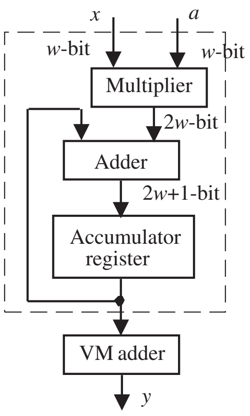


Figure 1.19 Carry-save multiplier-accumulator.

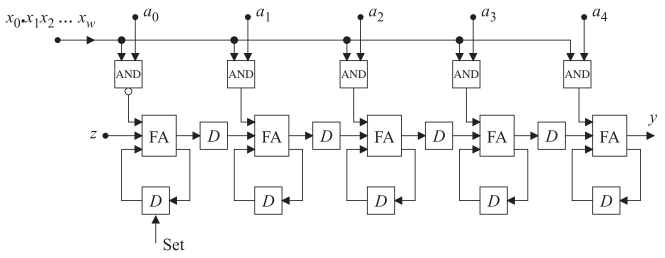


Figure 1.20 Serial/parallel multiplier-accumulator.

partially merged into the adder-tree of the multiplier. It is favourable to insert the sum and carry bits as early as possible in the tree in order to fully exploit the compressors. It may even be favourable to use 4:2 compressors.

1.4.1 SOP Computation

Alternatively, SOP can be realized by serial/parallel multiplier with an additional input that allows computations of the type $y = a \times x + z$ (is shown in Figure 1.20). The extra input allows a value z to be added at the same level of significance as x. A multiplier-accumulator is obtained if the output y is truncated/rounded to the same word length as x and added to the subsequent multiplication. A full precision multiplier-accumulator is obtained if the part of y that is truncated is saved and used to set the sum D flip-flops instead of resetting them at the start of a multiplication.

1.4.2 Linear-Phase FIR Filters

Multiplier-accumulator processing elements are suitable for implementing linear-phase FIR filters, where Eq. (1.8) can be rewritten like

$$y = \sum_{i=1}^{N/2} a_i \left(x_i \pm x_{N-i}\right) \tag{1.9}$$

where we have assumed that N is even. This requires pre-addition of the two delayed input samples. This addition can be incorporated into the adder-tree of the multiplier.

FIR filters realized in the transposed direct form can be implemented by replacing the accumulator register with a register bank that stores the values in the delay elements.

1.4.3 Polynomial Evaluation (Horner's Method)

A related case to SOP is the evaluation of a polynomial, or a series expansion, using Horner's method[1]. An expression of the form

$$y = \sum_{i=1}^{N} a_i x^i \tag{1.10}$$

is rewritten

$$y = \left(\dots\left(\left(\left(0 + a_N\right) x + a_{N-1}\right) x + a_{N-2}\right) x + \dots + a_1\right) x \tag{1.11}$$

Figure 1.21 shows how the generic MAC processor can be modified to evaluate Eq. (1.11). Also in this case can the addition be incorporated into the adder-tree of the multiplier.

1.4.4 Multiple-Wordlength SOP

Different applications could have different requirements on accuracy and dynamic range. For example, it is desirable to use the same hardware to process both music and images, but it may be inefficient if the image processing is dominant, since the former require a word length in the range 16–24 bits, while for the latter it is often sufficient with eight bits. However, it is possible to segment a carry-save multiply-accumulate processor with a long word length. For example, a 64×64-bit SOP processor may be segmented to simultaneously perform either two 32×32-bit, four 16×16-bit, or eight 8×8-bit SOP

1 William Horner, 1819. In fact, this method was derived by Isaac Newton in 1711.

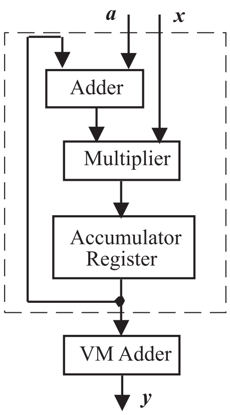


Figure 1.21 Carry-save realization of Horner's method.

products, However, this requires that some additions circuitry to be inserted into the adder-tree to separate input and outputs of different segments.

1.5 Squaring

Squaring is a special case of multiplication, which is relevant since the complexity can be significantly reduced compared to a general multiplication. Squaring finds applications in, for example polynomial evaluation through Estrin's method, see Section 1.7.2, integer exponentiation, and in certain DSP algorithms such as spectrum analysis.

1.5.1 Parallel Squarers

The partial product array for a six-bit squarer using unsigned multiplication as in Figure 1.6 is shown in Figure 1.22. It can be noted that each partial product $x_i x_j$ appears twice when $i \neq j$. Hence, it is possible to replace these two partial products with one partial product in the column with next higher significance. This results in the folded partial product array illustrated in Figure 1.23. Here, it is also utilized that $x_i x_i = x_i$, so no gates are needed to compute those partial products.

It can be noted from Figure 1.23 that the middle column will always contain the most partial products. To reduce the maximum number of partial products it is possible to apply the identity from Reference [32]

$$x_i + x_i x_{i+1} = 2x_i x_{i+1} + \overline{x_i} x_{i+1} \tag{1.12}$$

This results in the partial product array shown in Figure 1.24.

$$\begin{array}{ccccccccccccc}
 & & & & & & \times & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
 & & & & & & & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
\cline{7-13}
 & & & & & & & x_6x_1 & x_6x_2 & x_6x_3 & x_6x_4 & x_6x_5 & x_6x_6 \\
 & & & & & & x_5x_1 & x_5x_2 & x_5x_3 & x_5x_4 & x_5x_5 & x_5x_6 & \\
 & & & & & x_4x_1 & x_4x_2 & x_4x_3 & x_4x_4 & x_4x_5 & x_4x_6 & & \\
 & & & & x_3x_1 & x_3x_2 & x_3x_3 & x_3x_4 & x_3x_5 & x_3x_6 & & & \\
 & & & x_2x_1 & x_2x_2 & x_2x_3 & x_2x_4 & x_2x_5 & x_2x_6 & & & & \\
+ & & x_1x_1 & x_1x_2 & x_1x_3 & x_1x_4 & x_1x_5 & x_1x_6 & & & & & \\
\hline
 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 & z_9 & z_{10} & z_{11} & z_{12}
\end{array}$$

Figure 1.22 Partial products for a six-bit squarer.

$$\begin{array}{ccccccccccccc}
 & & & & & & \times & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
 & & & & & & & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
\cline{7-13}
 & & x_1x_2 & x_1x_3 & x_2x_3 & x_2x_4 & x_3x_4 & x_3x_5 & x_4x_5 & x_4x_6 & x_5x_6 & & x_6 \\
 & & x_1 & & x_1x_4 & x_1x_5 & x_2x_5 & x_2x_6 & x_3x_6 & & x_5 & & \\
 & & & & x_2 & & x_1x_6 & & x_4 & & & & \\
+ & & & & & & x_3 & & & & & & \\
\hline
 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 & z_9 & z_{10} & 0 & x_6
\end{array}$$

Figure 1.23 Folded partial products for a six-bit squarer.

$$\begin{array}{ccccccccccccc}
 & & & & & & \times & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
 & & & & & & & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
\cline{7-13}
 & x_1x_2 & \overline{x_1}x_2 & x_1x_3 & \overline{x_2}x_3 & x_2x_4 & \overline{x_3}x_4 & x_3x_5 & \overline{x_4}x_5 & x_4x_6 & \overline{x_5}x_6 & & x_6 \\
 & & & x_2x_3 & x_1x_4 & x_1x_5 & x_2x_5 & x_2x_6 & x_3x_6 & x_5x_6 & & & \\
+ & & & & & x_3x_4 & x_1x_6 & x_4x_5 & & & & & \\
\hline
 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 & z_9 & \overline{x_5}x_6 & 0 & x_6
\end{array}$$

Figure 1.24 Partial products for a six-bit squarer after applying Eq. (1.12) to reduce the maximum number of partial products in a column.

$$\begin{array}{ccccccccccccc}
 & & & & & & \times & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
 & & & & & & & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\
\cline{7-13}
 & x_1x_2 & b_{1,2,3} & a_{1,2,3} & b_{2,3,4} & a_{2,3,4} & b_{3,4,5} & a_{3,4,5} & b_{4,5,6} & a_{4,5,6} & \overline{x_5}x_6 & & x_6 \\
 & & & & x_1x_4 & x_1x_5 & x_2x_5 & x_2x_6 & x_3x_6 & & & & \\
+ & & & & & & x_1x_6 & & & & & & \\
\hline
 & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 & a_{4,5,6} & \overline{x_5}x_6 & 0 & x_6
\end{array}$$

Figure 1.25 Partial products for a six-bit squarer after applying Eq. (1.13) to reduce the number of partial products.

A method for reducing the number of partial products at the expense of slightly more complicated partial product generation was suggested in Reference [33]. Here, it is noticed that the partial products $\overline{x_i}x_{i+1}$, x_ix_{i+2}, and $x_{i+1}x_{i+2}$ can never be all at the same time, and, hence, the following identity can be applied:

$$2\overline{x_i}x_{i+1} + x_ix_{i+2} + x_{i+1}x_{i+2} = 2b_{i,i+1,i+2} + a_{i,i+1,i+2} \tag{1.13}$$

where the expressions for $b_{i,i+1,i+2}$ and $a_{i,i+1,i+2}$ are straightforward to derive. The resulting partial product array is shown in Figure 1.25. Both methods discussed above can be adapted to be used for signed squarers.

1.5.2 Serial Squarers

To derive a suitable algorithm sequentially computing the square x^2 of a number x, we will first discuss the special case where x is a fractional, unsigned binary number as in Eq. (1.1) [34,35]. Now, let the function $f(x)$ represent the square of the number x, that is,

$$f(x) = x^2 \tag{1.14}$$

The computation of $f(x)$ can be carried out in w iterations by repeatedly squaring the sum of the most significant bit of a number and the other bits of that number. In the first step, $f(x)$ is decomposed into the square of the most significant bit of x with a rest term and a remaining term.

$$\begin{aligned} f_1 &= f\left(\sum_{k=1}^{w} x_k 2^{-k}\right) = \left(\sum_{k=1}^{w} x_k 2^{-k}\right)^2 \\ &= \left(x_1 2^{-1} + \sum_{k=2}^{w} x_k 2^{-k}\right)^2 \\ &= x_1 2^{-2} + x_1 2^0 \sum_{k=2}^{w} x_k 2^{-k} + \underbrace{f\left(\sum_{k=2}^{w} x_k 2^{-k}\right)}_{f_2} \end{aligned} \tag{1.15}$$

In the next step, $f_2 = f(x - x_1)$ is decomposed in the same manner into the square of the most significant bit of $x - x_1$ with sum of other terms, resulting finally with a remaining square $f_3 = f(x - x_1 - x_2)$. The scheme is repeated as long as there are bits left to process. Examining this scheme we find that in order to input a bit-serial word x with the least significant bit first, we have to reverse the order of the iterations in the scheme above. The iterative algorithm then can be written as

$$f_j = \Delta_j + f_{j+1} \tag{1.16}$$

where

$$j = w,\ w-1,\ \ldots,\ 1 \tag{1.17}$$

$$\Delta_j = 2^{-2j} x_j + 2^{1-j} x_j \sum_{k=j+1}^{w} x_k 2^{-k} \tag{1.18}$$

In each iteration j we accumulate the previous term f_{j+1} and input the next bit x_j. If $x_j = 1$, then we add the square of the bit weight and the weights of the bits that have arrived prior to bit x_j shifted left $(1 - j)$ positions. Examination

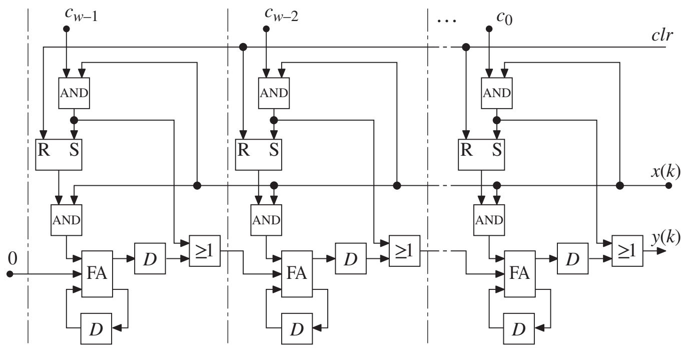


Figure 1.26 Serial squarer for unsigned numbers.

of the bit weights accumulated in each iteration reveals that the sum converges toward the correct square with at least one bit in each step, going from the least-significant bit toward more significant bits in the result.

An implementation of the algorithm above is shown in Figure 1.26. It uses a shift-accumulator to shift the accumulated sum to the right after each iteration. Thus, left-shifting of the stored x_i's are avoided and the addition of the squared bit weight of the incoming x_j is reduced to a shift to the left in each iteration. The implementation consists of w regular bit-slices, which make it suitable for hardware implementation.

The operation of the squarer in Figure 1.26 is as follows: All D flip-flops and SR flip-flops are assumed to be cleared before the first iteration. In the first iteration, the control signal c_0 is high while the remaining control signals are low. This allows the first bit $x(0) = x_w$ to pass the AND gate on top of the rightmost bit-slice. The value x_w is then stored in the SR flip-flop of the same bit-slice for later use. Also, x_w is added to the accumulated sum via the OR gate in the same bit-slice. The least significant output bit $y(0) = y_{2w}$ then becomes available at the output after pure gate delays. Then a shift to the left follows.

The following iterations are carried out in the same manner, with the input bits $x(i) = xw - i$ in sequence along with one of the control signals c_i high, respectively. During the iterations, the result bits $y(i) = y_{2w-i}$ will become available at the output. Then, $x(i)$ has to be zero-extended w locations to access the bits stored in the accumulator. The last control signal clr is used to clear the SR flip-flops before the squaring of next operand can take place.

To adapt the squarer to the case of a two's-complement number as in Eq. (1.2), we sign-extend a two's-complement number to at least $2w - 1$ bits and do not clear the SR flip-flops until the next computation is to take place. Then,

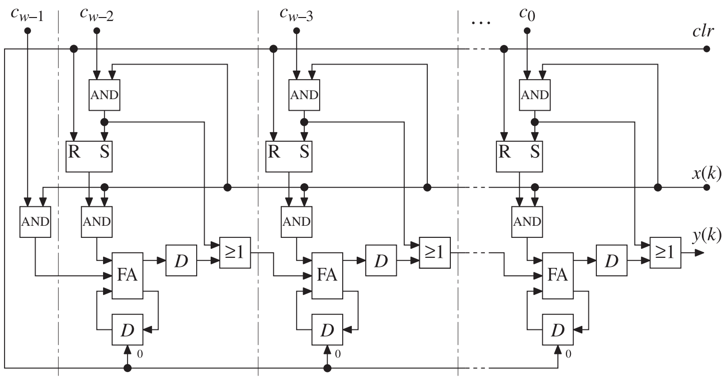


Figure 1.27 Serial squarer for two's-complement numbers.

the squarer computes

$$y = \sum_{j=1}^{w} x_0 2^{j+1} \sum_{k=1}^{w} x_k 2^{-k} + \left(\sum_{k=0}^{w} x_k 2^{-k}\right)^2$$
$$= x_0 2^{w+1} \sum_{k=1}^{w} x_k 2^{-k} + \underbrace{\left(-x_0 + \sum_{k=1}^{w} x_k 2^{-k}\right)^2}_{x^2} \tag{1.19}$$

Here, we can see that the result will contain an error in the accumulated sum. But, since this error exists only in bits of higher significance than the desired result, this error will not affect the desired output result. Further, if we sign extend the input signal with more than $2w - 1$ bits, the error will be scaled accordingly, and stay within the squarer. An implementation of a squarer for two's-complement numbers is shown in Figure 1.27. The only drawback compared to the squarer in Figure 1.26, is that we now need to clear the error in the accumulator before the next squaring can take place.

1.5.3 Multiplication Through Squaring

It is possible to multiply two numbers by using only squarers. This primarily finds applications in memory-based computations as storing the square of a number require significantly less memory compared to storing the product of two numbers. The product of two numbers, a and b, can be computed as [36]

$$a \times b = \frac{(a+b)^2 - (a-b)^2}{4} \tag{1.20}$$

Alternative expressions are also available.

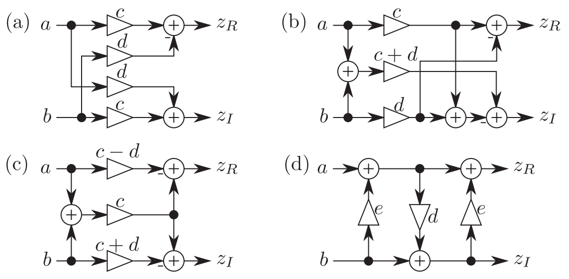


Figure 1.28 Complex multipliers based on (a) Eq. (1.21), (b) Eq. (1.22), (c) Eq. (1.23), and (d) lifting for reversible transforms.

1.6 Complex Multiplication

Multiplication of complex numbers is a time and area consuming arithmetic operation. It is required in many applications, for example, in a fast Fourier transform (FFT) processor for use in an orthogonal frequency-division multiplex (OFDM) system. Multiplication of complex numbers can also be considered into two cases: when both the multiplier and multiplicand are variable or when only the multiplicand is variable.

1.6.1 Complex Multiplication Using Real Multipliers

Consider the complex multiplication $z = k \times x$, where x is the multiplicand, $x = a + jb$ and k is the multiplier, $k = c + jd$. We have

$$z = (c + jd)(a + jb) = (ca - db) + j(da + cb) \tag{1.21}$$

A direct realization of Eq. (1.21) requires four multipliers and two adders/subtractors as shown in Figure 1.28a.

We may rewrite Eq. (1.21) as follows:

$$\begin{aligned} z_R &= ca - db \\ z_I &= (c + d)(a + b) - ca - db \end{aligned} \tag{1.22}$$

which requires only three real multiplications and five real additions. An alternative version is

$$\begin{aligned} z_R &= c(a + b) - (c + d)b \\ z_I &= c(a + b) - (c - d)a \end{aligned} \tag{1.23}$$

which also requires only three real multiplications and five real additions. In fact, there exist several such expressions with only three multiplications [37].

In many applications, for example, FFT, the multiplier, $c + jd$, is constant and $c \pm d$ can be precomputed. This reduces a complex multiplication to only three real multiplications and three/four real additions, as shown in Figure 1.28b and c, for Eqs (1.22) and (1.23), respectively. Obviously, multiplications by ± 1 and $\pm j$ are easily implemented. In addition, further savings can be made for multipliers of the form $\pm c \pm jc$ since the complex multiplication reduces to two real additions of a and b with different signs and two real multiplications by c. This case applies when $\cos(n\pi/4) \pm j\sin(n\pi/4)$ and $n =$ odd. This simplification allows an eight-point DFT to be realized by using only four real multiplications. Hence, a complex multiplication requires three real multiplications and some additions. A more efficient technique is based on distributed arithmetic where only two units are required, which from chip area, throughput, and power consumption points of view are comparable to two real multipliers [3].

1.6.2 Lifting-Based Complex Multipliers

In many transforms and other applications, the complex multiplication is a rotation with an angle α, that is, $k = c + jd = \cos(\alpha) + j\sin(\alpha)$. First, it should be noted that when $\cos(\alpha)$ and $\sin(\alpha)$ are represented in a binary representation, the magnitude of the complex number can never be exactly one (unless $\alpha = n\pi/2$ rad). This in turn means that the inverse rotation can never be exact, there will always be a gain and/or angle error. To get around this, a lifting-based complex multiplier can be used [38]. The lifting-based complex multiplication is written in matrix form as

$$\begin{aligned}
\begin{bmatrix} z_R \\ z_I \end{bmatrix} &= \begin{bmatrix} c & -d \\ d & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \\
&= \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \\
&= \begin{bmatrix} 1 & \frac{\cos(\alpha)-1}{\sin(\alpha)} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin(\alpha) & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{\cos(\alpha)-1}{\sin(\alpha)} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \qquad (1.24) \\
&= \begin{bmatrix} 1 & e \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ d & 1 \end{bmatrix} \begin{bmatrix} 1 & e \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \qquad (1.25)
\end{aligned}$$

where

$$e = \frac{\cos(\alpha) - 1}{\sin(\alpha)} = \frac{c - 1}{d} \qquad (1.26)$$

The inverse rotation can similarly be written as

$$\begin{aligned}\begin{bmatrix}\cos(\alpha) & -\sin(\alpha)\\ \sin(\alpha) & \cos(\alpha)\end{bmatrix}^{-1} &= \left(\begin{bmatrix}1 & e\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ d & 1\end{bmatrix}\begin{bmatrix}1 & e\\ 0 & 1\end{bmatrix}\right)^{-1}\\ &= \begin{bmatrix}1 & e\\ 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}1 & 0\\ d & 1\end{bmatrix}^{-1}\begin{bmatrix}1 & e\\ 0 & 1\end{bmatrix}^{-1}\\ &= \begin{bmatrix}1 & -e\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ -d & 1\end{bmatrix}\begin{bmatrix}1 & -e\\ 0 & 1\end{bmatrix}\end{aligned} \tag{1.27}$$

Now, independent of what the values d and e are after quantization, the forward and reverse rotations will always cancel exactly. However, it should be noted that evaluating the three matrix multiplications leads to

$$\begin{bmatrix}1 & e\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ d & 1\end{bmatrix}\begin{bmatrix}1 & e\\ 0 & 1\end{bmatrix} = \begin{bmatrix}1+de & e(de+2)\\ d & 1+de\end{bmatrix} \tag{1.28}$$

and that the $\sin(\alpha)$ terms of the resulting rotation are different, although similar since $d \approx -e(de+2)$.

The lifting-based complex multiplier can be efficiently realized using three multiplications and three additions as shown in Figure 1.28d. However, a potential drawback from an implementation perspective is that all operations are sequential.

1.7 Special Functions

The need of computing non-linear functions arises in many different applications. The straightforward method of approximating an elementary function is of course to just store the values in a look-up table typically leads to large tables, even though the resulting area from standard cell synthesis grows slower than the number of memory bits [39]. Instead, it is of interest to find ways to approximate elementary functions using a trade-off between arithmetic operations and look-up tables. In addition, the CORDIC algorithm, which is discussed in a separate chapter, is an efficient approach for computing certain functions. For a more thorough explanation of these and other methods, the readers may refer to Reference [40] and other chapters of this book.

1.7.1 Square Root Computation

Computing the square root is commonly performed in one of two different iterative ways, by computing one digit per iteration or by iterative refinement of a temporary result. The methods used are similar to division algorithms because the computation of square root can be seen as dividing the radicand, x, with the square root, $z = \sqrt{x}$, as, $z = x/z$.

1.7.1.1 Digit-Wise Iterative Square Root Computation

While only the binary (radix-2) case is considered here, generalizations to higher radices can be performed. Assuming $0 < X < 1$ and using a "reminder" after the ith iteration, r_i, initialized to the radicand, $r_0 = X$, and a partially computed square root

$$Z_i = \sum_{k=1}^{i} z_k 2^{-k} \tag{1.29}$$

one digit of the result, z_i, is determined in iteration i. After each iteration, we have

$$X = Z_i Z_i + 2^{-i} r_i \tag{1.30}$$

Clearly, if $2^{-i} r_i$ is smaller than $2^{-i+1} r_{i-1}$, the result Z_i will be more accurate than Z_{i-1}. In general, the iteration for square root extraction is

$$r_i = 2r_{i-1} - z_i \left(2Z_{i-1} + z_i 2^{-i}\right) = 2r_{i-1} - 2Z_{i-1} z_i - z_i^2 2^{-i} \tag{1.31}$$

Schemes similar to restoring, nonrestoring, and SRT division can be defined for selecting the next digit [41]. For the quotient digit selection scheme similar to SRT division, the square root is usually restricted to $1/2 \le z < 1$, which corresponds to $1/4 \le x < 1$. The selection rule is then

$$z_i = \begin{cases} 1, \frac{1}{2} \le 2r_{i-1} \\ 0, -\frac{1}{2} \le 2r_{i-1} < \frac{1}{2} \\ -1, 2r_{i-1} < -\frac{1}{2} \end{cases} \tag{1.32}$$

Meaning that only the first few bits of $2r_{i-1}$ must be inspected to determine the correct digit of the result. Note that the normalization, $1/4 \le x < 1$, always gives $z_1 = 1$ and, hence, $r_1 = 2x - \frac{1}{2}$.

1.7.1.2 Iterative Refinement Square Root Computation

The Newton–Raphson method for solving equation systems can be applied for square root computations. The Newton–Raphson recurrence is

$$x_{i+1} = x_i - \frac{f\left(x_i\right)}{f'\left(x_i\right)} \tag{1.33}$$

and the equation to solve in this case is $f(z) = z^2 - x = 0$. This gives the recurrence

$$z_{i+1} = \frac{1}{2}\left(z_i + \frac{x}{z_i}\right) \tag{1.34}$$

Hence, each iteration requires a division, an addition, and a bit-shift.

Instead, it turns out that it is more efficient to compute the reciprocal square root $\rho = 1/\sqrt{x}$. Solving the equation $f(\rho) = 1/\rho^2 - z$, the recurrence for this computation is

$$\rho_{i+1} = \rho_i\left(\frac{1}{2}\left(3 - \rho_i^2 x\right)\right) \tag{1.35}$$

Each iteration now requires a square, two multiplications, subtraction from a constant, a bit-shift. Although there are more computations involved, the operations are simpler to implement compared to a division. The first estimate, ρ_0, can be read from a table. Then, for each iteration, the error is reduced quadratically.

The square root can then easily be determined from the reciprocal square root as $z = x\rho$.

1.7.2 Polynomial and Piecewise Polynomial Approximations

It is possible to derive a polynomial $p(x)$ that approximates a function $f(x)$ by performing a Taylor expansion for a given point d as

$$p(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(d)}{k!}(x-d)^k \tag{1.36}$$

When the polynomial is restricted to a certain number of terms it is often better to optimize the polynomial coefficients, as there are some accuracy to be gained. To determine the best coefficients is an approximation problem where typically there are more constraints (number of points for the approximation) than variables (polynomial order). This problem can be solved for a minimax solution using, for example, Remez's exchange algorithm or linear programming. For a least square solution the standard method to solve overdetermined systems can be applied. The result will be a polynomial of order N

$$p(x) = \sum_{k=0}^{N} a_k x^k \tag{1.37}$$

The polynomial approximations can be efficiently and accurately evaluated using Horner's method, discussed in Section 1.4.3. Hence, there is no need to compute any powers of X explicitly and a minimum number of arithmetic operations are used.

The drawback of Horner's method is that the computation is inherently sequential. An alternative is to use Estrin's method [40], where x^2 is used in a tree structure for increasing the parallelism and reducing the critical path. Estrin's method for polynomial evaluation can be written as

$$p(x) = \left(a_3x + a_2\right) x^2 + \left(a_1x + a_0\right) \tag{1.38}$$

for a third-order polynomial. For a seventh-order polynomial, it becomes

$$p(x) = \left(\left(a_3x + a_2\right) x^2 + \left(a_1x + a_0\right)\right) x^4 + \left(\left(a_3x + a_2\right) x^2 + \left(a_1x + a_0\right)\right) \tag{1.39}$$

As can be seen, Estrin's method also maps well to MAC-operations.

The required polynomial order depends very much on the actual function that is approximated [40]. An approach to obtain a higher resolution despite using a lower polynomial order is to use different polynomials for different ranges. This is referred to as piecewise polynomials. An L segment Nth-order piecewise polynomial with segment breakpoints x_l, $l = 0,\ 1,\ \dots,\ L-1$ can be written as

$$p(x) = \sum_{k=0}^{N} a_{k,l} \left(x - x_l\right)^k,\ x_l \le x \le x_{l+1} \tag{1.40}$$

From an implementation point of view, it is often practical to have 2^k uniform segments and let the k most significant bits determine the segmentation, as these directly forms the segment number l. However, it can be shown that in general the total complexity is reduced for nonuniform segments. An illustration of a piecewise polynomial approximation is shown in Figure 1.29 where uniform segments and a parallel implementation of Horner's method is used for the polynomial evaluation.

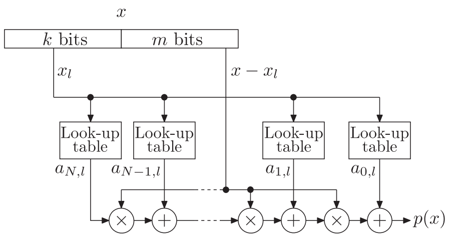


Figure 1.29 Piecewise polynomial approximation using uniform segmentation based on the k most significant bits.

References

1 D. Liu, *Embedded DSP Processor Design: Application Specific Instruction Set Processors*, Morgan Kaufmann, 2008.

2 A. P. Chandrakasan and R. W. Brodersen, Minimizing power consumption in digital CMOS circuits. *Proc. IEEE*, vol. 83, no. 4, 1995, pp. 498–523.

3 L. Wanhammar, *DSP Integrated Circuits*, Academic Press, 1999.

4 K. Johansson, O. Gustafsson, and L. Wanhammar, Power estimation for ripple-carry adders with correlated input data, in *Proc. Int. Workshop Power Timing Modeling Optimization Simulation*, Springer, January 2004, pp. 662–674.

5 R. Zimmermann, Binary adder architectures for cell-based VLSI and their synthesis, Ph.D. dissertation, Swiss Federal Institue of Technology (ETH), 1998.

6 T. G. Noll, Carry-save architectures for high-speed digital signal processing. *J. VLSI Signal Process. Syst.*, vol. 3, no. 1–2, 1991, pp. 121–140.

7 A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*. Imperial College Press, 2007.

8 Y.-N. Chang, J. H. Satyanarayana, and K.K. Parhi, Systematic design of high-speed and low-power digit-serial multipliers. *IEEE Trans. Circuits Syst. II*, vol. 45, no. 12, 1998, pp. 1585–1596.

9 K. Johansson, O. Gustafsson, and L. Wanhammar, Multiple constant multiplication for digit-serial implementation of low power FIR filters. *WSEAS Trans. Circuits Syst.*, vol. 5, no. 7, 2006, pp. 1001–1008.

10 K. K. Parhi, A systematic approach for design of digit-serial signal processing architectures. *IEEE Trans. Circuits Syst.*, vol. 38, no. 4, 1991, pp. 358–375.

11 H. Suzuki, Y.-N. Chang, and K. K. Parhi, Performance tradeoffs in digit-serial DSP systems, in *Proc. Asilomar Conf. Signals Syst. Comput.*, vol. 2, November 1998, pp. 1225–1229.

12 P. Nilsson, Arithmetic and architectural design to reduce leakage in nano-scale digital circuits, in *Proc. Europ. Conf. Circuit Theory Design*, August 2007, pp. 372–375.

13 C. R. Baugh and B. A. Wooley, A two's complement parallel array multiplication algorithm. *IEEE Trans. Comput.*, vol. C-22, no. 12, 1973, pp. 1045–1047.

14 B. C. Paul, S. Fujita, and M. Okajima, ROM-based logic (RBL) design: a low-power 16 bit multiplier. *IEEE J. Solid-State Circuits*, vol. 44, no. 11, 2009, pp. 2935–2942.

15 M. Faust, O. Gustafsson, and C.-H. Chang, Fast and VLSI efficient binary-to-CSD encoder using bypass signal. *Electron. Lett.*, vol. 47, no. 1, 2011, pp. 18–20.

16 O. L. Macsorley, High-speed arithmetic in binary computers. *Proc. IRE*, vol. 49, no. 1, 1961, pp. 67–91.

17 Y. H. Seo and D. W. Kim, A new VLSI architecture of parallel multiplier–accumulator based on radix-2 modified Booth algorithm. *IEEE Trans. VLSI Syst.*, vol. 18, no. 2, 2010, pp. 201–208.

18 Y. C. Lim, Single-precision multiplier with reduced circuit complexity for signal processing applications. *IEEE Trans. Comput.*, vol. 41, no. 10, 1992, pp. 1333–1336.

19 J. P. Wang, S. R. Kuang, and S. C. Liang, High-accuracy fixed-width modified Booth multipliers for lossy applications. *IEEE Trans. VLSI Syst.*, vol. 19, no. 1, 2011, pp. 52–60.

20 D. D. Caro, N. Petra, A. G. M. Strollo, F. Tessitore, and E. Napoli, Fixed-width multipliers and multipliers-accumulators with min-max approximation error. *IEEE Trans. Circuits Syst. I*, vol. 60, no. 9, 2013, pp. 2375–2388.

21 C. S. Wallace, A suggestion for a fast multiplier. *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 1, 1964, pp. 14–17.

22 L. Dadda, Some schemes for parallel multipliers. *Alta Frequenza*, vol. 34, no. 5, 1965, pp. 349–356.

23 K. C. Bickerstaff, M. J. Schulte, and E. E. Swartzlander, Parallel reduced area multipliers. *J. VLSI Signal Process. Syst.*, vol. 9, no. 3, 1995, pp. 181–191.

24 P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, Optimal circuits for parallel multipliers. *IEEE Trans. Comput.*, vol. 47, no. 3, 1998, pp. 273–285.

25 S. T. Oskuii, P. G. Kjeldsberg, and O. Gustafsson, Power optimized partial product reduction interconnect ordering in parallel multipliers, in *Proc. Norchip.*, November 2007, pp. 1–6.

26 Z. J. Mou and F. Jutand, overturned-stairs adder trees and multiplier design. *IEEE Trans. Comput.*, vol. 41, no. 8, 1992, pp. 940–948.

27 H. Eriksson, P. Larsson-Edefors, M. Sheeran, M. Sjalander, D. Johansson, and M. Scholin, Multiplier reduction tree with logarithmic logic depth and regular connectivity, in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2006, pp. 4–8.

28 P. F. Stelling and V. G. Oklobdzija, Design strategies for optimal hybrid final adders in a parallel multiplier. *J. VLSI Signal Process. Syst.*, vol. 14, no. 3, 1996, pp. 321–331.

29 R. Lyon, Two's complement pipeline multipliers. *IEEE Trans. Commun.*, vol. 24, 1976, no. 4, 1976, pp. 418–425.

30 O. Gustafsson and L. Wanhammar, Optimal logic level pipelining for digit-serial implementation of maximally fast recursive digital filters, in *Proc. National Conf. Radio Sciences*, 2002.

31 A. R. Cooper, Parallel architecture modified Booth multiplier. *IEE Proc. G Circuits Devices Syst.*, vol. 135, no. 3, 1988, pp. 125–128.

32 R. K. Kolagotla, W. R. Griesbach, and H. R. Srinivas, VLSI implementation of 350 MHz 0.35 μm 8 bit merged squarer. *Electron. Lett.*, vol. 34, 1998, no. 1, 1998, pp. 47–48.

33 K.-J. Cho and J.-G. Chung, Parallel squarer design using pre-calculated sums of partial products. *Electron. Lett.*, vol. 43, no. 25, 2007, pp. 1414–1416.

34 P. Ienne and M. A. Viredaz, Bit-serial multipliers and squarers. *IEEE Trans. Comput.*, vol. 43, 1994, no. 12, 1994, pp. 1445–1450.

35 M. Vesterbacka, K. Palmkvist, and L. Wanhammar, Serial squarers and serial/serial multipliers, in *Proc. National Conf. Radio Sciences*, 1996.

36 E. L. Johnson, A digital quarter square multiplier. *IEEE Trans. Comput.* vol. 29, no. 3, 1980, pp. 258–261.

37 A. Wenzler and E. Luder, New structures for complex multipliers and their noise analysis, in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 2, April 1995, pp. 1432–1435.

38 S. Oraintara, Y. J. Chen, and T. Q. Nguyen, Integer fast Fourier transform. *IEEE Trans. Signal Process.*, vol. 50, 2002, no. 3, 2002, pp. 607–618.

39 O. Gustafsson and K. Johansson, An empirical study on standard cell synthesis of elementary function lookup tables, in *Proc. Asilomar Conf. Signals Syst. Comput.*, 2008, pp. 1810–1813.

40 J.-M. Muller, *Elementary Functions*. Springer, 2006.

41 M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Elsevier, 2004.