EURASIP Journal on Advances in Signal Processing

RESEARCH Open Access

Using the complement of the cosine to compute trigonometric functions

David Guerrero Martos*, Alejandro Millán Calderón, Jorge Juan Chico, Julian Viejo Cortés, Manuel J. Bellido Díaz, Paulino Ruiz-de-Clavijo Vazquez and Enrique Ostúa Arangüena

*Correspondence: guerre@dte.us.es
Departamento de Tecnología Electrónica, Universidad de Sevilla, Avenida de Reina Mercedes S.N., 41012 Sevilla, Spain

Abstract

The computation of the sine and cosine functions is required in devices ranging from application-specific signal processors to general purpose floating-point units. Even in the latter case, the required functionality can be reduced to computing the sine and/or cosine of multiples of a constant angle. The latency of a sine/cosine generator can be reduced by using look-up tables. However, a direct implementation with look-up tables may be unfeasible if the input space is huge. In such a case, look-up tables with a number of entries lower than the size of the input space can be used indirectly. In previously published methods, the reduction in the number of table entries is obtained at the expense of increasing the table width and the computational cost. This paper introduces an alternative technique that makes it possible to reduce the size of the look-up tables as well as the required multiplications. The proposed technique can be used to implement sine/cosine generators of huge input space. It has been used to implement several twiddle factor generators in reconfigurable hardware and has enabled the number of look-up tables to be reduced by between 6 and 26% with respect to previous table-based techniques. Also, these implementations are about 50% faster than those based on Volder's algorithm.

Keywords: Trigonometric functions, Computational cost, Signal processing, Discrete Fourier transform

1 Introduction

The computation of sine and cosine functions is fundamental in a wide range of applications, including that of signal processing [1, 2]. Obvious examples are the computation of discrete cosine transform (DCT), discrete sine transform (DST), and their inverses (IDCT and IDST) [3]. A fused sine-and-cosine implementation is of major interest because various methods compute both and numerous applications require both [1]. In this paper, the focus is on the implementation of functional units that provide the sine and cosine of multiples of a constant angle ϕ, that is, $\sin(n\phi)$ and $\cos(n\phi)$, where n is an integer given as an input. Applications of such functional units include the following:

- Implementing the sine and/or cosine functions in arithmetic units. For example, suppose an arithmetic unit must compute the sine and/or cosine of a number x using

the IEEE 754-1985 double-precision format [4]. x is coded in a 64-bit word with 3 fields called *sign* (1 bit), *exponent* (11 bits), and *significand* (52 bits). The significand is a number in the range $[1, 2-2^{-52}]$ coded in fixed-point, while the exponent is an integer laying in the range $[-1022, 1023]$ coded in excess 1023. If the exponent is lower than -27, then the unit can simply return 1 as the cosine and x as the sine, assuming rounding to the nearest representable value [5]. Otherwise, it can return the sine and/or cosine of $n\phi$, where ϕ is the constant $2^{-27-52} = 2^{-79}$ and n is the integer $x2^{79}$.

- Generating the set of coefficients, called *twiddle factors*, of a discrete Fourier transform (DFT) [6]. The DFT of a complex sequence x of length L is another complex sequence X of the same length defined by $X(k) = \sum_{t=0}^{L-1} x(t) W_L^{tk}$, where $W_L = e^{\phi i}$ and $\phi = -2\pi/L$. The twiddle factors are the integer powers of W_L, and there are L different twiddle factors. Thus, the twiddle factor of index n is $W_L^n = (e^{\phi i})^n = e^{n\phi i} = \sin(n\phi)i + \cos(n\phi)$.

Since the sine and cosine functions are computationally expensive, in applications where a low latency is required, the generator is implemented using look-up tables (LUT). This implementation approach is problematic if the input space is large. For example, consider the arithmetic unit previously mentioned: as stated, exponents lower than -27 can be dismissed. However, even if the input angle is restricted to $[0, 2\pi)$, it is still necessary to consider 30 different exponent values and 2^{52} different significand values. A direct implementation would therefore require an LUT of $30 * 2^{52}$ entries. Another example is given by the DFT engines required in applications such as required in Power Line Communications (PLC) [7], Digital Video Broadcasting—Terrestrial 2 (DVB-T2) [8], photon counting [9], and radio astronomy [10]. In those applications, the sequence can be as long as 2^{13}, 2^{15}, 2^{27}, and 2^{30}, respectively, and hence the coefficient tables are large in comparison with other elements of the DFT engine [11]. In this paper, we propose an innovative technique to reduce the resources required to implement a sine/cosine generator in application-specific integrated circuit (ASIC) and configurable logic. We have implemented an open-source tool to automate the design of twiddle factor generators of arbitrary size and precision using the proposed technique.

The rest of the paper is organized as follows. In the next section, the notation used is introduced and optimization techniques are presented to reduce the number of entries of the required LUT to a number proportional to the input space. In Section 3, optimization techniques are given that enable LUTs to be employed with a total number of entries that grows sublinearly with the input space. The new proposed technique is introduced in Section 4. The experiments are described in Section 5. The corresponding performance results are shown and discussed in Section 6. The last section provides a summary of the conclusions.

2 Argument reduction

As mentioned in the introduction, our objective is to efficiently implement a functional unit that provides $\sin(n\phi)$ and $\cos(n\phi)$, where n is an integer provided as an input to the unit, and ϕ is a constant angle that depends on the application. Hereinafter, the input of the functional unit will be denoted as I and the number of bits of I will be denoted as w. Furthermore, the following definitions will be used:

Definition 1 *A real number ϕ is trigonometric rational if and only if $\frac{\phi}{\pi}$ is rational.*

For example, the angle $\phi = -2\pi/L$, used in the definition of the twiddle factors in Section 1, is trigonometric rational, while the angle $\phi = 2^{-79}$ mentioned in the arithmetic unit example of Section 1 is not.

Definition 2 *The trigonometric Carmichael function of a trigonometric rational number ϕ is the minimum natural number $\lambda_t(\phi)$ such that $\frac{\lambda_t(\phi)\phi}{2\pi}$ is an integer.*

This definition is useful in the calculation of the size of the output space of the generator. This size is the minimum of $\lambda_t(\phi)$ and the size of the input space. Note that $\lambda_t(0) = 1$. In the DFT example, $\lambda_t(\phi)$ is equal to the length of the transform. The function λ_t can also be employed to make the following simplification. Suppose that ϕ has been defined as a trigonometric rational number whose absolute value is very large: $|\phi| \gg 2\pi$. In this case, an angle α with $|\alpha| < 2\pi$ can be found such that the functionality of the generator, that is, computing $\sin(n\phi)$ and $\cos(n\phi)$, is equivalent to computing $\sin(n\alpha)$ and $\cos(n\alpha)$. In order to obtain such α:

1. Take the integer $k = \frac{\lambda_t(\phi)\phi}{2\pi}$
2. Take the remainder r of the division $\frac{k}{\lambda_t(\phi)}$. Note that k and r have the same sign and $|r| < \lambda_t(\phi)$.
3. $\alpha = \frac{r2\pi}{\lambda_t(\phi)}$.

Definition 3 *The trigonometric Shannon entropy of a trigonometric rational number ϕ is $H_t(\phi) = log_2(\lambda_t(\phi))$.*

If ϕ is trigonometric rational, in order to maximize the size of the output space of the generator, that is, get an output space of size $\lambda_t(\phi)$, the minimum number of bits required to code the input is $\lceil H_t(\phi) \rceil$.

Definition 4 *ϕ is trigonometric binary if and only if it is trigonometric rational and $H_t(\phi)$ is an integer.*

The latter definition is relevant since, in many applications, the constant angle ϕ is trigonometric binary. For example, consider the algorithms designed to compute efficiently the DFT called fast Fourier transform (FFT) [12] algorithms: many of these algorithms require the length of the transform L to be a power of 2 [13], that is, $log_2(L)$ must be an integer, and hence ϕ must be trigonometric binary since $H_t(\phi) = log_2(\lambda_t(-2\pi/L)) = log_2(L)$. Moreover, in applications where ϕ is trigonometric binary, it is irrelevant whether the representation of n is either unsigned or two's complement as long as $w \geq H_t(\phi)$. As an example, consider the functional unit specified in [1]. This unit computes the sine and cosine of πx where x is a number in the interval $[-1, 1)$ coded in fixed-point two's complement. Let S be the value represented by the input I in integer two's complement, $x = S/2^{w-1}$, and hence $\pi x = S\pi/2^{w-1}$. Thus, the functional unit computes the sine and cosine of $S\phi$, where $\phi = \pi/2^{w-1}$. Let n be the value represented by I in unsigned integer representation. It is easy to prove that $\sin(S\phi) = \sin(n\phi)$ and $\cos(S\phi) = \cos(n\phi)$. Therefore, the functionality of the unit is

equivalent to the computation of the sine and cosine of $n\phi$. This is exemplified in Table 1 for $w = 3$.

Hereinafter, an unsigned notation for n is assumed. In the following subsections, we will see optimization techniques that require a trigonometric rational value of ϕ. In these subsections, the trigonometric Carmichael function of ϕ, $\lambda_t(\phi)$, is abbreviated to L.

2.1 Periodicity

If the size of the input space of the functional unit is greater than L, then the periodicity of the sine and cosine can be used to compute $\sin(n\phi)$ and $\cos(n\phi)$ in the following way:

1. Compute $n \bmod L$. As noted by [1], if ϕ is trigonometric binary, then this computation has no cost since the result is simply the $H_t(\phi)$ least significant bits of the input I.
2. Use a subgenerator to compute the sine and cosine of $(n \bmod L)\phi$. The input space of the subgenerator is $\mathbb{Z}_L$, smaller than the original input space, and hence, it can be implemented using a smaller LUT.

In the optimization shown in the following subsections, it is assumed that the input space of the generator is $\mathbb{Z}_L$.

2.2 Sign reduction

If the input space of the functional unit is $\mathbb{Z}_L$, then it is possible to implement it with another subgenerator with the same value of ϕ but whose input space is $\mathbb{Z}_{\lfloor L/2 \rfloor + 1}$, that is, its size is roughly half of the size of the original input space. This optimization takes into account the following trigonometric identities:

$$\begin{aligned} \sin(\alpha) &= -\sin(2\pi - \alpha) \\ \cos(\alpha) &= \cos(2\pi - \alpha) \end{aligned} \tag{1}$$

If $n \leq L/2$, then the input of the subgenerator is n and its output is the output of the functional unit. Otherwise, the input of the subgenerator is $L - n$, the cosine output of the unit is the cosine output of the subgenerator, and the sine output of the unit is the opposite of the sine output of the sine output of the subgenerator. Note that if ϕ is trigonometric binary, then $L - n$

Table 1 Values returned by the functional unit specified in [1] when the input I has exactly three bits ($\phi = \pi/2^{3-1} = \pi/4$)

I	S	n	x ‖ $S/4$	πx ‖ $S\phi$	$n\phi$	$\sin(\pi x)$ ‖ $\sin(S\phi)$ ‖ $\sin(n\phi)$	$\cos(\pi x)$ ‖ $\cos(S\phi)$ ‖ $\cos(n\phi)$
000	0	0	0	0	0	0	1
001	1	1	$\frac{1}{4}$	$\frac{\pi}{4}$	$\frac{\pi}{4}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
010	2	2	$\frac{2}{4}$	$\frac{2\pi}{4}$	$\frac{2\pi}{4}$	1	0
011	3	3	$\frac{3}{4}$	$\frac{3\pi}{4}$	$\frac{3\pi}{4}$	$\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$
100	−4	4	−1	$-\frac{4\pi}{4}$	$\frac{4\pi}{4}$	0	−1
101	−3	5	$-\frac{3}{4}$	$-\frac{3\pi}{4}$	$\frac{5\pi}{4}$	$-\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$
110	−2	6	$-\frac{2}{4}$	$-\frac{2\pi}{4}$	$\frac{6\pi}{4}$	−1	0
111	−1	7	$-\frac{1}{4}$	$-\frac{\pi}{4}$	$\frac{7\pi}{4}$	$-\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$

can be obtained by simply taking the two's complement. In the next subsection, another optimization is presented for the implementation of the subgenerator when L is even.

2.3 Quadrant reduction

If L is even and the size of the input space of the functional unit is $\lfloor L/2 \rfloor + 1 = L/2 + 1$, then it can be implemented using a subgenerator with the same value of ϕ but whose input space is reduced to $\mathbb{Z}_{\lfloor L/4 \rfloor + 1}$. This optimization uses the following trigonometric identities:

$$\begin{aligned} \sin(\alpha) &= \sin(\pi - \alpha) \\ \cos(\alpha) &= -\cos(\pi - \alpha) \end{aligned} \tag{2}$$

If $n \leq L/4$, then the input of the subgenerator is n and its output is the output of the functional unit. Otherwise, the input of the subgenerator is $L/2 - n$, the sine output of the unit is the sine output of the subgenerator, and the cosine output of the unit is the opposite of the cosine output of the subgenerator. Again, the computation of $L/2 - n$ is simply a two's complement if ϕ is trigonometric binary. In turn, the optimization described in the next subsection can be used to implement the subgenerator if L is multiple of 4.

2.4 Octant reduction

Finally, if L is a multiple of 4 and the input space of the functional unit is $\mathbb{Z}_{\lfloor L/4 \rfloor + 1} = \mathbb{Z}_{L/4+1}$, then it can be implemented with a subgenerator with the same value of ϕ but whose input space is reduced to $\mathbb{Z}_{\lfloor L/8 \rfloor + 1}$ by applying the following trigonometric identities:

$$\begin{aligned} \sin(\alpha) &= \cos(\pi/2 - \alpha) \\ \cos(\alpha) &= \sin(\pi/2 - \alpha) \end{aligned} \tag{3}$$

If $n \leq L/8$, then the input of the subgenerator is n and its output is the output of the functional unit. Otherwise, the input of the subgenerator is $L/4 - n$, the sine output of the unit is the cosine output of the subgenerator and the sine output of the unit is the cosine output of the subgenerator. Once more, a simple two's complement provides $L/4 - n$ if ϕ is trigonometric binary.

With the previous optimizations, a sine/cosine generator can be implemented with an LUT of $\lfloor L/8 \rfloor + 1$ entries. For example, the functional unit described in [1] previously mentioned can be implemented as shown in Fig. 1. As previously discussed, the functionality of that unit is equivalent to computing the sine and cosine of $n\phi$, where n is the number provided by input I in unsigned integer notation, the angle ϕ is $2\pi/2^w$, and w is the number of bits of I. In this case, ϕ is a trigonometric binary. In this implementation, the output is provided in some type of sign-magnitude notation such as one of the IEEE 754-1985 floating-point formats, and $w > 3$ so $\lambda_t(\phi)$ is multiple of 4 and an LUT of only $2^{w-3} + 1$ entries is required. The LUT of the figure should return the sine and cosine of angles in the range $[0, \pi/4]$ and, since they are all positive, there is no need to store the sign bits. Instead, they are computed using a simple XOR gate *(4c)*. The LUT has been implemented using a direct access memory. In order to prevent the problem of dealing with a direct access memory with a number of positions that is not a power of two, the access of the entry of the LUT corresponding to $n = 2^{w-3}$ is detected by a simple logic gate *(4a)* and is treated separately. In that case, the LUT returns the sine and cosine of

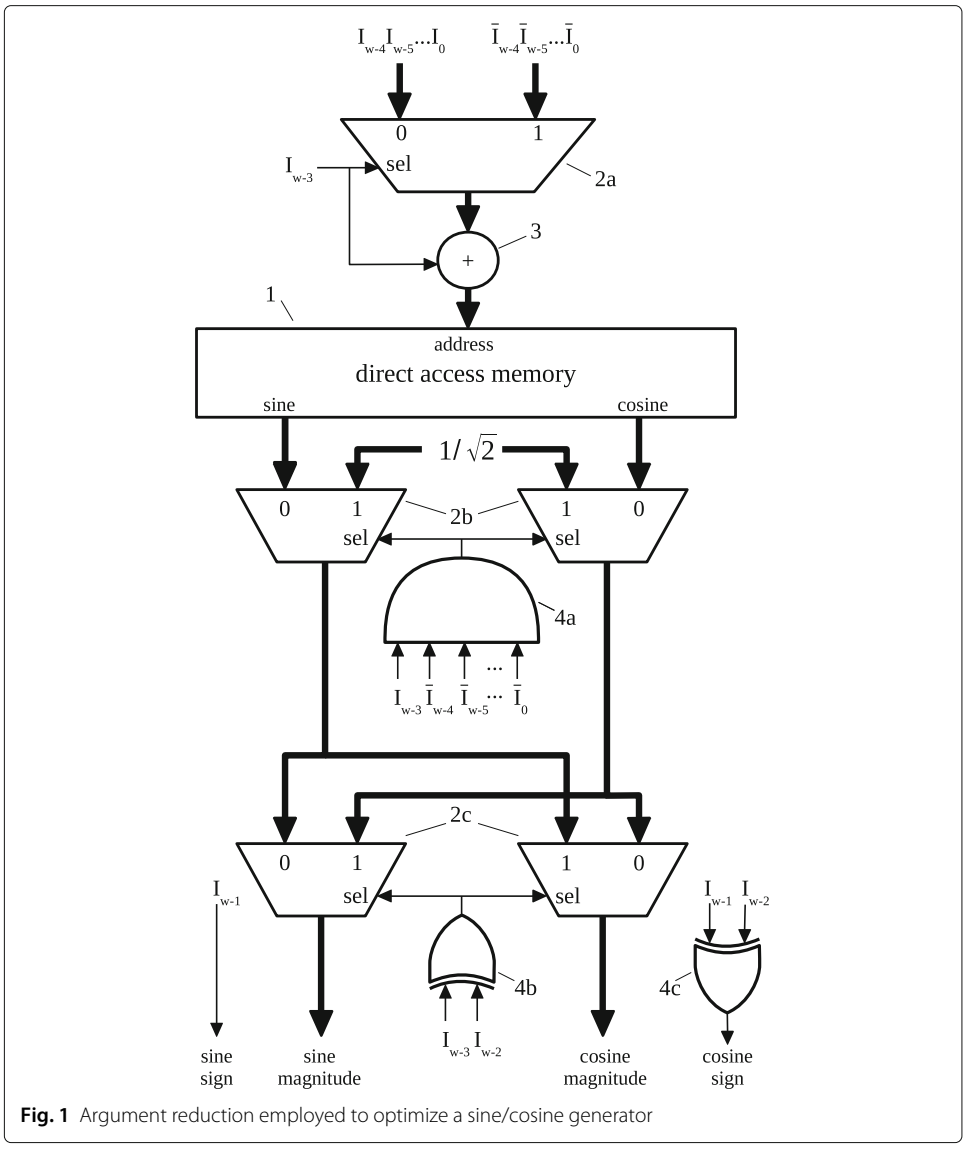

Fig. 1 Argument reduction employed to optimize a sine/cosine generator

$\pi/4$, that is, $1/\sqrt{2}$, using a pair of multiplexers *(2b)*. The address lines of the memory are fed with the $w-3$ least significant bits of I or with its two's complement depending on I_{w-3}, using the adder *(3)* and the multiplexer *(2a)*. The gate *(4b)* is employed to ascertain whether the magnitude of the sine and the cosine should be interchanged with the multiplexers *(2c)*.

3 Sublinear optimizations

The optimizations described in the previous sections have the following drawbacks:

- They require a subgenerator with an input space greater than $\lambda_t(\phi)/8$, that is, its input space grows linearly with $\lambda_t(\phi)$. In many applications, the subgenerator cannot be directly implemented using an LUT with a number of entries proportional to the input space since it is excessively large. For example, even if the octant optimization could be directly applied to the arithmetic unit mentioned in Section 1, it would only reduce the size of the input space of the required subgenerator to roughly $30 * 2^{49}$.

- They can only be directly applied if ϕ is trigonometric rational. Furthermore, the quadrant and octant optimizations require $\lambda_t(\phi)$ to be even and a multiple of 4, respectively. Hence, in order to apply them in the arithmetic unit of the example of Section 1, a workaround similar to that shown in [1, 14] is necessary. For example, assuming the angle is positive, the arithmetic unit could execute the following steps to compute the sine and cosine:

  1. Divide the angle by 2π. This can be implemented efficiently by multiplying the angle by the precomputed value of the reciprocal of 2π.
  2. Take the fixed-point representation I of the fractional part of the previous division.
  3. Return the sine and cosine of $n\phi$, where n is the number represented by I in unsigned integer notation, $\phi = \frac{2\pi}{2^w}$, and w is the width of I.

  This last step can be carried out by a generator that can be implemented using the optimizations described in the previous section. However, this approach has its own drawbacks: first, the cost of a multiplication is introduced; second, the reciprocal of 2π is not rational, and hence, its exact representation cannot be stored. Moreover, even if we could use an exact representation of the reciprocal of 2π, the exact product of the angle by the reciprocal is not rational unless the angle is zero, that is, the exact result of the division of the angle by 2π is not rational. Hence, its representation cannot be exact and an error is introduced [14].

In the following subsections, optimizations without these drawbacks are described.

3.1 Branching

This optimization, used in [1], accepts an arbitrary value of ϕ, although it was originally employed for a trigonometric rational value. When branching is applied, the generator is implemented using two subgenerators, M_1 and M_0, which we call *branches*. The inputs of the branches are denoted by $A(1)$ and $A(0)$, while the widths of these inputs are denoted by $L(1)$ and $L(0)$, respectively. These are chosen so that the width of the input of the generator, I, is $w = L(1) + L(0)$. M_1 provides the sine and cosine of integer multiples of ϕ_1, that is, the sine and cosine of $n_1\phi_1$, where n_1 is the value represented by $A(1)$ and $\phi_1 = 2^{L(0)}\phi$. On the other hand, M_0 provides the sine and cosine of $n_0\phi_0$, where n_0 is the value represented by $A(0)$ and $\phi_0 = \phi$. The least significant bits of I are connected to $A(0)$, while the rest are connected to $A(1)$. Since I is the concatenation of $A(1)$ and $A(0)$, the value represented by I is

$$n = n_1 2^{L(0)} + n_0 \tag{4}$$

and hence,

$$n\phi = n_1 2^{L(0)}\phi + n_0\phi = n_1\phi_1 + n_0\phi_0 \tag{5}$$

Since the sines and cosines of $n_1\phi_1$ and $n_0\phi_0$ are provided by M_1 and M_0, the sine and cosine of their sum can be computed by applying the following trigonometric identities:

$$\begin{aligned} \sin(A+B) &= \sin(A)\cos(B) + \cos(A)\sin(B) \\ \cos(A+B) &= \cos(A)\cos(B) - \sin(A)\sin(B) \end{aligned} \tag{6}$$

Alternatively, we can say that each subgenerator M_k provides the complex $\sin(n_k\phi_k)i + \cos(n_k\phi_k) = e^{n_k\phi_k i}$, and the generator can provide the value $e^{n\phi i}$ by computing the complex product $e^{n_1\phi_1 i}e^{n_0\phi_0 i}$. Indeed, computing the product of two complexes, each of a unitary module, is equivalent to computing the sine and cosine of the sum of two angles from the sine and cosine of those angles and implies four real products, a real sum, and a real subtraction. A generalization of this branching technique was proposed in [15] to compute twiddle factors. Note that the sum of the sizes of the input spaces of M_1 and M_0 is minimum when $L(1)$ and $L(0)$ differ by no more than 1. In this case, such a sum grows with the square root of the size of the original input space, that is, sublinearly [15]. Accordingly to [1], floating-point sine/cosine applications will benefit from a fixed-point conversion of the datapath around these functions.

3.2 Tree generator

The implementation of the branches was not detailed in the previous subsection. In the generator described in [1], the branch M_0 computes its output using the Taylor series, while M_1 is implemented with an LUT of affordable size. Further optimization could be achieved if one or both branches were, in turn, implemented with sub-branches. This recursive application of the branch optimization is used by the tree generator described in [16]. In general, the tree generator requires a set of subgenerators that we will call *leaves*, complex multipliers, and, if the implementation is sequential or pipelined, registers. The following notation is employed for its description:

- w: width of the input of the tree generator
- $I = I_{w-1}I_{w-2}\dots I_1I_0$: input of the tree generator
- $n = \sum_{t=0}^{w-1} I_t 2^t$: number represented by the input of the tree generator
- m: number of leaf subgenerators employed
- $M_0, M_1, \dots, M_{m-1}$: the m leaves
- $L(k)$: width of the input of the leaf M_k
- $A(k) = A(k)_{L(k)-1}\dots A(k)_0$: input of the leaf M_k
- $n_k = \sum_{t=0}^{L(k)-1} A(k)_t 2^t$: number represented by the input A(k)
- $SL(k) = \sum_{t=0}^{k-1} L(t) = \begin{cases} 0 \text{ if } k = 0 \\ L(k-1) + SL(k-1) \text{ if } k > 0 \end{cases}$: total number of input lines of the leaves with an index lower than k
- ϕ_k: angle defined by $\phi_k = (2^{SL(k)})\phi$

Each leaf subgenerator M_k provides the sine and cosine of $n_k\phi_k$. The leaves are chosen such that the sum of the widths of their inputs is equal to the width of the input of the tree generator:

$$w = SL(m) = \sum_{k=0}^{m-1} L(k) \tag{7}$$

The input lines of each leaf M_k are connected to the input lines of the tree generator from $I_{SL(k)}$ to $I_{SL(k+1)-1}$, that is, each input line $A(k)_t$ is connected to $I_{t+SL(k)}$:

$$A(0) = I_{L(0)-1}\dots I_1I_0$$

$$A(1) = I_{L(0)+L(1)-1}\dots I_{L(0)+1}I_{L(0)}$$

$$\vdots$$

$$A(m-1) = I_{w-1} \dots I_{SL(m-1)+1} I_{SL(m-1)}$$

Hence, the input value n represented by I becomes:

$$\begin{aligned} n = \sum_{t=0}^{w-1} I_t 2^t = \sum_{k=0}^{m-1} \sum_{t=SL(k)}^{SL(k)+L(k)-1} I_t 2^t = \\ \sum_{k=0}^{m-1} \sum_{t=0}^{L(k)-1} I_{t+SL(k)} 2^{t+SL(k)} = \\ \sum_{k=0}^{m-1} \sum_{t=0}^{L(k)-1} A(k)_t 2^{t+SL(k)} = \\ \sum_{k=0}^{m-1} \Big(\sum_{t=0}^{L(k)-1} A(k)_t 2^t \Big) 2^{SL(k)} = \sum_{k=0}^{m-1} n_k 2^{SL(k)} \end{aligned} \tag{8}$$

and therefore the angle whose sine and cosine must be computed by the tree generator can be written as:

$$n\phi = \sum_{k=0}^{m-1} n_k 2^{SL(k)} \phi = \sum_{k=0}^{m-1} n_k \phi_k \tag{9}$$

Hence, the angle $n\phi$ is the sum of the subangles $n_k\phi_k$, or, alternatively, the complex $e^{n\phi i}$ is the product $e^{n_0\phi_0 i} e^{n_1\phi_1 i} \dots e^{n_{m-1}\phi_{m-1} i}$. Again, since the sine and cosine of the subangles are provided by the leaves, the sine and cosine of $n\phi$ can be computed with complex multiplications. Taking this into account, the structure of the generator described in [16] becomes a directed rooted binary tree with m leaves. Each vertex corresponds to a component whose output is a complex of unitary module. Each internal vertex has exactly two children and corresponds to a complex multiplier that computes the product of the outputs of the components associated to these children. The components corresponding to the m leaves are the m subgenerators and provide the complex values $e^{n_k\phi_k i}$. The output of the tree generator is the output of the component corresponding to the root vertex. Hereinafter, the height of the tree will be denoted as h. The following recommendations may improve the efficiency of the design:

- It is desirable to minimize the height of the tree h in order to reduce latency and rounding errors. This is achieved if the structure of the generator is a complete binary tree.
- If each leaf is implemented with an LUT, the total number of entries is minimum when the width of the inputs of those LUTs differ by no more than 1. To this end, let q be the quotient obtained by dividing w by m, and let r be the remainder. A total of r LUTs must have inputs of width $q+1$. The other LUTs must have inputs of width q.
- If the above recommendation is followed, then the total number of entries decreases when m increases. For a fixed height h, the maximum possible value of m is 2^h, and therefore, the total number of entries can be minimized by using 2^h leaves.

In order to ascertain the power of this approach, suppose we use a complete binary tree with height $h = \lfloor log_2(w) \rfloor$. In this case, the number of subgenerators m would be no greater than w, and each subgenerator would have no more than 2 input lines. If each

subgenerator is implemented with an LUT, then an upper bound on the total number of entries is $4w$, that is, the total number of entries grows logarithmically with the size of the input space of the tree generator. Hence, the implementation of a sine/cosine generator with an input space as large as that required by the arithmetic unit mentioned in Section 1 is feasible with a tree generator. Note that a tree generator can be combined with the argument reduction mentioned in Section 2. For example, in [1], argument reduction is first applied, and hence, only a subgenerator with an input space of roughly $1/8$ of the original size is required. That subgenerator is then implemented with a tree generator of height $h = 1$.

In the following sub-subsections, we will see several optimizations that can be applied to the tree generator. In the rest of the paper, $\phi > 0$ is assumed for the sake of simplicity, although in practice this is not a restriction since $\cos(n\phi) = \cos(n|\phi|)$ and $\sin(n\phi) = \mathrm{sgn}(\phi) * \sin(n|\phi|)$.

3.2.1 Quadrant restriction

This optimization can be applied to *quadrant restricted* sine/cosine generators, which are defined as follows:

Definition 5 *Given a functional unit with an integer input $n \geq 0$ that computes one or more trigonometric functions of $n\phi$, where $\phi > 0$ is a constant, the unit is quadrant restricted if and only if $\frac{\pi}{2\phi}$ is an upper bound on its input space.*

If a sine/cosine generator is quadrant restricted, then it must compute the sine and cosine of an angle $n\phi$ in the interval $[0, \pi/2]$. Since both functions are positive in that interval, the following optimizations are possible:

- As in the example of Section 2, if the generator is implemented with an LUT, there is no need to store the sign bits.
- If it is implemented with a tree generator, no signed adders, subtracters, nor multipliers are required.

For example, the tree generator used in [1] is quadrant restricted, and hence, the complex multiplier requires no signed arithmetic components and the LUT employed does not need to store the sign bits. Note that even if a tree generator is not quadrant restricted, it may contain quadrant restricted branches that can benefit from these optimizations.

3.2.2 Leading zeros of the sine

This optimization is useful when the sine values of a quadrant-restricted generator are coded in fixed-point. In this case, an upper bound on the sine output is $\sin(n_{\max}\phi)$, where $n_{\max}$ is the maximum of the input space. Consequently, if the k most significant bits of the fixed-point representation of $\sin(n_{\max}\phi)$ are 0, those bits of the sine output of the generator are always 0, and the following optimizations are possible:

- If the generator is implemented with an LUT, then there is no need to store the k most significant bits of the sine.
- If the generator feeds a complex multiplier of a tree generator, the size of its real multipliers can be reduced.

3.2.3 Leading ones of the cosine

This optimization is useful when the cosine values of a quadrant-restricted generator are coded in fixed-point using all the bits for the fractional part. In this case, the representable value nearest to $\cos(0) = 1$ corresponds to the word with all the bits equal to 1. A lower bound on the cosine output is $\cos(n_{\max}\phi)$, where $n_{\max}$ is the maximum of the input space, and therefore, if the k most significant bits of the fixed-point representation of $\cos(n_{\max}\phi)$ are 1, those bits of the cosine output of the generator are always 1. Hence, if the generator is implemented with an LUT, there is no need to store the k most significant bits of the cosine.

To exemplify these optimizations, suppose a generator must provide the sine and cosine of $n\phi$ in fixed-point notation with 8 fractional bits and no integer bits rounding to the nearest representable value. In this example, $\phi = 2\pi/2^{11} = \pi/2^{10}$, that is, it is trigonometric binary and $\lambda_t(\phi) = 2^{11}$ is a multiple of 4, and hence, we first apply argument reduction and treat the case $n = 2^8$ separately as in the example of Section 2. A subgenerator still has to be subsequently implemented with an input space of size 2^8 ($w = H_t(\phi) - 3 = 8$). We implement this subgenerator with a tree generator of height $h = 1$, and therefore, there are 2 leaves ($m = 2^h = 2$) as shown in Fig. 2. Since this tree generator is quadrant restricted, it does not require signed arithmetic components. Each leaf is implemented with direct access memory *(5)* of depth 2^4. Note that the sine/cosine values must be stored with a precision of 17 bits to compensate for rounding errors. The leaf M_0 provides the sines and cosines of the multiples of $\phi_0 = 2^0\phi = \pi/2^{10}$, while the leaf M_1 provides the sines and cosines of the multiples of $\phi_1 = 2^4\phi = \pi/2^6$. The greatest angle whose sine and cosine is stored in M_0 is $15\pi/2^{10}$. The 4 most significant bits of the representation of the sine of this angle are 0 and, since the generator is quadrant restricted, the 4 most significant bits of the other representations of the sines stored in M_0 are also 0, and therefore, there is no need for them to be stored. On the other hand, the 9 most significant bits of the representation of the cosine of that angle are equal to 1, and therefore, there is no need for them to be stored either. Similar optimizations can be applied to M_1, but in this case, only one bit can be saved. Note that we only need two integer multipliers of size 13×17 *(6a)* and two of size 17×17 *(6b)* instead of four multipliers of size 17×17 thanks to the leading zeros of the sine. The leading ones of the cosine cannot be employed to reduce the size the arithmetic components in a similar way. In the last stage, an adder *(3)* provides the sine of the tree generator and a subtracter *(7)* provides the cosine.

4 Sine/complement generator

In the optimizations described in Section 3, the angle whose sine/cosine must be computed is decomposed into two subangles, A and B. Two subgenerators, called branches, are employed to compute the sine/cosine of A and B, and then the sine/cosine of $A + B$ is computed by applying the identities 6. This method presents the following drawbacks:

- If the maximum value of one of the angles A or B is small, then its sine is close to 0, while its cosine is close to 1. In this case, the product $\cos(A)\cos(B)$ can be orders of magnitude greater than $\sin(A)\sin(B)$, and hence, smearing may occur when computing $\cos(A + B) = \cos(A)\cos(B) - \sin(A)\sin(B)$.

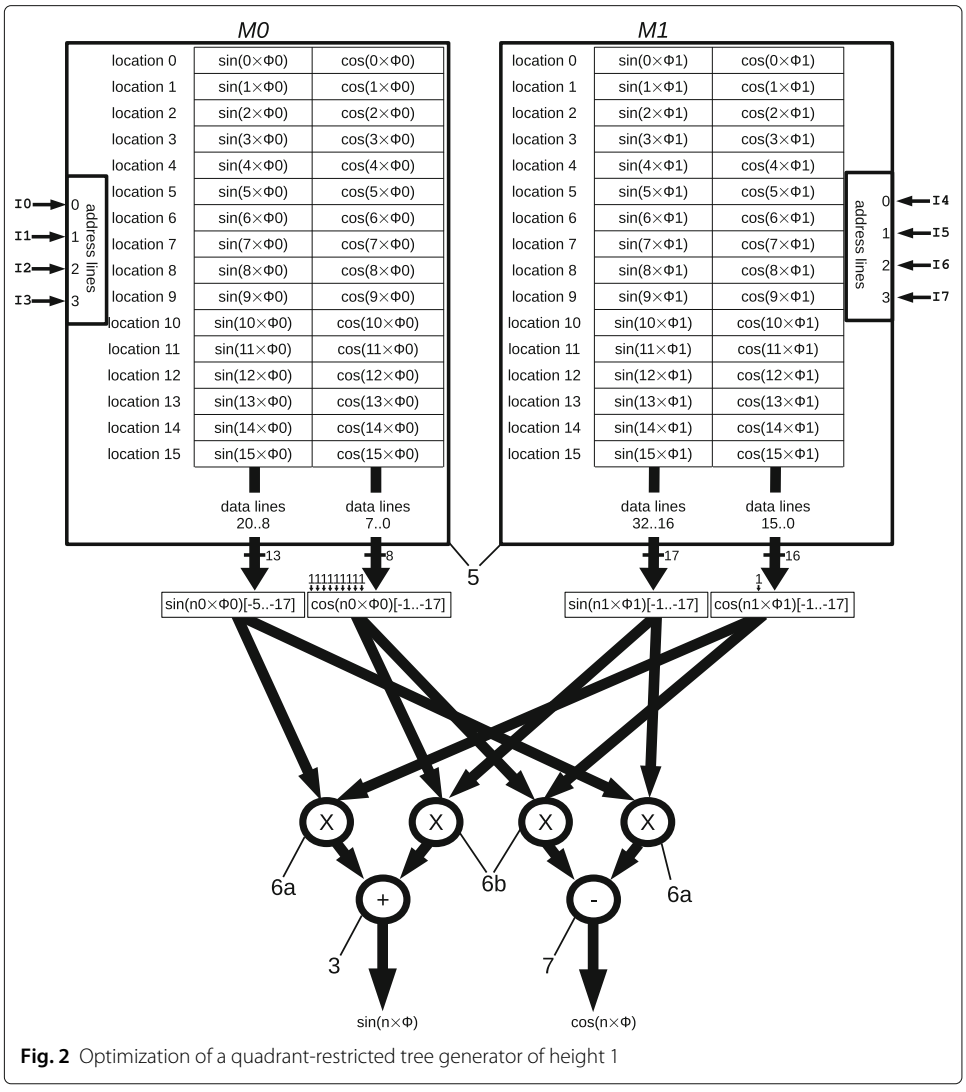

Fig. 2 Optimization of a quadrant-restricted tree generator of height 1

- Unlike the optimization described in Section 3.2.2, the one described in Section 3.2.3 fails to help in the reduction of the required arithmetic components.

These problems can be solved by using another type of generator that we call *sine/complement generator*. Such generator receives an integer n and computes the sine and the *complement of the cosine* of $n\phi$, that is defined as follows:

Definition 6 *The complement of the cosine of x is com*$(x) = 1 - \cos(x)$

It is possible to compute the sine and the complement of the cosine of the sum of two angles, A and B, from the sines and complements of the cosines of those angles by using a functional unit called a *trigonometric adder* [17]. Similar to the complex multiplier, the trigonometric adder can be implemented with adders *(3)*, subtracters *(7)*, and multipliers *(6)* as depicted in Fig. 3. This trigonometric adder implementation uses the following trigonometric identities derived from those of 6:

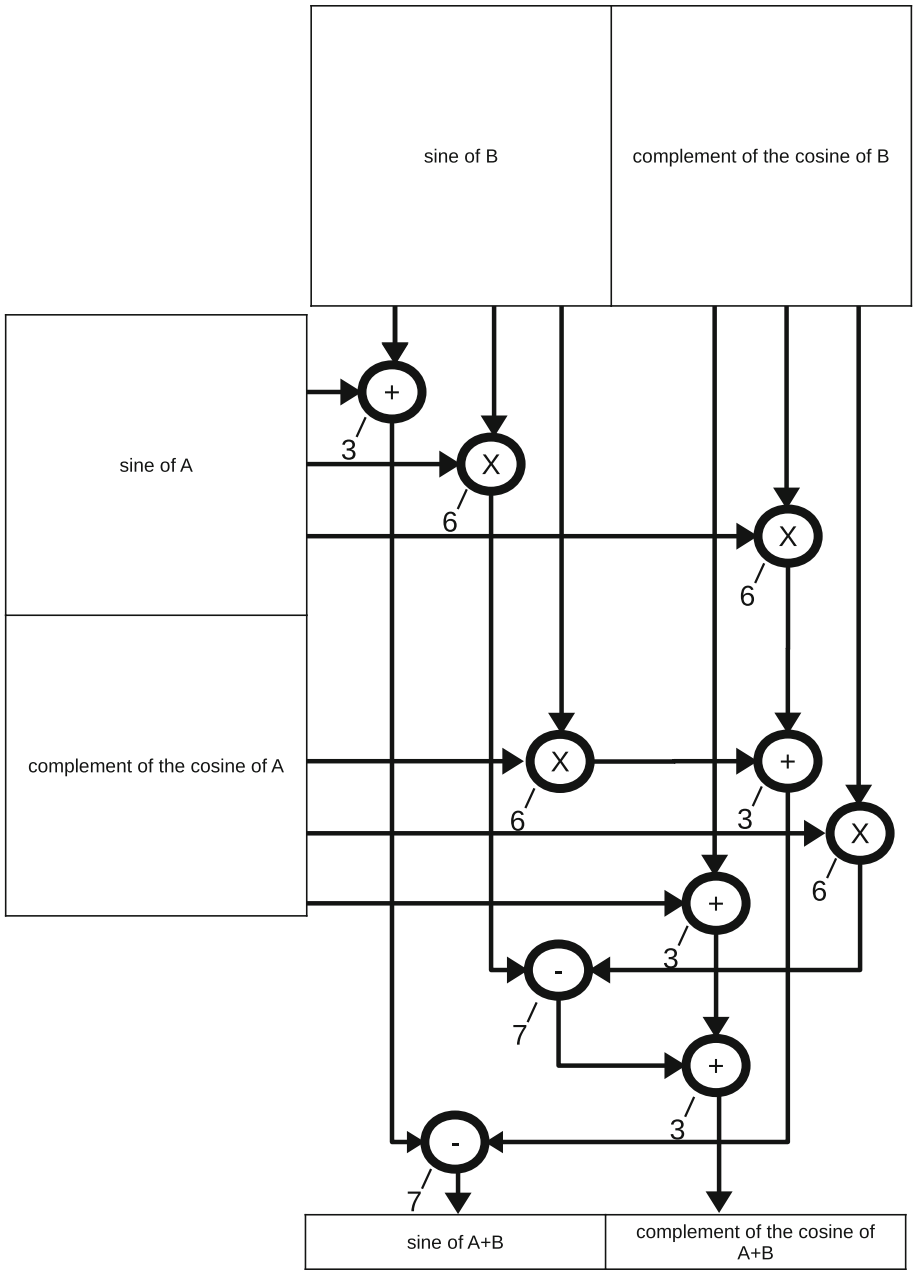

Fig. 3 Implementation of a trigonometric adder

$$\begin{aligned} \sin(A+B) = \\ \sin(A)+\sin(B)-[\sin(A)\mathrm{com}(B)+\mathrm{com}(A)\sin(B)] \\ \mathrm{com}(A+B) = \\ \mathrm{com}(A)+\mathrm{com}(B)+[\sin(A)\sin(B)-\mathrm{com}(A)\mathrm{com}(B)] \end{aligned} \tag{10}$$

Trigonometric adders enable the implementation of a sine/complement generator using a tree structure similar to that described in Section 3.2. Such an implementation, described in [18], requires a set of sine/complement subgenerators (the leaves of the tree)

as well as trigonometric adders (the internal vertex). Furthermore, if the generator is quadrant restricted, then optimizations similar to those described in Section 3.2 can be applied:

- Since the complement of the cosine is also positive in $[0, \pi/2]$, the trigonometric adders can be implemented without signed arithmetic components.
- If fixed-point representation is used, the leading zeros of the sines make it possible to reduce the size of the integer multipliers and that of the leaves.

In this case, the optimization described in Section 3.2.3 cannot be applied, but if fixed-point representation is used, we can use the following optimization that we call *leading zeros of the complement*: if a branch is quadrant restricted, an upper bound on its complement output is $\mathrm{com}(n_{\max}\phi)$, where $n_{\max}$ is the maximum of the input space of the branch. Therefore, if the k most significant bits of the fixed-point representation of $\mathrm{com}(n_{\max}\phi)$ are 0, those bits of the complement output of the branch are always 0. If the branch is implemented with an LUT, then there is no need to store those bits. Note that, unlike the optimization described in Section 3.2.3, this optimization enables a reduction of the size of the required multipliers.

A sine/cosine generator can be implemented with a sine/complement generator by simply adding a trivial arithmetic circuit to subtract the complement of the cosine from 1. In fact, if fixed-point notation is used, then this arithmetic circuit is not necessary since the trigonometric adder corresponding to the root can be easily modified to provide $\cos(n\phi)$ instead of $\mathrm{com}(n\phi)$ at no additional cost. To this end, instead of $\mathrm{com}(n\phi) = \mathrm{com}(A + B)$, the root vertex computes its opposite $-\mathrm{com}(n\phi) = -\mathrm{com}(A + B)$ by using the following equation:

$$\begin{aligned} -\mathrm{com}(A+B) = \\ [\,\mathrm{com}(A)\mathrm{com}(B) - \sin(A)\sin(B)] - [\,\mathrm{com}(A) + \mathrm{com}(B)] \end{aligned} \tag{11}$$

subsequently 1 can be added to $-\mathrm{com}(n\phi)$ in order to obtain $\cos(n\phi)$. Note that this last operation is merely toggling the integer bit of the representation of $-\mathrm{com}(n\phi)$. If a sine/cosine generator or a branch of it is quadrant restricted, then it should be implemented employing a complement generator due to the following reasons:

- The smearing problems are lessened by using the formulae 10.
- In contrast to the leading ones of the cosine optimization, the leading zeros of the complement optimization make it possible to reduce the size of the multipliers.

As an example, Fig. 4 shows how to implement a sine/cosine generator with an input I of width $w = 11$ using a sine/complement generator whose topology is a tree of height $h = 2$. The sine/complement generator uses $m = 2^h = 4$ subgenerators, M_0, M_1, M_2, and M_3, which have been implemented with direct access memories *(8)*. Following the recommendations of Section 3.2 to minimize the total number of memory locations, M_3 has 2 address lines ($q = \lfloor w/m \rfloor = 2$) and each of the other 3 remaining memories ($r = w - mq = 3$) has an additional address line, and therefore, $L(3) = 2$ and $L(2) = L(1) = L(0) = 3$. Hence, $SL(0) = 0$, $\phi_0 = 2^0\phi$, $SL(1) = 3$, $\phi_1 = 2^3\phi$, $SL(2) = 6$, $\phi_2 = 2^6\phi$, $SL(3) = 9$, and $\phi_3 = 2^9\phi$. Each memory M_k contains the sines and the complement of the cosines of the multiples of $\phi_k = (2^{SL(k)})\phi$, and therefore, its output provides the sine and the complement of the cosine of $n_k\phi_k$, where n_k is the value of its address lines. Each

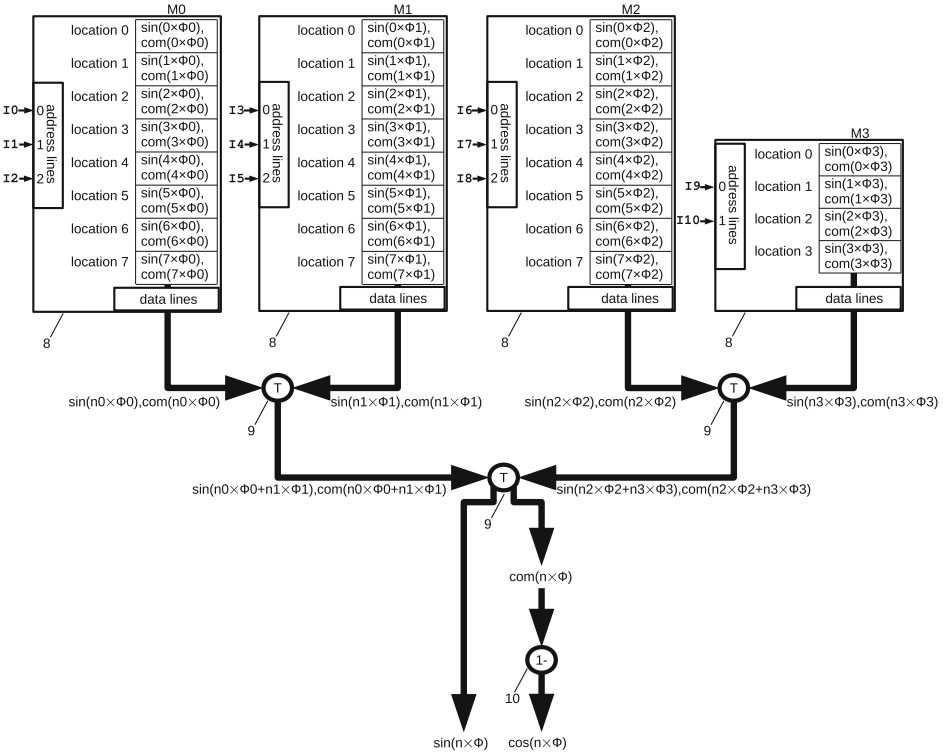

Fig. 4 Sine/cosine generator implemented with a sine/complement generator

address line t of each memory M_k is connected to $I_{t+SL(k)}$, that is, the inputs of M_0, M_1, M_2, and M_3 are connected to $I_2I_1I_0$, $I_5I_4I_3$, $I_8I_7I_6$, and $I_{10}I_9$, respectively. Hence, $n = n_0 2^{SL(0)} + n_1 2^{SL(1)} + n_2 2^{SL(2)} + n_3 2^{SL(3)} \implies n\phi = n_0 2^{SL(0)}\phi + n_1 2^{SL(1)}\phi + n_2 2^{SL(2)}\phi + n_3 2^{SL(3)}\phi = n_0\phi_0 + n_1\phi_1 + n_2\phi_2 + n_3\phi_3$. Three trigonometric adders are used *(9)*. Those connected directly to the memories are employed to compute the sine and the complement of the cosine of the angles $n_0\phi_0 + n_1\phi_1$ and $n_2\phi_2 + n_3\phi_3$. The other computes the sine and the complement of the cosine of $n\phi = n_0\phi_0 + n_1\phi_1 + n_2\phi_2 + n_3\phi_3$. A trivial arithmetic circuit *(10)* subtracts the complement of the cosine of $n\phi$ from 1 to obtain the cosine of $n\phi$.

5 Experiments

In order to measure the possible enhancements that the proposed approach may provide, we have written an open-source tool, called `twiddle.py`, to automate the design of twiddle factor generators that use the proposed optimization. The tool admits arbitrary output precision and tree height. We selected the same size for the input, the sine output, and the cosine output. In the current version, the sequence length must be a powers of 2 so it is possible to apply argument reduction and only quadrant-restricted generators are needed. The sequence lengths in our experiments ranged from 2^{16} to 2^{23}. The `twiddle.py` tool follows the recommendations of Section 3.2 to minimize the size of the memories. More information about the tool can be found in the section "Availability of data and materials." The values computed by the generators are faithfully rounded.

6 Results and discussion

6.1 Multipliers

As described in Section 3.2.3, the leading ones of the cosine optimization make it possible to reduce the size of the memories, but not the size of the required arithmetic components. We claimed that the leading zeros of the complement optimization, described in Section 4, were more convenient because they make it possible to reduce not only the size of the memories, but also the size of the multipliers. To ascertain such claim, we used the `twiddle.py` tool to implement generators of height 1 with input size ranging from 16 to 23 bits. The comparison of the size of the complement multiplicands and the corresponding cosine multiplicands is shown in Table 2. The width in bits of the cosine and the complement of the cosine is reported in the subcolumns *cosine* and *compl* respectively. The relative reduction is reported in the subcolumns *red.*

The first fractional bit of the cosine multiplicand is always zero. However, the precision of the multiplicands must be two bits higher than the output precision to compensate for rounding errors. For this reason, the size of the cosine multiplicands of both branches must be a bit greater than the output data width. The size of the corresponding complement multiplicand of the left branch is just one bit lower so the relative saving decreases with the data width. On the other hand, the size of the complement multiplicand of the right branch is almost constant and is never greater than 4 so the relative saving increases with the data width and ranges from 83 to 90%.

As noted by [1], an obvious way to further reduce the size of the memories is to replace them by sub-branches with memories of smaller depth at the cost of more multiplications. Of course, if the implementation is fully pipelined or combinational, the additional multiplications should be carried out by new multipliers. If such replacement is carried out, the angles corresponding to most of these new memories are remarkably smaller. For this reason, the leading zeros of the complement optimization should further reduce the size of the additional multipliers. To ascertain this, we used the `twiddle.py` to carry out such replacements by increasing the tree height one stage. The saving obtained for each branch is reported in the next sub-subsections.

6.1.1 Left branch

The size of the operands of the left branch is shown in Table 3. As in the tree of height 1, the saving obtained by using the proposed implementation in its left sub-branch is negligible, but the saving in its right sub-branch is remarkable, ranging from 41 to 53%.

Table 2 Multiplicand width comparison of generators with a tree structure of height 1

Data width	Left branch			Right branch		
	Cosine	Compl.	Red. (%)	Cosine	Compl.	Red. (%)
16	17	16	6	16	2	88
17	18	17	6	17	3	83
18	19	18	5	18	2	89
19	20	19	5	19	3	85
20	21	20	5	20	2	90
21	22	21	5	21	3	86
22	23	22	4	22	4	83
23	24	23	4	23	3	88

Table 3 Multiplicand width comparison of the left branch of generators with a tree structure of height 2

Data width	Left sub-branch			Right sub-branch		
	Cosine	Compl.	Red. (%)	Cosine	Compl.	Red. (%)
16	19	18	5	19	9	53
17	20	19	5	20	11	45
18	21	20	5	21	12	43
19	22	21	5	22	13	41
20	23	22	4	23	12	48
21	24	23	4	24	13	46
22	25	24	4	25	14	44
23	26	25	4	26	15	42

6.1.2 Right branch

As shown in Table 4, the saving obtained by using the proposed implementation in the right branch is astonishing. To begin, the saving corresponding to its left sub-branch is very high, ranging from 77 to 88%. Moreover, the saving corresponding to its right sub-branch is always 100%. This means that the complement multiplicand of the right sub-branch is always zero and the corresponding multipliers can be removed.

6.2 Implementation

The hardware description of the twiddle factor generators with a sine/complement tree of height 1 were implemented in a Xilinx (Virtex) 7 XC7VX485T-2FFG1761 field-programmable gate array (FPGA) chip. We also implemented the equivalent sine/cosine tree generators described in [1] and cordinate rotation digital computer (CORDIC) generators to measure the relative enhancements and penalties. Following the notation used in [1], the reference sine/cosine tree implementations will be called *SinAndCos*. The proposed implementations will be called *SinAndCom*. The *SinAndCos* and CORDIC implementations are generated by the `flopoco` open-source tool version 4.1.2 available at http://flopoco.gforge.inria.fr. The tested implementations are combinational, but are embedded in a dummy sequential module in order to obtain delay estimations from the synthesis tool. Synthesis was carried out with the (Vivado) Design Suite tool of Xilinx version 2017.2.1 using the default options. The only exception is that the use of the Digital Signal Processing (DSP) blocks was disabled in order to extrapolate the results to other

Table 4 Multiplicand width comparison of the right branch of generators with a tree structure of height 2

Data width	Left sub-branch			Right sub-branch		
	Cosine	Compl.	Red. (%)	Cosine	Compl.	Red. (%)
16	19	4	79	19	0	100
17	20	3	85	20	0	100
18	21	4	81	21	0	100
19	22	5	77	22	0	100
20	23	4	83	23	0	100
21	24	3	88	24	0	100
22	25	4	84	25	0	100
23	26	5	81	26	0	100

Table 5 Delay of the sine/complement generators

Data width	Delay (ps)
16	15026
17	17243
18	18547
19	16592
20	17183
21	18628
22	19791
23	19914

reconfigurable devices as well as to ASIC. The obtained results are reported in the next sub-subsections.

6.2.1 Delay

The delay of the slowest path of the generators is shown in Tables 5, 6, and 7. The *SinAndCom* approach turned out to be about 50% faster than CORDIC but was slower than *SinAndCom* in most cases. For the lower data widths, the penalty was as high as 26%. One of the reasons is that, in order to apply Eq. 10, it is necessary to execute more explicit additions and subtractions than in Eq. 6. The penalty is not that severe for higher widths, and there are some little speed-ups ranging from 5 to 7%.

6.2.2 Resources

The resource utilization of the generators are shown in Tables 8, 9, and 10. No DSP units were used. As mentioned in Section 3.2, in order to save resources, the `flopoco` tool does not implement the right branch of the *SinAndCom* generators with a memory. Instead, the corresponding values are evaluated by using the Taylor series. The same optimization can be applied to the *SinAndCom* approach, but the `twiddle.py` does not implement it yet. For this reason, this comparison is not fair to the *SinAndCom* approach but, even without the Taylor series optimization, *SinAndCom* used less LUTs in every case. The best saving was as high as 26%. The implementation of the Taylor optimization should provide higher relative savings. Also, as mentioned in Section 6.1, the relative saving should be remarkable if higher tree structures are used. Unfortunately, unlike `twiddle.py`, the current version of `flopoco` only supports tree structures of height 1 so we were unable to ascertain this. Regarding CORDIC, for low data widths, the *SinAndCom* has enabled the number of look-up tables to be reduced by between 20 and 27%, while for higher data widths, it introduces a penalty as high as 15% in the resource utilization.

Table 6 Delay of the CORDIC generators

Data width	Delay (ps)	SinAndCom speed-up (%)
16	28292	47
17	30660	44
18	31769	42
19	34167	51
20	35845	52
21	37474	50
22	38459	49
23	41852	52

Table 7 Delay of the sine/cosine generators

Data width	Delay (ps)	SinAndCom speed-up (%)
16	12877	− 17
17	13692	− 26
18	15433	− 20
19	16362	− 1
20	18500	7
21	19590	5
22	17862	− 11
23	19078	− 4

Table 8 Resource utilization of the sine/complement generators

Data width	Look-up tables
16	696
17	828
18	863
19	1390
20	1161
21	1711
22	1735
23	2025

Table 9 Resource utilization of the CORDIC generators

Data width	Look-up tables	SinAndCom saving (%)
16	934	25
17	1037	20
18	1175	27
19	1270	− 9
20	1362	15
21	1512	− 13
22	1657	− 5
23	1765	− 15

Table 10 Resource utilization of the sine/cosine generators

Data width	Look-up tables	SinAndCom saving (%)
16	739	6
17	1121	26
18	1040	17
19	1600	13
20	1325	12
21	1994	14
22	2238	23
23	2602	22

7 Conclusions

In this paper, we propose a table based-sine/cosine computation technique. In the proposed technique, the complement of the cosine is computed before the cosine itself in order to reduce the size of the required multiplications. We have released an open-source tool to automate the design of twiddle factor generators of arbitrary size and precision using the proposed technique. Several twiddle factor generators have been implemented in a Xilinx (Virtex) 7 XC7VX485T-2FFG1761 FPGA chip using the proposed technique and other techniques described in [1]. The proposed technique is remarkably faster than CORDIC. Also, when compared with previous table-based implementations with a tree structure of height 1, the proposed technique enabled a remarkable saving in the hardware resources at the expense of delay. To increase the relative saving, some of the memories could be replaced by circuits that evaluate the corresponding values. Further research is required to measure the benefits that such optimization could provide. Also, it would be interesting to make new comparisons if a version of `flopoco` supporting higher tree structures is released.

Abbreviations

ASIC: Application-specific integrated circuit; CORDIC: Coordinate rotation digital computer; DCT: Discrete cosine transform; DSP: Digital signal processing; DFT: Discrete fourier transform; DST: Discrete sine transform; DVB-T2: Digital video broadcasting—terrestrial 2; FFT: Fast fourier transform; FPGA: Field-programmable gate array; IDCT: Inverse discrete cosine transform; IDST: Inverse discrete sine transform; PLC: Power line communications; LUT: Look-up table

Acknowledgements

Not applicable.

Authors' contributions

Conceptualization, D.G.; data curation, D.G. and J.J; formal analysis, D.G.; funding acquisition, J.J. and P.R.; investigation, D.G.; methodology, D.G.; project administration, J.J.; resources, A.M., M.B. and E.O.; software, D.G. and A.M.; supervision, M.B.; validation, D.G.; visualization, P.R.; writing of the original draft and preparation, D.G. and J.J.; writing, review, and editing, A.M., J.V, and M.B. The authors read and approved the final manuscript.

Funding

This work has been partially supported by the Ministerio de Economía, Industria y Competitividad of Spain under project TIN2017-89951-P (BootTimeIoT) and by the European Regional Development Fund (ERDF).

Availability of data and materials

Software used to generate the Verilog descriptions of the twiddle calculators implemented and the corresponding testbenches using the complement approach:

- Project name: twiddle complement faithfull
- Project home page: https://gitlab.com/davidguerrero/twiddle-complement-faithfull
- Archived version: 1.0
- Operating system: Platform independent
- Programming language: python
- License: The software itself has a GPL license. However, the Verilog descriptions generated by this software are covered by patents WO2018104566A1, P201831134, and P201831133.

Software used to generate the VHDL descriptions of the sine/cosine generators described in [1]:

- Project name: FloPoCo
- Project home page: http://flopoco.gforge.inria.fr/
- Archived version: 4.1.2
- Operating system: GNU Linux, MacOS, Windows
- Programming language: C++
- License: still being decided

Competing interests

The authors developed the inventions covered by patents WO2018104566A1, P201831134, and P201831133.

Received: 10 September 2019 Accepted: 6 July 2020
Published online: 22 July 2020

References

1. F. de Dinechin, M. Istoan, G. Sergent, Fixed-point trigonometric functions on FPGAS. SIGARCH Comput. Archit. News. **41**(5), 83–88 (2014). https://doi.org/10.1145/2641361.2641375
2. K. J. Lin, C. C. Hou, in *Proceedings of the IEEE 2nd Global Conference on Consumer Electronics (GCCE 2013)*, Implementation of trigonometric custom functions hardware on embedded processor, (Tokyo, 2013), pp. 155–157. https://doi.org/10.1109/GCCE.2013.6664782
3. H. Huang, L. Xiao, J. Liu, Cordic-based unified architectures for computation of DCT/IDCT/DST/IDST. Circ. Syst. Signal Proc. **33**(3), 799–714 (2014). https://doi.org/10.1007/s00034-013-9661-9
4. D. Goldberg, What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991). https://doi.org/10.1145/103162.103163
5. V. Lefevre, J. M. Muller, in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, Worst cases for correct rounding of the elementary functions in double precision, (Vail, 2001), pp. 111–118. https://doi.org/10.1109/ARITH.2001.930110
6. T. Kulshreshtha, A. S. Dhar, Cordic-based high throughput sliding DFT architecture with reduced error-accumulation. Circ. Syst. Signal Proc. **37**(11), 5101–5126 (2018). https://doi.org/10.1007/s00034-018-0810-z
7. IEEE Standard for broadband over power line networks: medium access control and physical layer specifications. IEEE Std 1901-2010, 1–1586 (2010). https://doi.org/10.1109/IEEESTD.2010.5678772
8. S.-Y. Lin, C.-L. Wey, M.-D. Shieh, Low-cost FFT processor for DVB-T2 applications. IEEE Trans. Consum. Electron. **56**(4), 2072–2079 (2010). https://doi.org/10.1109/TCE.2010.5681074
9. R. H. Stanton, in *Proceedings of the 31st Annual SAS Symposium on Telescope Science*, Photon counting - one more time, (Big Bear Lake, 2012), pp. 177–184. http://adsabs.harvard.edu/abs/2012SASS...31..177S
10. H. Nakahara, H. Nakanishi, T. Sasao, On a wideband fast Fourier transform for a radio telescope. SIGARCH Comput. Archit. News. **40**(5), 46–51 (2012). https://doi.org/10.1145/2460216.2460225
11. F. Qureshi, O. Gustafsson, in *Proceedings of the 2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, Analysis of twiddle factor memory complexity of radix-2i pipelined FFTs, (Pacific Grove, 2009), pp. 217–220. https://doi.org/10.1109/ACSSC.2009.5470121
12. J. G. Nash, Distributed-memory-based FFT architecture and FPGA implementations. Electronics. **7**(7) (2018). https://doi.org/10.3390/electronics7070116
13. J. W. Cooley, P. A. W. Lewis, P. D. Welch, Historical notes on the fast Fourier transform. Proc. IEEE. **55**(10), 1675–1677 (1967). https://doi.org/10.1109/PROC.1967.5959
14. R. A. Smith, A continued-fraction analysis of trigonometric argument reduction. IEEE Trans. Bus. Econ. **44**(11), 1348–1351 (1995). https://doi.org/10.1109/12.475133
15. H. Kang, B. Yang, J. Lee, Low complexity twiddle factor multiplication with ROM partitioning in FFT processor. Electron. Lett. **49**(9), 589–591 (2013). https://doi.org/10.1049/el.2013.0689
16. D. Guerrero, J. Viejo, P. Ruiz-de-Clavijo, J. Juan, M. J. Bellido, A. Millan, E. Ostua, J. I. Villar, J. Quiros, A. Muñoz, Digital Electronic circuit for calculating sines and cosines of multiples of an angle. **WO2018104566A1** (2018)
17. D. Guerrero, A. Millan, J. Juan, J. Viejo, M. J. Bellido, P. Ruiz-de-Clavijo, E. Ostua, Dispositivo Electrónico Calculador de Funciones Trigonométricas. **P201831134** (2019)
18. D. Guerrero, A. Millan, J. Juan, J. Viejo, M. J. Bellido, P. Ruiz-de-Clavijo, E. Ostua, Dispositivo Electrónico Calculador de Funciones Trigonométricas Y Usos Del mismo. **P201831133** (2019)

Publisher's Note