New Insights on Ling Adders*

Alvaro Vázquez and Elisardo Antelo
Dept. of Electronic and Computer Science
University of Santiago de Compostela. SPAIN
alvaro@dec.usc.es, elisardo@dec.usc.es

Abstract

Adders are critical for microprocessor design. Current designs use variations of parallel prefix schemes. A method introduced by Ling [7] may improve this kind of adders. However, as recent research publications demonstrate, the use of the Ling scheme in prefix adders is not a mature and clear concept. In this work we show how to easily extend any existing prefix adder topology to use the Ling method. Moreover, we use this methodology to implement the Ling scheme in a flagged prefix adder, which is an interesting building block for floating point units.

1. Introduction

Adders along with multipliers are the basic modules for datapaths in microprocessors and digital data processing devices. The adder design space has been extensively studied by both industry and academia. For a representative set of references see [4] (chapter 2) and [14]. There exist many instances of adders, namely carry ripple, carry skip, carry select, conditional sum, carry look-ahead, prefix (Kogge-Stone, Han-Carlson, Ladner-Fisher, Brent-Kung, Knowles adders,..) and Ling adders among others. In this paper we deal with Ling adders in connection with prefix adders.

In 1981 Ling [7] proposed a new carry recurrence to reduce the logical depth for carry computation in carry look-ahead structures. Due to its own carry definition (called pseudo-carry by some authors), the theory behind this adder is considered somehow different than the other adder schemes with conventional carry definitions [3, 12, 13]. For instance, not up to very recently, different research works [2, 5, 11, 15] proposed the formulation of the Ling recurrence as a prefix computation to obtain high performance parallel adder implementations[1]. In fact, the results of these works reveal that the use of the Ling recurrence as a prefix computation leads to slightly faster adders than existing high performance implementations (around 10% speed improvement). Although this is a small improvement, adders are usually in critical paths in microprocessors with aggressive frequencies, so that such improvements are considered in fact significant. Moreover, variations of the Ling method have been used in industrial designs [9].

These results demonstrate that it is worth a deep understanding of the theory behind the Ling approach. The above mentioned recently published research works do not introduce enough insight in the theory of using the Ling recurrence for prefix computations. This statement will be justified further in Section 2. In this work we present a novel view of the Ling approach. Specifically, we obtain the Ling scheme directly from a standard prefix formulation of the carry computation, and show that any prefix adder can be transformed into a Ling adder with minor modifications and with the corresponding speed improvement. We use the same methodology to extend a flagged prefix adder [1] with the Ling scheme.

The structure of this work is as follows. Section 2 presents recent developments regarding the use of the Ling recurrence for prefix computations. Section 3 shows how the Ling method can be derived from a standard prefix formulation. Section 4 presents a simple method to transform a standard prefix adder into a Ling adder. Section 5 considers the special case of sparse carry-tree adders. Section 6 shows how to use the Ling scheme in a flagged prefix adder for speed improvement, and finally Section 7 is devoted to the conclusions.

2. Recent Work on Prefix Adders Using Ling Recurrence

In this section we introduce the notation, and present the Ling recurrence, and its use in prefix adders. Let $X = \sum_{i=1}^{n} x_i 2^{i-1}$ and $Y = \sum_{i=1}^{n} y_i 2^{i-1}$ the operands to be added so that $S = X + Y = \sum_{i=1}^{n+1} s_i 2^{i-1}$. Intermediate carries are denoted by c_i (i.e. the carry to bit position

*This work was supported in part by the Ministry of Education and Science of Spain under contract TIN 2007-67537-C03. The authors would also like to thank IBM for their support.

[1]Probably this was already known by some industrial designers.

i) and to simplify the presentation we express the carry in (c_{in}) as a generated carry from a phantom position 0 (i.e. $c_1 = c_{in}$). The conventional carry recurrence is described as[2]

$$c_{i+1} = g_i + a_i\, c_i \tag{1}$$

where $g_i = x_i\, y_i$ is the generate signal and $a_i = x_i + y_i$ is the alive signal. Note that we assume $g_0 = c_{in}$ and $a_0 = 0$.

It is well-known that the prefix operator (denoted by $\bullet$ in this work) allows the computation of a carry into a bit position only in terms of the preceding bits of the input operands. Specifically the carry into position $i+1$ is computed using the following product

$$(c_{i+1}, A_{i,0}) = \prod_{j=0}^{i} (g_j, a_j) \tag{2}$$

where $A_{i,0} = a_i....a_0$ (this is always 0 since $a_0 = 0$ and its final value is not computed in the implementation), and the product uses the $\bullet$ operator, defined as

$$(g_{j+1}, a_{j+1}) \bullet (g_j, a_j) = (g_{j+1} + a_{j+1}\, g_j, a_{j+1}\, a_j)$$

This operator is associative and idempotent, so that highly parallel tree-like structures can be used for the computation of the carries. During the carry computations (evaluation of expression (2)), intermediate results are called block generate and block alive. That is, for two arbitrary positions i and k the block generate signal ($G_{i,k}$) and block alive signal ($A_{i,k}$) are

$$(G_{i,k}, A_{i,k}) = \prod_{j=k}^{i} (g_j, a_j)$$

The block generate and alive signals allow the computation of the output carry from a block in terms of its input carry, that is $c_{i+1} = G_{i,k} + A_{i,k} c_k$. The final sum bits are computed as

$$s_{i+1} = p_{i+1} \oplus c_{i+1} \tag{3}$$

where $p_{i+1} = x_{i+1} \oplus y_{i+1}$.

Ling [7] used an alternative carry, h_{i+1}, instead of c_{i+1}. The recurrence for the Ling carries is

$$h_{i+1} = g_i + a_{i-1} h_i \tag{4}$$

The relation between the two kind of carries is the following [3] (obtained from (1) and (4) and using the fact that $g_i\, a_i = (x_i y_i)(x_i + y_i) = x_i y_i = g_i$)

$$c_{i+1} = a_i\, h_{i+1} \tag{5}$$

or alternatively,

$$h_{i+1} = c_{i+1} + c_i \tag{6}$$

[2]We represent the logical *xor* by $\oplus$, the logical *or* by "+" and the logical *and* by an empty space between signals.

The final sum bits are obtained as

$$s_{i+1} = p_{i+1}\overline{h_{i+1}} + (p_{i+1} \oplus a_i)\, h_{i+1} \tag{7}$$

As before, block generate and block alive signals (we refer to them as $HG_{i,k}$ and $HA_{i,k}$ respectively) can be defined for this carry recurrence. Ling used the recurrence to obtain the generate and alive for 4-bit blocks and showed that its logic implementation, directly from the inputs, is significantly less complex (in logical depth and hardware) than the corresponding standard formulation. Although the computation of the sum bits seems to be more complex (equation (7) vs. (3)), a carry-select structure requires only one 2:1 multiplexer after the computation of h_{i+1}, which is of similar delay as the xor used in the conventional approach. This fact leaded to faster carry look-ahead adders that used blocks of 4 bits as building blocks.

Recently the prefix operator has been used to obtain the Ling carries ([2, 5, 11, 15] among others), resulting in fast parallel tree-like implementations. In [5, 11] they claim that Ling carries follow the associative and indempotency properties using the standard prefix operator. However to use the operator they first obtain for each j position the block signals $HG_{j,j-3}$ and $HA_{j,j-3}$ (as Ling did) and then use the prefix operator to combine then to obtain the h_i values. No demonstration is provided. In [2] a different approach is presented. For each bit position they obtain new defined signals $G^*_i = g_i + g_{i-1}$ and $P^*_i = a_i\, a_{i-1}$. Then Ling carries are computed using the prefix operator as follows:

$$h_{i+1} = (G^*_i, P^*_{i-1}) \bullet (G^*_{i-2}, P^*_{i-3}) \bullet ....$$

That is, for i even (odd) they use only the even (odd) G^* terms and the odd (even) P^* terms. They claim that the resultant adders have reduced fanout requirements in comparison to conventional prefix adders. From their discussion one concludes that the formulation of the Ling adders as a prefix computation leads to structures with different properties than the corresponding conventional prefix adders. In [15] several parallel prefix adder topologies are optimized for power-performance. They claim that all of the analyzed adders implement the Ling recurrence. However they do not provide details on how the prefix computation is preformed using the Ling carries.

Moreover, although Ling adders are specially attractive for sparse-tree adder topologies (see Section 5), recent industrial implementations with aggressive power-performance requirements do not take into account this kind of optimization (see for instance [8] among others).

We conclude that the use of the Ling scheme in high-performance parallel prefix adders, although very important, it is not a mature concept with a clear and simple theory background. In the next section we present our view of parallel prefix Ling adders. Our aim is to present Ling adders as a simple optimization of the prefix computation.

3. Derivation of the Ling Method from a Prefix Formulation

We present Ling adders as a transformation that can be applied to any adder formulated in terms of the prefix operator. We use the following identity for each of the (g_j, a_j) pairs in the standard carry formulation

$$(g_j, a_j) = (0 + g_j\ a_j, a_j\ 1) = (0, a_j) \bullet (g_j, 1)$$

This is based on the property $g_j\ a_j = g_j$, which is the basis for the Ling approach. Therefore, the expression for carry computation results in

$$(c_{i+1}, A_{i,0}) = \prod_{j=0}^{i} (0, a_j) \bullet (g_j, 1) \tag{8}$$

After combining the terms $(0, a_{j-1})$ with its left hand side term $(g_j, 1)$ we obtain

$$(c_{i+1}, A_{i,0}) = (0, a_i) \bullet \prod_{j=0}^{i} (g_j, a_{j-1}) \tag{9}$$

with $a_{-1} = 0$. We define a "special carry" c'_{i+1} as

$$(c'_{i+1}, A_{i-1,0}) = \prod_{j=0}^{i} (g_j, a_{j-1}) \tag{10}$$

Therefore,

$$(c_{i+1}, A_{i,0}) = (0, a_i) \bullet (c'_{i+1}, A_{i-1,0})$$

From the definition of c'_{i+1} (equation (10)) we obtain

$$(c'_{i+1}, A_{i-1,0}) = (g_i, a_{i-1}) \bullet (c'_i, A_{i-2,0})$$

that is $c'_{i+1} = g_i + a_{i-1}c'_i$, which corresponds to the definition of the Ling carry (see (4)). Therefore, $h_{i+1} = c'_{i+1}$, and then the Ling carry is expressed as a prefix computation according to expression (10).

The evaluation of the prefix operation over two consecutive terms of the product given in (10) results in

$$(g_j, a_{j-1}) \bullet (g_{j-1}, a_{j-2}) = (g_j + g_{j-1}, a_{j-1}\ a_{j-2})$$

where, again, we used the fact that $a_{j-1}\ g_{j-1} = g_{j-1}$. Then, it is convenient to define a "simplified" prefix operator $\circ$ as $(g_l, a_l) \circ (g_r, a_r) = (g_l + g_r, a_l\ a_r)$. Therefore, the evaluation of $(h_{i+1}, A_{i-1,0})$ (Ling carries) is simplified as follows:

- For i odd (even number of terms):

$$\prod_{j=0}^{\lfloor i/2 \rfloor} (g_{2j}, a_{2j-1}) \circ (g_{2j-1}, a_{2j-2}) \tag{11}$$

- For i even (odd number of terms) pairing from the left

$$\left[\prod_{j=1}^{\lfloor i/2 \rfloor} (g_{2j}, a_{2j-1}) \circ (g_{2j-1}, a_{2j-2}) \right] \bullet (g_0, 0) \tag{12}$$

- For i even (odd number of terms) pairing from the right

$$(g_i, a_{i-1}) \bullet \prod_{j=0}^{\lfloor i/2 \rfloor - 1} (g_{2j+1}, a_{2j}) \circ (g_{2j}, a_{2j-1}) \tag{13}$$

Thus, when one of the variants shown in (11−13) are implemented for each i in a tree-like structure, the first level can be simplified in comparison to the conventional approach ($\circ$ instead of the $\bullet$ operator). For 2-ary trees the first level corresponds directly to the implementation of the $\circ$ operator, requiring one gate level instead of the two required for the $\bullet$ operator. For r-ary trees ($r > 2$) the first level involves both $\circ$ and $\bullet$ operators (Note that the $\circ$ operator is not associative with $\bullet$), and its implementation is simplified in comparison to an implementation using only $\bullet$ operators (the case of 4-ary trees corresponds to the simplification proposed by Ling). Optimized structures can be obtained by merging the generation of g and a signals with the first level of the carry-tree, using directly the adder inputs.

4. Transformation of a Prefix Adder into a Ling Adder

In this section we present a simple method to transform a prefix adder specification into a Ling adder. From the previous section we conclude that the introduction of Ling carries leads to the following changes in a conventional adder:

- The prefix operations are performed over pairs (g_j, a_{j-1}) instead of (g_j, a_j).
- The prefix operations at the first level of the tree-like computation are simplified since the $\circ$ operator is used.
- The sum bit is obtained as indicated by (7).

Figure 1 shows an example of the transformation process for 8-bit adders. The prefix adder shown in the figure is a variant of the (211) family of adders proposed in [6]. The resultant Ling adder has the same topology but with different logic in the first and last stages. The potential advantage in speed comes from the implementation of the first stage of the carry-tree, which requires a single gate level (nand gate) instead of a complex gate (or-and-invert). Although the last stage of the Ling adder is more complex, the critical path corresponds to a two input multiplexer which has a delay comparable to a two input xor gate. The same procedure can be applied to any of the variants of prefix adder topologies.

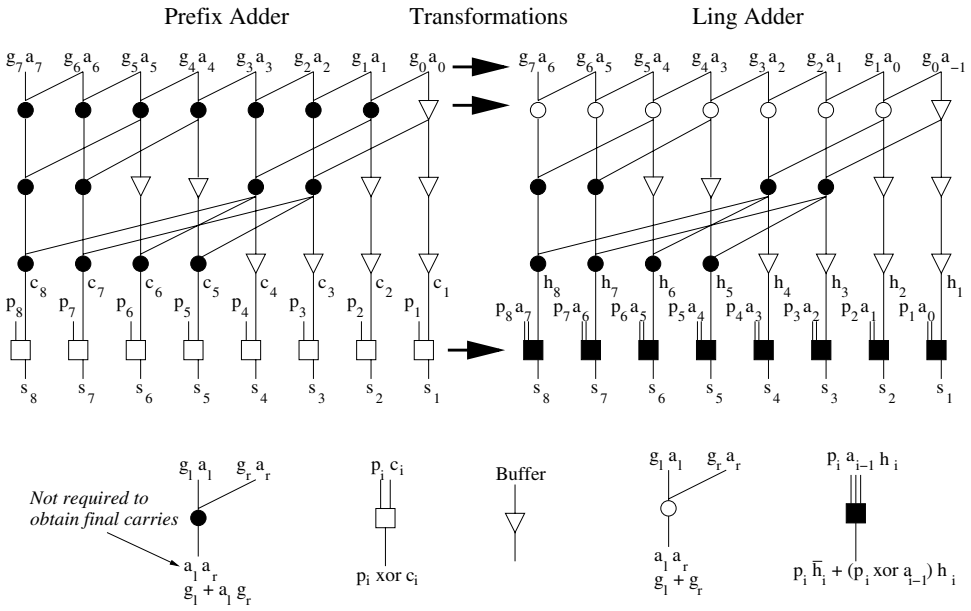


Figure 1. Transformation of a prefix adder into a Ling adder.

Regarding the hardware complexity, the Ling adder reduces the complexity of the first stage of the tree, but requires an additional xor operation per bit (assuming that an xor and a 2:1 multiplexer have similar complexity). We estimate an increase in hardware complexity for the Ling adder of about 3 equivalent nand gates per bit.

5. Sparse Carry-Tree Prefix Adders

Sparse carry trees generate one carry for every k bits. In parallel to the carry computation, two conditional sums are computed (with input carry zero and one respectively) for each group of k bits. Then the computed carries select the correct result. These kind of adders have interesting characteristics in terms of low power consumption [8].

For a group of k bits that begins at position i, we call $S0_{k,i}$ ($S1_{k,i}$) to the addition assuming a logical zero (one) carry into this group. Once the carry c_i is determined, the result is obtained as

$$S_{k,i} = \overline{c}_i \; S0_{k,i} + c_i \; S1_{k,i}$$

The implementation of sparse carry-tree prefix Ling adders is straightforward [5]. From the specification of the sparse tree of the corresponding standard prefix adder, the Ling sparse tree proposed in [5] is obtained following the rules stated in Section 4. The conditional addition is computed as

$$\begin{aligned} S_{k,i} &= \overline{h_i \; a_{i-1}} \; S0_{k,i} + h_i \; a_{i-1} \; S1_{k,i} \\ &= \overline{h}_i \; S0_{k,i} + h_i \; (S0_{k,i} \; \overline{a}_{i-1} + S1_{k,i} \; a_{i-1}) \end{aligned}$$

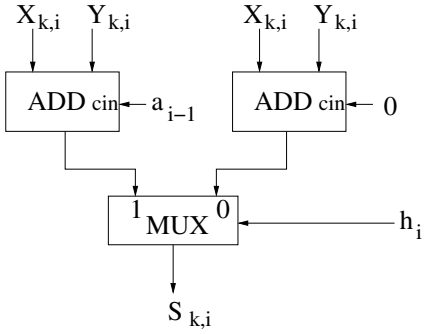


Figure 2. Conditional sum for final sum computation in a sparse tree Ling adder.

The term $(S0_{k,i} \; \overline{a}_{i-1} + S1_{k,i} \; a_{i-1})$ is efficiently computed using as carry input a_{i-1} instead of the logical one in the corresponding conditional adder. The resultant structure for final sum computation is shown in Figure 2 (for a group of k bits). The hardware complexity of this part of the adder should be similar in both cases. Therefore, we do not expect any increase in hardware complexity due to the Ling transformation. In contrast, the implementations for sparse carry–tree Ling adders presented in [2] require between 5-10% more hardware than the corresponding standard sparse carry–tree adders. This increase in hardware comes from the more complex carry-select block due to their formulation of the Ling adder.

6. Transformation of a Flagged Prefix Adder into a Flagged Ling Adder

Fast parallel evaluation of two related additions or subtractions is demanded to improve key arithmetic operations such as modulo $2^n - 1$ addition, signed magnitude addition, absolute value calculation and also to merge rounding with significand addition in IEEE floating-point computations. Some current floating-point units use a compound adder to perform this dual addition. A binary compound adder is a 2's complement adder modified to increment a fixed amount the result of a sum in an additional little constant time [10].

A high performance implementation of a compound adder is the flagged prefix adder proposed in [1]. It supports dual additions ($X+Y$ and $X+Y+1$) or subtractions ($X-Y$ and $Y-X$) with a slight increment of the hardware complexity and the delay of a prefix tree adder. In a flagged prefix tree, the flagged bit i indicates that the possible carry generated after incrementing X+Y one ulp (unit in the last place), is propagated from lsb (least significant bit) to position i. By definition, it is equal to the carry propagate group $P_{i-1,0}$ given by

$$P_{i-1,0} = \prod_{j=0}^{i-1} p_j = \overline{c_i}\, A_{i-1,0} \qquad (14)$$

where $c_i = G_{i-1,0}$ and $\prod$ means the logic AND operation. The carry alive groups $A_{i-1,0}$ can be computed in parallel to the carries c_i in a prefix tree with a final level of full logic prefix cells. Inverting the bits of $X+Y$ that have a flag bit one produces $X+Y+1$. This adder is easily improved to perform absolute differences and sign magnitude addition. For subtractions, the bits of the subtrahend are inverted and added to the minuend obtaining $X+\overline{Y} = X-Y-1$. If this addition is positive, the absolute difference is obtained by incrementing this sum one ulp (**inc** =1) as $(X+\overline{Y})+1 = X-Y$. If it is negative, the absolute difference is obtained inverting the sum bits, that is, $\overline{X+\overline{Y}} = Y-X$. The resultant sum bits are computed as

$$s_i = (p_i \oplus c_i \oplus cmp)\,\overline{inc} + (p_i \oplus (c_i + A_{i-1,0}))\; inc \quad (15)$$

where **inc** is used to select X+Y (**inc**=0) or X+Y+1 (**inc**=1) and **cmp** controls a bit inversion.

A flagged prefix adder can be converted into a flagged Ling adder applying transformations similar to those described in Section 4. Fig. 3 shows a block diagram of a flagged prefix adder and the modifications required to obtain the equivalent Ling adder. The only difference with respect to the transformation described in Section 4 affects to the sum cells. The white squares represent the flagged prefix sum cells that computes equation (15), while the black squares are the corresponding flagged Ling sum cells.

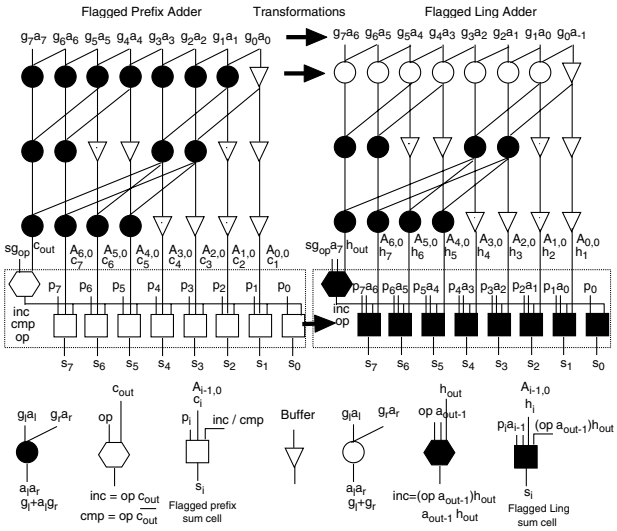


Figure 3. Transformation of a flagged prefix adder into a flagged Ling Adder.

The transformation method that follows can be applied to any flagged prefix sum cell, although we present a single example of a high-performance configuration for sign magnitude addition. The flagged prefix sum cell proposed in [1], implements equation (15) as

$$s_i = (p_i \oplus cmp) \oplus (c_i + A_{i-1,0}\; inc) \qquad (16)$$

This leads to a reduced hardware complexity sum cell. However, this configuration is not the most suitable for high-performance implementations, since in practical applications, signals **inc** and **cmp** are buffered and they depend on c_{out}. For instance, in a sign magnitude adder, these signals are given by $inc = c_{out}\; op$ and $cmp = \overline{c_{out}}\; op$, where op is the effective operation. The proposed implementation of a flagged prefix sum cell for sign magnitude addition is shown in the left side of Fig. 4(a). This sum cell implements equation (15) as

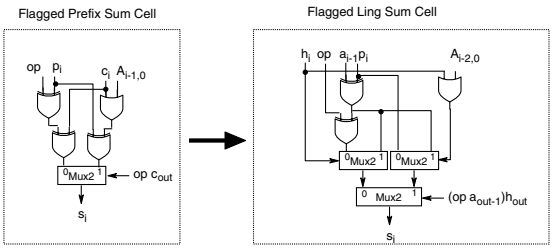
$$s_i = \Big(p_i \oplus (c_i + A_{i-1,0})\Big)\; inc + \Big(p_i \oplus c_i \oplus op\Big)\; \overline{inc}$$

with $inc = c_{out}\; op$.

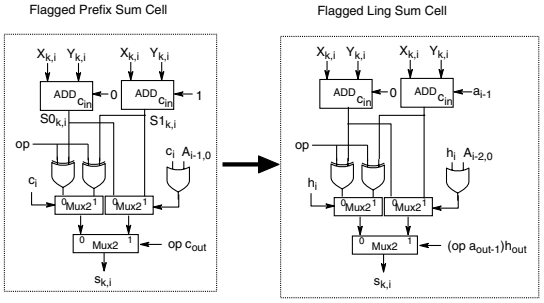
The logic expression for the equivalent flagged Ling sum cell of Fig. 4 is given by

$$\begin{aligned} s_i &= \Big((p_i \oplus a_{i-1})(h_i + A_{i-2,0}) + p_i\overline{h_i + A_{i-2,0}}\Big)inc_h \\ &+ \Big((p_i \oplus op \oplus a_{i-1})h_i + (p_i \oplus op)\overline{h_i}\Big)\overline{inc_h} \end{aligned}$$

with $inc_h = (op\; a_{out-1})h_{out}$. This flagged Ling sum cell have an additional hardware complexity of one XOR gate



(a) Binary sum cell.



(b) Two-conditional sum cell.

Figure 4. Transformation of flagged prefix into flagged Ling sum cells.

compared to the equivalent flagged prefix sum cell, but inc_h is obtained in one gate delay less than inc.

The proposed sum cell to implement sign magnitude addition in a flagged sparse prefix tree is shown in Fig. 4(b). The two conditional sums $S0_{k,i}$ and $S1_{k,i}$ are evaluated in parallel to carry computation and then the carries c_i, c_{out} and signal op select the correct result for each group in a final flagged sum cell. The result for each k-bit group in a flagged sparse carry-tree prefix adder is computed as

$$\begin{aligned} S_{k,i} &= \left(S1_{k,i}(c_i + A_{i-1,0}) + S0_{k,i}\overline{c_i + A_{i-1,0}}\right) inc \\ &+ \left((S1_{k,i} \oplus op)c_i + (S0_{k,i} \oplus op)\overline{c_i}\right) \overline{inc} \end{aligned}$$

The transformed flagged sparse carry-tree Ling sum cell of Fig. 4(b) computes each k-bit sum as

$$\begin{aligned} S_{k,i} &= \left(S1^*_{k,i}(h_i + A_{i-2,0}) + S0_{k,i}\overline{h_i + A_{i-2,0}}\right) inc_h \\ &+ \left((S1^*_{k,i} \oplus op)h_i + (S0_{k,i} \oplus op)\overline{h_i}\right) \overline{inc_h} \end{aligned}$$

where $S1^*_{k,i} = S1_{k,i}a_{i-1} + S0_{k,i}\overline{a_{i-1}}$. Both prefix and Ling sum cells of Fig. 4(b) present approximately the same hardware complexity, but inc_h is faster than inc.

7. Conclusions

We presented the Ling scheme as a transformation that can be applied to any prefix adder specification. The resultant Ling adders require minor modifications of the standard prefix adders, preserving the same topology but requiring less logical depth (roughly 10% reduction). The scheme is of special interest for sparse carry-tree adders, which are being used in industrial designs to achieve aggressive power-performance figures. Based on this simple formulation, we extended the use of the Ling method to the flagged prefix adder.

References

[1] N. Burgess, "The Flagged Prefix Adder and its Applications in Integer Arithmetic", *Journal of VLSI Signal Processing*, vol. 31, no. 3, pp. 263–271, July 2002.

[2] G. Dimitrakopoulos and D. Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders", *IEEE Trans. on Computers*, vol. 54, no. 2, pp. 225–231, Feb. 2005.

[3] R.W. Doran, "Variants of an Improved Carry-Lookahead Adder", *IEEE Trans. on Computers*, vol. 37, no. 9, pp. 1110–1113, Sep. 1988.

[4] M. Ercegovac and T. Lang, "Digital Arithmetic", Morgan Kauffman, 2003.

[5] J. Grad and J.E. Stine, "A Hybrid Ling Carry-Select Adder", in *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004*, vol. 2, pp. 1363–1367, Nov. 2004.

[6] S. Knowles, "A Family of Adders", in *Proc. 15th IEEE Symposium on Computer Arithmetic*, pp. 277–284, June 2001.

[7] H. Ling, "High-Speed Binary Adder", *IBM Journal of Research and Development*, vol 25, pp. 156–166, May 1981.

[8] S.K. Mathew et al., "A 4-GHz 300-mW 64-bit Integer Execution ALU with Dual Supply Voltages in 90-nm CMOS", *IEEE Journal of Solid-State Circuits*, vol. 40, no 1, pp. 44–51, Jan. 2005.

[9] S. Naffziger, "A Sub-Nanosecond 0.5μm 64b Adder Design", in *Proc. IEEE Solid-State Circuits Conf.*, pp. 362–363, Feb. 1996.

[10] A. Tyagi, "A Reduced-Area Scheme for Carry-Select Adders", *IEEE Trans. on Computers*, vol. C-42, no. 10, pp. 1163–1170, Oct. 1993.

[11] B. Zeydel, T. Kluter, and V. Oklobdzija, "Efficient Mapping of Addition Recurrence Algorithms in CMOS", in *Proc. 17th IEEE Symposium on Computer Arithmetic*, pp. 107–113, June 2005.

[12] N. Quach and M. J. Flynn, "High Speed Addition in CMOS", *IEEE Trans. on Computers*, vol. 41, no. 12, pp. 1612-1615, Dec. 1992.

[13] Y. Wang, C. Pai, and X. Song, "The Design of Hybrid Carry-Lookahed/Carry-Select Adders", *IEEE Trans. Circuits Syst. II*, vol. 49, no. 1, pp. 16–24, Jan. 2002.

[14] R. Zimmermann, "Binary Adder Architectures for Cell-Based VLSI and their Synthesis", Swiss Federal Institute of Technology Zurich, 1998.

[15] R. Zlatanovici et al., "Power-Performance Optimal 64-Bit Carry-Lookahead Adders", in *Proc. ESSCIRC 2003*, pp. 321–324, Sep. 2003.