Constant Addition utilizing Flagged Prefix Structures

James E. Stine, Christopher R. Babb, and Vibhuti B. Dave
Electrical and Computer Engineering Department
Illinois Institute of Technology
Chicago, Illinois 60616
Email: {jstine, cbabb, vdave}@ece.iit.edu

***Abstract*— The role of addition and subtraction in digital systems is sometimes complicated due to the arrival of certain operands arriving at different times. For example, floating-point arithmetic typically requires the exponent logic to wait until an output is received from post-normalization. Previously, logic designers have resorted to the use of conditional sum and carry-select adders to make their design efficient. Recently, a new technique has been proposed that utilizes the concept of flagged prefix addition. Flagged prefix addition utilizes the parallel-prefix adder and slightly modifies it to yield a new adder that is capable of adding $A+B$ or $A+B+1$. Moreover, alterations to the logic can easily be made to allow difference operations to occur as well. This paper presents an extension to the flagged prefix addition to allow an arbitrary number to be added to the logic. Results are shown for several design in AMI C5N 0.5 μm technology.**

I. Introduction

VLSI adders are critically important in digital designs since they are utilized in ALUs, memory addressing, cryptography, and floating-point units. Since adders are often responsible for setting the minimum clock cycle time in a processor, they can be critical to any improvements seen at the VLSI level. However, fast logarithmic time adders, such as carry-lookahead adders, can be impractical for a given VLSI implementation due to their prohibitive structures in terms of interconnect congestion and delay [1].

One method of improving carry-propagate adders for logarithmic time in VLSI design is to express it as a prefix computation [1], [2]. Using prefix computations are particularly attractive because they are easily expressed which leads to efficient implementations. In addition, the intermediate structures allow trade-offs between the amount of internal wiring and the fan-out of intermediate nodes, thereby, resulting in a more attractive combination of speed, area and power especially for sub-micron technologies [3].

Many applications require the addition or subtraction of two operands followed by the addition of a constant. Unfortunately, even with advances in technology, datapaths typically result to two or more carry-propagate adders to complete the operation. This is because there are many cases when a constant cannot be easily integrated within a given architecture. The conditional sum adder is one solution that generates a pair of sum and carry bits at each bit position. One pair assumes a carry-in of one and the other assumes a carry in of zero. The correct sum and carries are then subsequently selected using a tree of multiplexors. However, the conditional-sum adder suffers from large fan-out and large area constraints [4].

One particular solution is to modify a prefix adder so that the selected sum bits can be inverted as required to derive the second result [5], [6]. This new adder, called a *flagged prefix adder* can exploit a prefix adder with a small amount of additional hardware and exploit a prefix adder's flexibility for fan-out loading and wire densities. This paper extends the flagged prefix adder so that an arbitrary constant can be integrated within the structure.

II. Recurrence Relationship

Binary carry-propagate adders can be efficiently expressed as a prefix computation [7]. That is, through the basic operation of $c_{i+1} = a_i \cdot b_i + (a_i + b_i) \cdot c_i$. Parallel-prefix logic combines n inputs using an arbitrary associative operator $\circ$ to n outputs so that the output Sum_i depends only on the input operands [1].

The key to fast addition is the calculation of the carries c_i [8]. This can be computed utilizing a recurrence relationship:

$$c_{i+1} = g_i + p_i \cdot c_i \tag{1}$$

with the $generate$ or g signal being equal to $g_i = a_i \cdot b_i$, and the $propagate$ or p signal being equal to $p_i = a_i + b_i$. Some adders utilize propagate as $p_i = a \oplus b$ to exploit a specific circuit structure.

Prefix addition is carried out in three consecutive steps called the preprocessing stage, parallel-prefix carry computation, and the postprocessing stage as shown in Figure 1. Generate and propagate signals are typically produced within the pre-processing stage, whereas, the prefix structure computes the recurrence relationship for the carries. The postprocessing stage computes the sum completing the addition.

A. Flagged Prefix Addition

Flagged addition relies on the simple idea of speculatively computing two operands. These two values either compute a sum with a carry in of one or zero. A **flag** bit is introduced to enable logic to indicate which sum to compute [9], [10]. For example, the addition of $x = 9$ and $y = 78$ is shown below

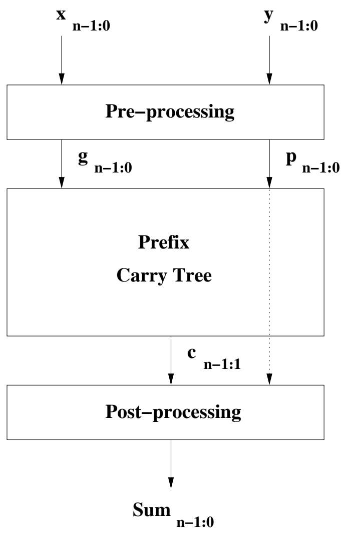

Fig. 1. Block diagram of a parallel prefix adder

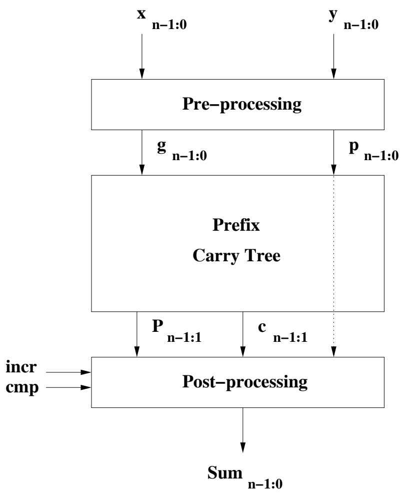

Fig. 2. Block diagram of a flagged-prefix adder

where a flag is utilized to add $x + y + 1$

$$\begin{aligned} x &= 0000\_1001 \\ y &= 0100\_1110 \\ Sum &= 0101\_0111 \\ F &= 0000\_1111 \\ Sum + 1 &= 0101\_1000 \end{aligned}$$

The flag bit is utilized to compute the new sum by utilizing the exclusive-or of the sum and the flag. Flagged prefix can be utilized to perform an increment operation (i.e. $x + y + 1$) or a decrement operation $a - b - 1$ [5], [6].

The flag bits have been shown to be easily generated from the prefix structure by relying on the group generate and group propagate signals from the prefix structure. From these two signals, *late carry* signals can be produced from $c_{k+1} = G_k + P_k \cdot incr$ where $incr$ indicates the bit that should be incremented. The increment can be easily computed in the post-processing stage or as a select signal for a conditional sum adder. A block diagram of the flagged prefix adder is shown in Figure 2.

To implement this operation, a pair of enable bits are utilized to invert the appropriate sum bits to derive the desired result. Two enable bits, cmp and $incr$ enable the inversion of the flagged bits or the inversion of the unflagged bits, respectively [5]. Normally, the case of $(cmp, incr) = (1, 1)$ is a "don't care" condition since the complement and increment operations cannot occur concurrently. A straightforward implementation of the flagged inversion is shown in Figure 3 [5]. Since this circuit has the availability of producing several inputs earlier, the critical path is minimally impacted.

III. Constant Flag Generation

For this section, the flagged prefix operation is extended for an arbitrary constant, M. That is, the computation of $x + y \pm M$. This paper only deals with adding unsigned values, however, based on previous work [6], the logic can easily be extended for modulo $2^w - 1$, absolute value, and sign-magnitude addition [6].

A major building block in arithmetic systems is the full adder (FA). A FA takes three bits x_k, y_k, and c_k and produces two outputs: a sum bit R_k and a carry bit c_{k+1}. Sometimes, because a FA counts the number of ones that are available at its input it sometimes is called a $(3, 2)$ counter. The traditional logic equations for a FA are:

$$\begin{aligned} R_k &= x_k \oplus y_k \oplus c_k \\ c_{k+1} &= x_k \cdot y_k + x_k \cdot c_k + y_k \cdot c_k \end{aligned} \quad (2)$$

Assuming the input is an arbitrary value to be added with a constant, the FA equations can be rewritten assuming R_k is a value that is to be augmented or decremented by a M_k, such that:

$$\begin{aligned} Sum_k &= R_k \oplus M_k \oplus c_k \\ c_{k+1} &= R_k \cdot M_k + R_k \cdot c_k + M_k \cdot c_k \end{aligned} \quad (3)$$

Utilizing these equations, the new function, called a flag function, can be computed such that $F_k = M_k \oplus c_{k+1}$ where

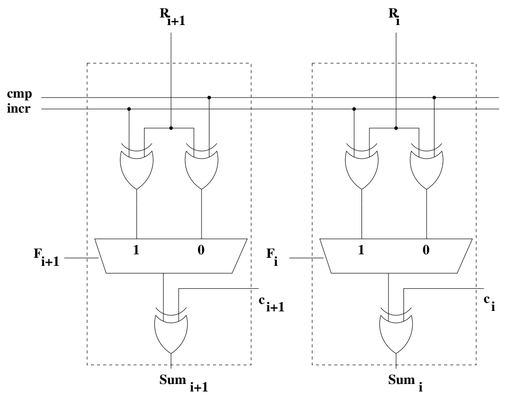

Fig. 3. Flagged inversion logic (Adapted from [5])

M_k	M_{k-1}	c_k	F_k
0	0	$R_{k-1} \cdot F_{k-1}$	$R_{k-1} \cdot F_{k-1}$
0	1	$R_{k-1} + \overline{F_{k-1}}$	$R_{k-1} + \overline{F_{k-1}}$
1	0	$R_{k-1} \cdot F_{k-1}$	$\overline{R_{k-1} \cdot F_{k-1}}$
1	1	$R_{k-1} + \overline{F_{k-1}}$	$\overline{R_{k-1}} \cdot F_{k-1}$

TABLE I

OUTPUT LOGIC FOR SELECTION OF REQUIRED RESULT.

M_k	M_{k-1}	$F_k, (c_k = 0)$	$F_k, (c_k = 1)$
0	0	$R_{k-1} \cdot F_{k-1}$	$\overline{R_{k-1}} \cdot F_{k-1}$
0	1	$R_{k-1} + \overline{F_{k-1}}$	$\overline{R_{k-1} \cdot F_{k-1}}$
1	0	$\overline{R_{k-1} \cdot F_{k-1}}$	$R_{k-1} + \overline{F_{k-1}}$
1	1	$\overline{R_{k-1}} \cdot F_{k-1}$	$R_{k-1} \cdot F_{k-1}$

TABLE II

MODIFIED OUTPUT LOGIC FOR SELECTION OF REQUIRED RESULT UTILIZING CARRY PRODUCED FROM THE PREFIX CPA.

F_k is the flag. The flag function is utilized such that it determines whether the current value is flagged to change [5]. Consequently, this structure can be formulated by developing flag equations based on speculative elements of the constant:

$$\begin{aligned} c_{k+1} &= \begin{cases} R_k \cdot c_k, & if\ M_k = 0 \\ R_k + c_k, & if\ M_k = 1 \end{cases} \qquad (4) \\ F_k &= \begin{cases} c_k, & if\ M_k = 0 \\ \overline{c_k}, & if\ M_k = 1 \end{cases} \end{aligned}$$

Similar to the conditional sum adder, the constant is chosen based on either 1 or 0 for the present bit or previous bit [11]. Utilizing the equations above, and plugging in the constant, assuming the input is M_k and M_{k-1}, new flag equations can be computed. In other words, two bits of the constant are examined to determine whether not the carry bit from the constant affects the current position. For example, assume that the $M_k = 0$ and $M_{k-1} = 1$. Utilizing the relationships in Equation 5,

$$\begin{aligned} c_{k+1} &= R_k \cdot c_k \ \exists \ M_k = 0 \qquad (5) \\ c_k &= R_{k-1} + c_{k-1} \ \exists \ M_{k-1} = 1 \\ \therefore \ c_{k+1} &= R_k \cdot F_k \\ \therefore \ c_k &= R_{k-1} + \overline{F_k - 1} \end{aligned}$$

Table I shows the complete table for both the flag and carry equations for c_k and F_k.

The generation of a flag computation is complicated due to the fact that the value of R_k is produced from the addition of two numbers x_k and y_k utilizing a carry-propagate adder (CPA). Therefore, not only is the computation based on a carry produced from the constant as described above, however, the carry of the prefix adder must be utilized within the flag equations. Based on this carry, this results in the following relationship based on the value of the carry from the CPA:

$$R_{k-1} = \begin{cases} x_{k-1} \oplus y_{k-1}, & if\ c_k = 0 \\ \overline{x_{k-1} \oplus y_{k-1}}, & if\ c_k = 1 \end{cases} \qquad (6)$$

In other words, Table I can be rewritten based on the value of the carry assuming $R_{k-1} = x_{k-1} \oplus y_{k-1}$ as shown by Equation 6. Table II shows the new logic equations for the flag based on the constant M_k. The new equations for $c_k = 1$ can be also produced based on Table I and the one's complement of the constant bits, M_k and M_{k-1}.

In order for the equations to be computed properly, some initial conditions are required in order to guarantee the equations work properly. In these assumptions, similar to other algorithms which two bits are examined at a time, the least significant bit assumes that 0 is examined at the M_{-1} position. That is, the following initial conditions are assumed:

$$\begin{aligned} R_{-1} &= M_{-1} = F_{-1} = 0 \qquad (7) \\ \therefore \ F_0 &= M_0 \end{aligned}$$

Utilizing the previous example in adding $x = 9$ and $y = 78$, and suppose the constant to be added is $M = 0011\_1001_2 = 57_{10}$, the flag equations utilizing the equations above are:

$$\begin{aligned} M &= 0011\_1001 \\ x &= 0000\_1001 \\ y &= 0100\_1110 \\ Sum &= 0101\_0111 \\ F &= 1100\_0111 \\ Sum + 57 &= 1001\_0000 \end{aligned}$$

A. *Modification to the prefix adder*

The modifications are done such that the prefix structure can handle the flag bit. The constant is assumed to be known beforehand so that each prefix cell can output the correct flag equation and pass it on to the post-processing block. However, the logic could be modified to handle the constant as an input. In summary, the following modifications are required:

- A new cell is added to the prefix block to pass on the group propagate signal. This input is utilized along with the group generate to produce the carry required to produce the flag.
- Flag logic, according to a specific constant, is required.
- A pair of "invert enable" bits are provided to enable the control of the inversion of the sum bits. In this paper, an exclusive-or gate is utilized to handle the inversion. A multiplexor could also be utilized.

As presented in [5], the original flagged prefix of $x + y + 1$ is introduced. Using Table II, the flag equations are:

$$F_k = \begin{cases} 1, & for\ F_0 \\ R_0, & for\ F_1 \\ \overline{R_{k-1}} \cdot F_{k-1} \cdot \overline{c_{k-1}} \ + & \\ \overline{R_{k-1}} \cdot F_{k-1} \cdot c_{k-1}, & for\ F_{n-1}, \ldots, F_2 \end{cases}$$

This produces an equation that relies on the carry produced from the CPA, as well as not affecting the critical path. The original derivation in [5] utilized both group generates and propagates to produce the correct carry for this specific example (e.g. c_{k-1}). However, carry propagate adders other than prefix adders could be utilized as well to still provide good results.

If another constant is chosen, logic for the flag changes according to Table II. Fortunately, the logic for the flag can be pre-computed, thus, not affecting the critical path and promoting time-borrowing. On the other hand, there may be specific designs that incorporate constants that may incur more delay than other constants. As stated previously, the constant could also be an input, however, this would dramatically affect the area since all equations within Table II are required.

IV. Results

Several designs were implemented to examine the impact upon a typical Application Specific Integrated Circuit (ASIC) design. An AMI C5N 0.5μm System-on-Chip (SoC) design flow is selected to examine the impact of these designs [12]. The nominal operating voltages for the library are 5.0 Volts and are simulated at $T = 25\ ^\circ$C. A Brent-Kung prefix adder is utilized for the implementation, however, other prefix structure can equally be implemented. Moreover, a C program is utilized to exhaustively test that the equations in Table II to correctly produce the correct sum.

Synthesis is performed with Synopsys Design Compiler. Cadence Silicon Ensemble is used in script mode and performs both, placement and routing. The Verilog netlists were optimized utilizing Synopsys directives for area. Other design factors could be employed to achieve lower delay and power considerations as well. Layouts are generated for the adders and parasitically extracted to obtain accurate numbers for area, delay, and speed. Delay numbers are obtained utilizing Synopsys' Pathmill. Pathmill is a cell-based static timing tool that utilizes netlists to achieve accurate delay estimates. The results are shown in Table III.

Results indicate that flagged prefix adders can be easily inserted into the prefix structure without significantly affecting the critical path. Interestingly, for all designs, the flagged prefix adder performed better than the non-flagged version. This occurred because Synopsys' synthesis package optimized the structure for area. More than likely, the improvement would be minimal for normal custom-layout implementations. However, the disadvantage for flagged prefix adders is that significant area is consumed as the operand size increases. However, with certain circuit styles, flag equations could be implemented more efficiently at the physical level. In should be noted that certain constants may cause certain problems in terms of implementation since some constants can cause the prefix structure to become more congested. On the other hand, for adding constants that have low amounts of entropy, this method achieves small amounts of hardware and high amounts of throughput without requiring an additional carry-propagate adder.

	Prefix		Flagged Prefix	
n	Delay (ns)	Area (mm^2)	Delay (ns)	Area (mm^2)
4	2.006	0.0477	2.028	0.0498
8	4.736	0.0671	4.409	0.0784
16	10.016	0.1159	9.890	0.1397

TABLE III
Post-layout Estimates for Brent-Kung Prefix and Flagged Adders (i.e. flagged is $x + y + 57$). Note: Since $M = 57$ is not implementable for $n = 4$, the constant, M, is truncated to the integer value 9 .

V. Conclusion

In this paper, flagged prefix addition is presented for arbitrary constant addition. With flagged addition, a constant can easily be inserted into a parallel-prefix adder scheme. Results in a sub-micron standard cell library indicate efficient designs with high amounts of speed. Flagged adders also allow flexibility in the algorithm's ability to execute time borrowing as well as computing differences. Since this structure is generic, it can equally be applied to other adders. Unfortunately, adders other than prefix adders might be prohibitive due to the excessive delay in computing the flags since the prefix structure computes the flags not on the critical path.

References

[1] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transaction on Computers*, vol. C-31, pp. 260–264, 1982.

[2] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, vol. C-22, pp. 783–791, 1973.

[3] S. Knowles, "A family of adders," in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 30–34.

[4] R. K. Yu and G. B. Zyner, "167 MHz floating-point multiplier," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, 1995, pp. 149–154.

[5] N. Burgess, "The flagged prefix adder for dual addition," in *Proceedings of SPIE-the international society for opticl engineering*, vol. 3461, 1998, pp. 567–575.

[6] ——, "The flagged prefix adder and its applications in integer arithmetic," *Journal of VLSI Signal Processing*, vol. 31, no. 3, pp. 263–271, 2002.

[7] H. Lindkvist and P. Andersson, "Techniques for Fast CMOS-based Conditional Sum Adders," in *Proceedings of the 1994 International Conference on Computer Design*, October 1994, pp. 626–635.

[8] S. Winograd, "On the time required to peform addition," *Journal of the ACM*, vol. 12, no. 2, pp. 277–285, April 1965.

[9] S. Cui, N. Burgess, M. J. Liebelt, and K. Eshraghian, "A GaAs IEEE floating-point standard single precision multiplier," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, 1995, pp. 91–97.

[10] A. Tyagi, "A reduced-area scheme for carry-select adders," *IEEE Transactions on Computers*, vol. 42, pp. 1163–1170, 1993.

[11] J. Sklansky, "Conditional-sum addition logic," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 6, pp. 226–231, 1960.

[12] J. Grad and J. E. Stine, "A Standard Cell Library for Student Projects," in *International Conference on Microelectronic Systems Education*. IEEE Computer Society Press, 2003, pp. 98–99.