Multi-Operand Addition
Ivor Page[1]

8.1 Motivation

The motivation for multi-operand adders comes from the need for inner-product calculations and multiplication (summing the partial products). Inner products are used in computer graphics, robotics, and other geometric application, where linear transformations (rotation, scaling, translation) of millions of vertices are computed by multiplying their (x, y, z, w) points in homogeneous coordinate space by 4×4 matrices. The inner product is also the main operation on which domain transformations, digital filters and other signal processing applications are based.

Dot diagrams are used to show the array of bits to be added, as in Figure 1:

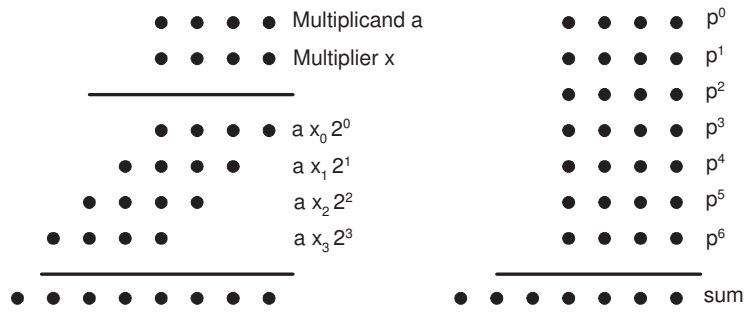

Figure 1: Four by Four Bit Multiply and the Sum of 7 Operands

In the diagrams, each dot represents a single bit.

The k partial products of the multiplier are easily formed by vectors of AND gates, $p^i = x_i.\langle a_{k-1} \cdots a_1, a_0 \rangle \times 2^i$, where $\langle a_{k-1} \cdots a_1, a_0 \rangle$ is the multiplicand and x_i is the i^{th} bit of the multiplier. All partial products are simultaneously available as the summation process begins. We will concentrate on multiple operand addition in this module and leave consideration of the summation of partial products to the module on multiplication.

8.2 Adder Tree

A fast addition of the multiple operands may be achieved by a binary tree of fast 2-input adders, as shown in Figure 2. Here, 6 operands are being added. The height of the tree for n operands is $\lceil log_2\ n \rceil$ and so the delay, if ripple-carry add units are used, is $O(nlog\ n)$.

[1]University of Texas at Dallas

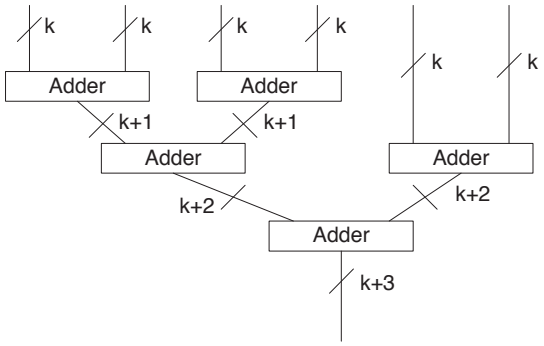

Figure 2: Binary tree of 2-input adders

Figure 3 shows the least significant stages of an array based on ripple-carry adders. Two input operands are added and the result p^0 is added to an intermediate result p^1.

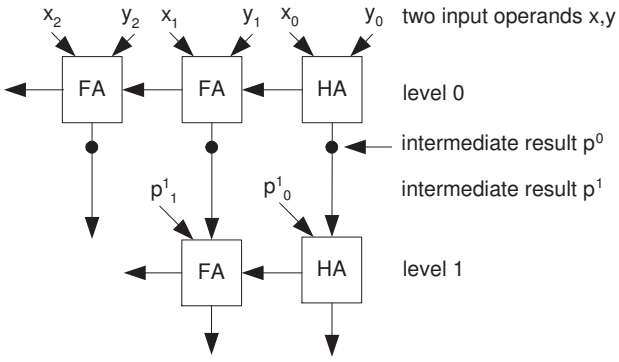

Figure 3: Least Significant Stages of the Ripple-Carry Arrays of Adders

8.3 Carry-Save Adders

The carry signals that propagate along the rows of the ripple-carry adders can instead be connected into the next row in a diagonal fashion. This strategy implements the *carry-save* technique. See Figure 4.

The delay of the array of carry-save adders is $O(log\ n)$ since the worst-case delay path corresponds to a carry signal that begins in the first row and ends in the last. The array does not, however, deliver the required sum, since the diagonal propagation of carry signals leaves two intermediate results at the bottom of the array, the sum outputs and the unresolved carries. An $O(log\ n)$ fast adder can add these and keep the total time to $O(log\ n)$.

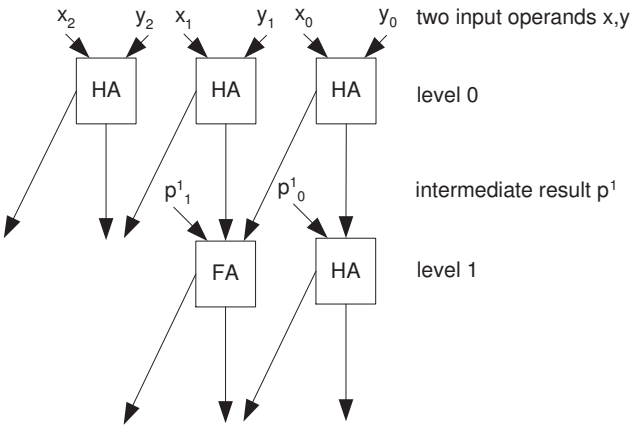

Figure 4: Upper-Right Corner of the Carry-Save Arrays of Adders

Each row of the CSA array reduces three inputs to two output operands. This observation leads to efficient tree structures, such as illustrated in Figure 5.

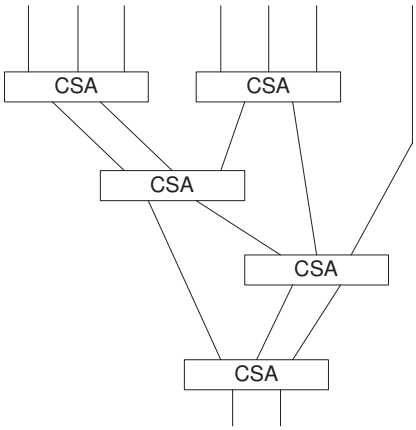

Figure 5: CSA tree for 7 operands

The corresponding dot diagram is given in Figure 6:

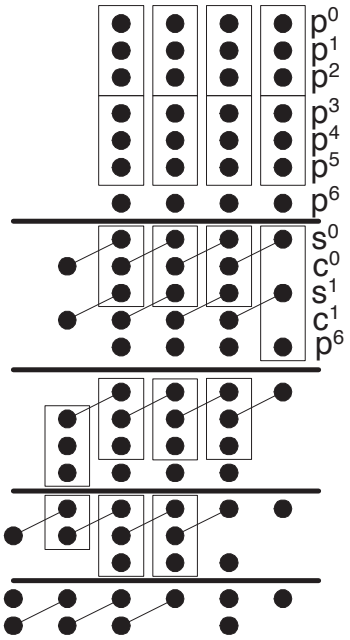

Figure 6: Dot Diagram for CSA tree for 7 operands

In Figure 6, rectangles are drawn around bit positions that are combined by full-adders and half-adders. The diagonal lines link the sum and carry outputs of the same adder cell and are present to help the reader understand the diagram.

The solid horizontal lines indicate breaks between the levels of the tree. The final two intermediate results are indicated by the two vectors of dots at the bottom of the dot diagram. These are added using a conventional fast 2-operand adder. The tree comprises 18 full-adders and one half-adder. The CSA units are all k-bits wide, as can be seen from the dot diagram of Figure 6 and the path widths indicated in Figure 7. The equivalent diagram in the text shows the actual bit positions of the inputs and outputs of the CSA units. The width of the carry-propagate adder at the bottom of the tree must be 5 bits according to Figure 6. An extra half-adder in the last level of the tree would reduce the CPA width to four bits.

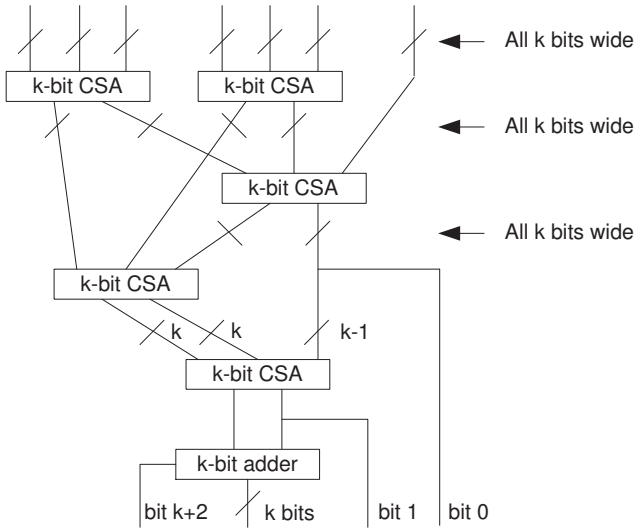

Figure 7: 7 operand CSA adder-array showing path and CSA widths

8.4 Wallace Trees

When CSA units are used to combine rows of bits as early as possible in the tree, a Wallace Tree is obtained. At every level of Figure 6, for each column of three dots, a full-adder combines them.

The CSA tree which reduces $n = 7$ k-bit operands to two $(k+2)$-bit operands is known as a 7-input Wallace Tree. The k-bit Wallace tree with n-inputs reduces its k-input bits to two bit vectors of length $(k + log_2\ n - 1)$.

Since each CSA reduces three input operands to two output operands, the height $h(n)$ of a Wallace tree for n inputs follows the recurrence relation:

$$h(n) = 1 + h(\lceil 2n/3 \rceil)$$

By removing the ceiling operator, we obtain a lower bound for the tree height:

$$h(n) \geq log_{1.5}(n/2)$$

Conversely, the maximum number of inputs $n(h)$ that can be added by a Wallace tree of height h is:

$$n(h) = \lceil 3n(h-1)/2 \rceil$$

which leads to the lower bound,

$$n(h) > 2(3/2)^{h-1}$$

The following table links the maximum number of inputs to tree heights 0 through 15.

h	n(h)	h	n(h)	h	n(h)	h	n(h)
0	2	4	9	8	42	12	211
1	3	5	13	9	63	13	316
2	4	6	19	10	94	14	474
3	6	7	28	11	141	15	711

8.5 Dadda Trees

In an n input Dadda tree, at the first level, the minimum number of CSA units is used to reduce the number of remaining operands to one of the values in the above table. For example, if we wish to add 32 input operands, the nearest value equal to or below 32 in the table is 28. Therefore 4 = 32-28 CSA units are needed in the first level. The numbers of operands in subsequent levels reduce according to the values in the table: 19, 13, 9, 6, 4, 3, 2. The corresponding numbers of CSA units at the levels are 4, 9, 6, 4, 3, 2, 1, 1. The height of a Dadda tree is the same as that of a Wallace tree and, for n aligned operands, the number of CSA units and their widths are the same as for a Wallace tree.