Design and Synthesis of Flagged Binary Adders with Constant Addition

Vibhuti Dave, Erdal Oruklu, and Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, Illinois, USA
{vdave,eoruklu,sansonic}@ece.iit.edu

***Abstract*— Multi-operand addition is utilized in various applications like, multiplication, convolution and several image processing algorithms such as filtering. Various adder architectures have been proposed to accomplish the addition of more than two operands, consuming minimum delay and area. Recently, a new technique called flagged prefix addition has been proposed that utilizes prefix tree adders to perform increment and decrement operations by generating flag bits. An extension to this adder has also been proposed enhancing the functionality of the same to allow the addition of any arbitrary number, thereby accomplishing multi-operand addition. This paper extends the idea of generating the flag bits to the carry-skip and the carry-select adders. A thorough evaluation has been performed to analyze the performance of the carry–select, carry–skip, and the prefix tree architectures incorporating the new design in terms of power, area, and delay.**

***Index Terms*—Binary Addition, Constant Addition, Flagged Addition, Prefix Computation.**

I. INTRODUCTION

Multi-operand addition is one of the fundamental operations inherited in many complex digital signal processing and also simple multiplication algorithms. Several multi-operand adders have been proposed to accomplish the required operation. One of the techniques proposed, is to arrange carry-save adders in an array of multiple stages forming various tree structures [1]. The commonly used carry save adders, are the [3,2], [4,2] and the [7,3] counters [1].

This paper investigates the use of parallel prefix, carry–select, and carry-skip adders [2] in order to accomplish basic multi-operand addition. Many applications require the addition of two operands followed by the augmentation of the result by a constant. A scheme has been proposed in [3] to accomplish this by using parallel prefix adders.

The parallel prefix adders are modified to generate a new adder design called the flagged prefix adders [4]. The flagged prefix adder uses a simple technique of inverting only selected sum bits to derive increment *(A+B+1)* and decrement *(A+B-1)* operations in addition to normal addition *(A+B)* and subtraction *(A-B)* outcomes. Additional intermediate outputs, called the flag bits are produced in order to select the appropriate sum bits to be inverted necessary to achieve the desired result. A similar concept of generating flag bits has been utilized to obtain the result of adding any arbitrary constant *(A+B+M)* following the addition/subtraction of two numbers [3]. This technique has proven to perform better than using two-stage adders in terms of area as well as delay. The amount of extra hardware needed to generate the flag bits is minimal in both cases since they directly depend on the propagate signals or the carry outputs from the adder.

This paper investigates the performance of parallel prefix, carry–skip, and carry–select adders for 16–bit and 32–bit operand sizes, modified to incorporate flagged binary addition for increment and decrement operations as well as for constant addition. The carry-skip adder has been selected since it utilizes propagate signals to generate the final carry signals and hence can also be utilized to compute the necessary flag bits. These flag bits can be used to generate the final sum bits for the increment and decrement operations. The carry signals can be used to compute the flag bits for the constant addition. The carry select adder does not utilize propagate signals and therefore has been modified only to perform constant addition.

The different adder architectures are reviewed in Section II. Section III describes the concept of flag bits and the necessary modifications that need to be made to incorporate the logic into fundamental adder architectures. Section IV presents the trade-offs in area speed and power for the different implementations. The final conclusion is presented in Section V.

II. BACKGROUND

Addition is a fundamental operation and often, the delay of the adder circuit determines the clock cycle time of a processor due to the rippling nature of the carry. This section describes three adder architectures that have been used to investigate the performance of flagged binary adders with the capability to add a constant to the result of the sum of two numbers. The key to fast addition is to calculate the

carry signals for all bit positions in parallel. The recurrence relationship presented in (1) achieves this conveniently by introducing the *generate* or *g* signal given by $g_i = a_i \cdot b_i$, and the *propagate* or *p* signal given by $p_i = a_i + b_i$, where *i* represents the bit position [5].

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

A. Prefix Adders

The parallel prefix adder accomplishes the computation of the output carries in parallel by expressing binary carry propagate addition as a prefix computation. Parallel Prefix logic combines *n* inputs using an arbitrary associative dot operator,○, to *n* outputs so that the outputs Sum_i depend only on the input operands. The ○ operator is shown in (2) where (g_1, p_1) and (g_2, p_2) are the inputs and, (G, P) are the outputs. [6]. The parallel prefix adder computes the sum in three stages. This is illustrated in Fig. 1 [4].

$$\begin{aligned} G &= g_1 + g_2 \cdot p_1 \\ P &= p_1 \cdot p_2 \end{aligned} \tag{2}$$

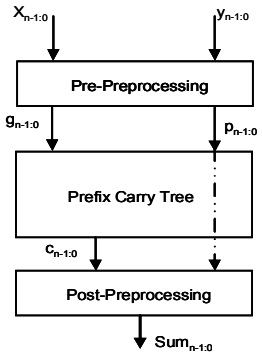

Figure 1. Parallel Prefix Adder

In the block diagram, illustrated in Fig. 1, *x* and *y* represent the *n*-bit operands. *p* and *g* represent the *propagate* and the *generate* signals described earlier. These signals are utilized to compute the carries by the prefix carry tree. The prefix carry tree is an interconnection of a number of *black, gray* and *buffer* cells where the logic for which is illustrated in Fig. 2. The *black* cell is a complex logic gate that performs the ○ operation. $G_{m:n}$ and $P_{m:n}$ represent the *Group Generate* and *Group Propagate* signals across the bits from significance *m* up to and including significance *n* [4]. The inputs to these cells come from the pre-processing stage of the adder. The *gray* cell is similar to the *black* cell, except that it does not output the *group propagate* signal. These cells are connected to form a multi-level tree structure. The output of the tree is then passed on to the post processing stage to produce the final *sum*. The prefix trees selected for the analysis of performance of flagged binary adders are the Brent-Kung, Ladner-Fischer [7], and the Kogge-Stone [8] structures. Each of these trees is represented in Fig. 3, 4 and 5 respectively.

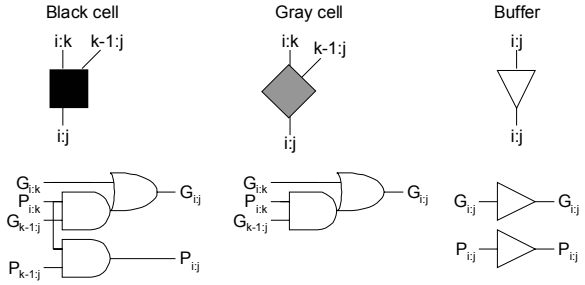

Figure 2. Logic Gates within the Prefix Carry Tree

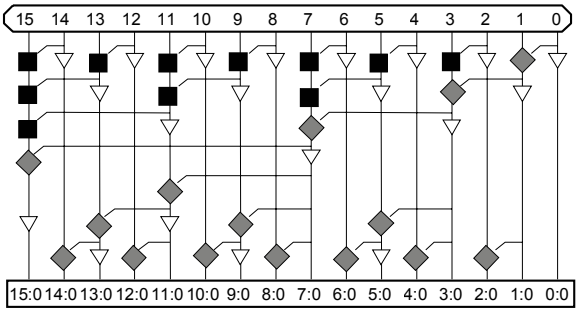

Figure 3. Brent - Kung Prefix Tree

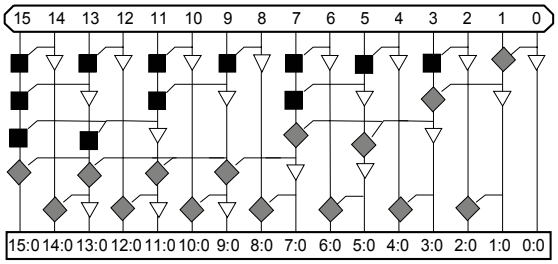

Figure 4. Ladner-Fischer Prefix Tree

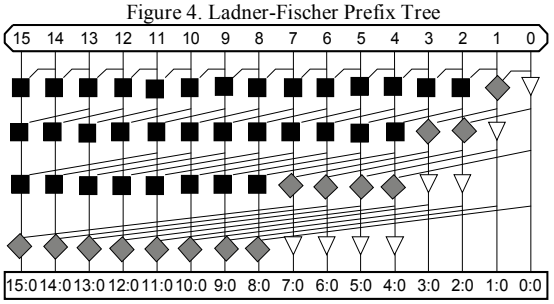

Figure 5. Kogge - Stone Prefix Tree

B. Carry Skip Adders

A carry-skip adder uses the concept of generating the *group propagate* signal in order to determine if the *carry out* of a set of bits is identical to the *carry in* [9]. The carry -skip adder uses a regular full adder for every bit position and in addition also generates the *bit propagate* signal, *p.* The adder structure is divided into blocks of consecutive stages with the full adder scheme modified to output the *bit propagate* signal. Every block generates a *group propagate* signal represented as $P_{i:k}$. This signal determines, whether the *carry out,* c'_{i+1} of the block is propagated to the next block,

or if it is skipped and instead, the input carry, c_k is directly selected as the *carry out,* c_{i+1}. This is expressed according to the following equation [9]:

$$c_{i+1} = \overline{P_{i:k}} c'_{i+1} + P_{i:k} c_j \quad (3)$$

The block diagram in Fig. 6, [9] describes this operation. A multiplexer is used for the selection. The 16-bit adder used for this paper can be seen in Fig. 7. The 32-bit adder uses a two-level implementation for the structure.

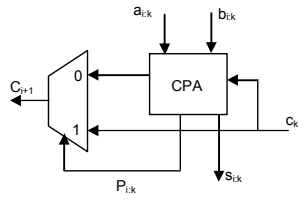

Figure 6. Single Carry Skip Adder Module

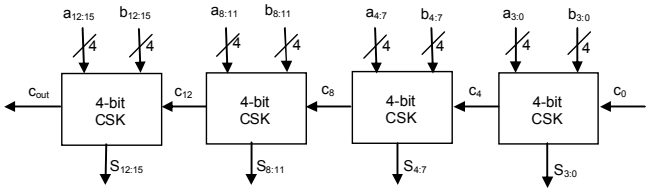

Figure 7. 16-bit Carry Skip Adder

C. *Carry Select Adders*

The under-lying strategy of the carry select adder is to generate two results in parallel. One result assumes the *input carry* to be a zero and the other assumes the *input carry* of one. The carry select adder is divided into blocks of m-bit vectors. Each block generates two outputs according to the equations presented in (4) [1].

$$\begin{aligned} (c_m^0, S^0) &= ADD(X, Y, c_0 = 0) \\ (c_m^1, S^1) &= ADD(X, Y, c_0 = 1) \end{aligned} \quad (4)$$

Here, X, Y and S are *m*-bit vectors. A 16-bit carry select adder used for this paper is shown in Fig.8. Once the input carry for a particular stage has been computed and assigned, the final result is selected from the two pre-computed sets.

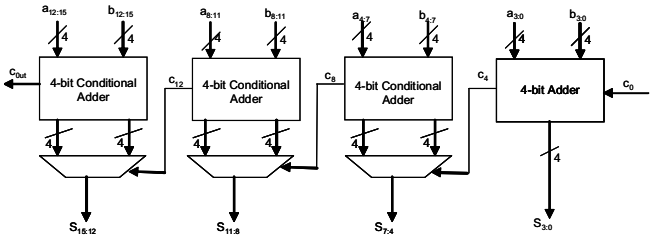

Figure 8. 16-bit Carry-Select Adder

III. FLAGGED BINARY ADDITION

Flagged Binary Addition is based on the concept of generating a set of bits called the *flag* bits [2]. The flag bits are utilized to invert selected sum bits at the output of a regular adder in order to accomplish results other than *A+B* and *A-B*. This was first proposed in [4]. This can be easily demonstrated with the following example:

x	=	0	0	0	0	1	0	0	1
y	=	0	1	0	0	1	1	1	0
Sum	=	0	1	0	1	0	1	1	1
F	=	0	0	0	0	1	1	1	1
Sum+1	=	0	1	0	1	1	0	0	0

The result, *Sum+1* is a result of exoring the flag bits, *F* and the *Sum* bits. The flag bits are very simply related to the *Group Propagate* signals [4] according to (5):

$$f_i = P_{i-1:0} \quad (5)$$

The resulting flag bits can be used according to the flag inversion logic presented in Fig. 9 [4], to compute results such as *(A+B+1), (B-A-1)*.

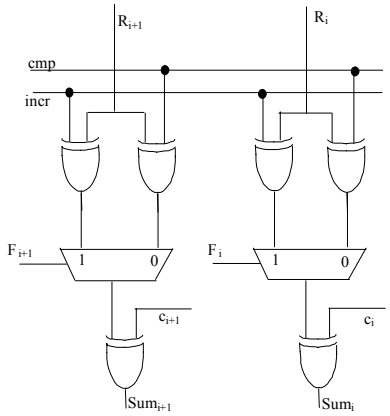

Figure 9. Flag Inversion Logic [4]

In [3], it is proposed to generate a new set of flag bits in order to compute *(A+B+M)*, where *M* is any constant. Assume that *R* is the result of adding two arbitrary inputs, *A* and *B,* and *R* needs to be augmented/decremented by a value, *M*. The full adder equations can be written as [3]

$$Sum_k = R_k \oplus M_k \oplus c_k$$

$$c_{k+1} = \begin{cases} R_k \cdot c_k & if \;\; M_k = 0 \\ R_k + c_k & if \;\; M_k = 1 \end{cases} \quad (6)$$

$$F_k = \begin{cases} c_k & if \;\; M_k = 0 \\ \overline{c_k} & if \;\; M_k = 1 \end{cases} \quad (7)$$

Utilizing the new set of equations, the new sum needs to be computed such that, $R+M=R \oplus F$, where *F* is the flag function. The flag bits can be seen as bits that indicate whether the current value is flagged to change. Consequently, the flag bits can be computed based on speculative elements of the constant. Two bits of the

constant are examined to determine whether or not the carry bit from the constant affects the current position. As derived in [3], the flag bits are computed according to Table I.

TABLE I. Output Logic For Selection Of Required Result Utilizing Carry Produced From The CPA

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	$\overline{R_{k-1}} \cdot F_{k-1}$	$R_{k-1} \cdot F_{k-1}$

$$R_{-1} = M_{-1} = F_{-1} = 0 \tag{8}$$

Continuing with the same example as presented before, with $x=9$, $y=78$, and $M=0011\_1001_2=57$, we get

M	=	0	0	1	1	1	0	0	1
x	=	0	0	0	0	1	0	0	1
y	=	0	1	0	0	1	1	1	0
Sum	=	0	1	0	1	0	1	1	1
F	=	1	1	0	0	0	1	1	1
Sum+57	=	1	0	0	1	0	0	0	0

In order to accomplish the extra functionality as a part of the flagged adder, minimal extra logic depending on the specific constant needs to be included. This adder with the additional functionality has been called the enhanced flagged adder.

A. Prefix Adder Modifications

Minimal amount of hardware and minor modifications are required in order to incorporate the flag logic within a prefix adder. Since the flag bits depend on the *group propagate* signal according to (4), it is necessary to compute the required *group propagate* signal. The prefix carry tree is modified in order to generate the required signals simply by changing all the *gray* cells to *black* cells, thereby obtaining *group propagate* signal in addition to the *group generate* output [4]. The resulting flag bits are utilized as presented in Fig. 9. This is called the Flag Inversion Logic [4] since the selected sum bits are inverted to achieve the desired result. The signals, *incr* and *cmp* are used to select the appropriate result from all the possible results as presented in Table II [4]. The minimal amount of hardware therefore comprises of two levels of XOR gates and a multiplexer, thereby affecting the critical path minimally.

TABLE II. Selection Logic

Incr	cmp	Result(Addition)	Result(Subtraction)
0	0	A+B	A-B-1
0	1	A+B+1	A-B
1	0	-(A+B+2)	B-A
1	1	-(A+B+1)	(B-A-1)

Flag bits, necessary for the enhanced flagged adder are computed using the carry signals obtained at the output of the prefix tree. The minimal amount of hardware required to compute the flag bits will depend on the constant that is being added. As can be seen in Table I, the extra hardware is a few OR and AND gates and n multiplexers to select the appropriate flag bit depending on the output carries from the prefix tree.

B. Carry Skip Adder Modifications

The carry-skip adder can conveniently incorporate the computation of flag bits since, it generates the *bit propagate* signals for every bit position. The block computing the sum for bits $m:n$ will also generate $P_{m:n}$. Therefore the computation of the flag bit for position i will require an AND gate according to the following equation:

$$P_0^m = \underset{i=0}{\overset{i=m}{AND}}\, p_i \tag{9}$$

The carry-skip adder for this paper is divided into group sizes of four. Therefore, the maximum fan-in for the AND gate used to compute the flag bit is four.

C. Carry Select Adder Modification

The carry-select adder does not generate the *propagate* signals and therefore has not been modified to implement the flagged binary adder. Instead it has been utilized only to incorporate the hardware for the enhanced version of the flagged adder. Again, the logic remains the same as for the previous two cases.

IV. RESULTS

The binary adders described in the previous sections are implemented and modified to be able to perform increment and decrement operations. The adders are also modified to incorporate the flag logic according to Table I, enhancing the functionality of flagged adders to perform constant addition. Each adder was designed for 16-bit and 32-bit operand sizes. An analysis was performed on all adders with regards to, area, delay, and power. The designs are implemented in the TSMC 0.18µm technology System-on-Chip design flow to investigate the power, area, and delay tradeoffs. Synthesis is performed with Cadence Build Gates and Encounter [10]. The nominal operating voltage is 1.8V and simulation is performed at T=25°C. Layouts are generated for each adder design and parasitically extracted to obtain numbers for area, delay and power. The results for conventional binary adders without any modifications are presented in Table III. It is observed, that the area increases by approximately two times for all architectures when the operand size is doubled. However, delay measurements do not vary by more than 6% with increase in operand size. The increase in power with change in operand size is significant for prefix structures compared to the carry-skip and carry-select adders. This can be due to the high fan in and significant wiring complexity associated with the prefix trees. A trade-off between area and delay can be observed for the Kogge-Stone prefix adder, which is observed to be the fastest design among all adder architectures and consumes maximum area.

TABLE III. Post Layout Estimates For Conventional Adders

Adder/Parameters	16-Bit			32-Bit		
	Area (mm2)	Delay (ns)	Power (mW)	Area (mm2)	Delay (ns)	Power (mW)
Brent Kung	0.2756	16.825	5.63E-04	0.6891	16.853	2.25E-03
Ladner Fischer	0.2763	14.025	5.81E-04	0.6907	14.118	2.19E-03
Kogge Stone	0.4961	11.412	7.78E-04	1.2403	11.435	3.12E-03
Carry Skip	0.5919	15.55	4.51E-04	1.1956	15.83	1.00E-03
Carry Select	0.7004	14.09	5.93E-04	1.2137	14.92	1.29E-03

TABLE IV. Post Layout Estimates For Flagged Binary Adders

Adder/Parameters	16-Bit			32-Bit		
	Area (mm2)	Delay (ns)	Power (mW)	Area (mm2)	Delay (ns)	Power (mW)
Brent Kung	0.2803	16.829	7.06E-04	0.7103	16.857	2.83E-03
Ladner Fischer	0.2821	14.032	7.27E-04	0.7163	14.125	2.91E-03
Kogge Stone	0.5362	11.417	9.92E-04	1.3418	11.44	3.96E-03
Carry Skip	0.6474	17.73	7.71E-04	1.3807	23.58	2.65E-03
Carry Select	N/A	N/A	N/A	N/A	N/A	N/A

TABLE V. Post Layout Estimates For Enhanced Flagged Adder

Adder/Parameters	16-Bit			32-Bit		
	Area (mm2)	Delay (ns)	Power (mW)	Area (mm2)	Delay (ns)	Power (mW)
Brent Kung	0.2921	16.901	8.08E-04	0.7143	17.054	2.96E-03
Ladner Fischer	0.2933	14.242	8.34E-04	0.8045	14.338	3.04E-03
Kogge Stone	0.5462	11.526	1.12E-03	1.5341	11.614	4.19E-03
Carry Skip	0.6632	18.36	1.66E-03	1.5918	24.159	2.72E-03
Carry Select	0.8047	17.257	1.30E-03	1.7214	24.637	3.61E-03

Table IV shows the post–layout estimates for flagged binary adders. Additional hardware is required to generate the necessary flag bits. This hardware accounts for an increase in area by approximately 4.5% for 16-bit as well as 32-bit operand sizes. However, the change in delay is more significant for the carry-skip designs. The flag logic has a minimal impact on the critical path for prefix structures since the propagate signals are obtained at the output of the prefix tree in parallel. For the carry skip design, the *group propagate* signal actually ripples from one stage to another. In addition, for the 32-bit design, a 2-level carry skip adder has been implemented resulting in a significant rise in delay. The extra hardware affects the carry-skip adder less significantly relative to the increase in power observed for the prefix designs. Table V shows results for the enhanced flagged binary adder. For the presented simulation results, the value of the constant has been chosen as 57. The extra hardware required to generate the necessary flag bits is observed to increase the area of the Kogge-Stone adder by 10% compared to the 6% increase for the Ladner-Fischer and Brent-Kung designs. This holds true for both operand sizes. The modifications for the flag logic fall in the critical path for the carry-skip and carry-select designs since it depends on the output carries for each bit position. This can be associated with the significant rise in delay for corresponding adder designs.

V. CONCLUSIONS

This paper extends the idea of generating flag bits to perform increment, decrement and constant addition operations to carry-skip and carry-select adders thereby not limiting the usefulness of flag bits to prefix adders. The Kogge-Stone performs the best in terms of delay for all three versions. The prefix adders would be able to incorporate the flag logic more efficiently than the carry-skip and carry-select designs. However, even for carry-skip and carry-select adders, augmentation of a result by a constant could be implemented more efficiently with the method described in the paper when compared to use of dual adders or carry-save adders.

REFERENCES

[1] Milos Ercogavac and Tomas Lang. *Digital Arithmetic*. Morgan Kauffmann, First Edition, 2004.

[2] A. Beaumont-Smith, N.Burgess, S.Lefrere, C.C.Lim. Reduced Latency IEEE Floating Point Standard Adder Architectures, 1999.

[3] James Stine, Chris Babb, V. Dave. Constant Addition utilizing Flagged Prefix Structures. In *7th Euromicro Conference on Digital System Design*, 2004.

[4] N.Burgess. The flagged Prefix Adder and its Applications in Integer Arithmetic. *Journal of VLSI Signal Processing*, 31(3):263-271,2002.

[5] S.Knowles. A family of adders. In *Proceedings in the 14th IEEE Symposium on Computer Arithmetic*, pages: 30-34, 1999.

[6] R. Brent, and H.Kung. A regular layout for parallel adders. *IEEE Transactions on Computers,* C-31:260-264,1982.

[7] R.E Ladner, and M.J Fischer. Parallel Prefix Computation. *Journal of the ACM,* 27:831-838,1980

[8] P. Kogge, and H..Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trasanctions on Computers,* C-22:783-791,1973.

[9] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their synthesis*. PhD Thesis. Swiss Federal Institute of Technology, Zurich, 1997

[10] J. Grad, and J. E Stine. A sandard cell library for student projects. In *International Conference on Microelectronics Systems Education*, pages 98-99. IEEE Society press 2003