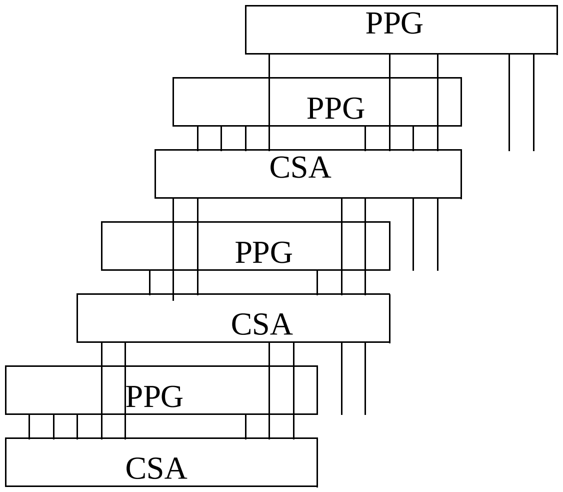
Carry Save Adder Trees in Multipliers

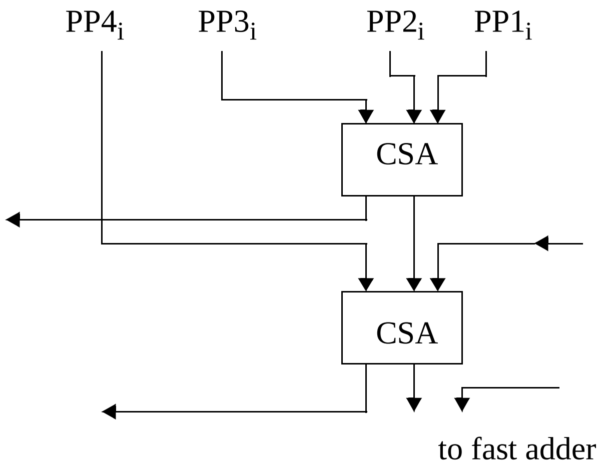
A naive organization for adding the partial products together in an array multiplier is to accumulate the partial products in each row and pass the accumulated partial products (using carry save adders) to the next row



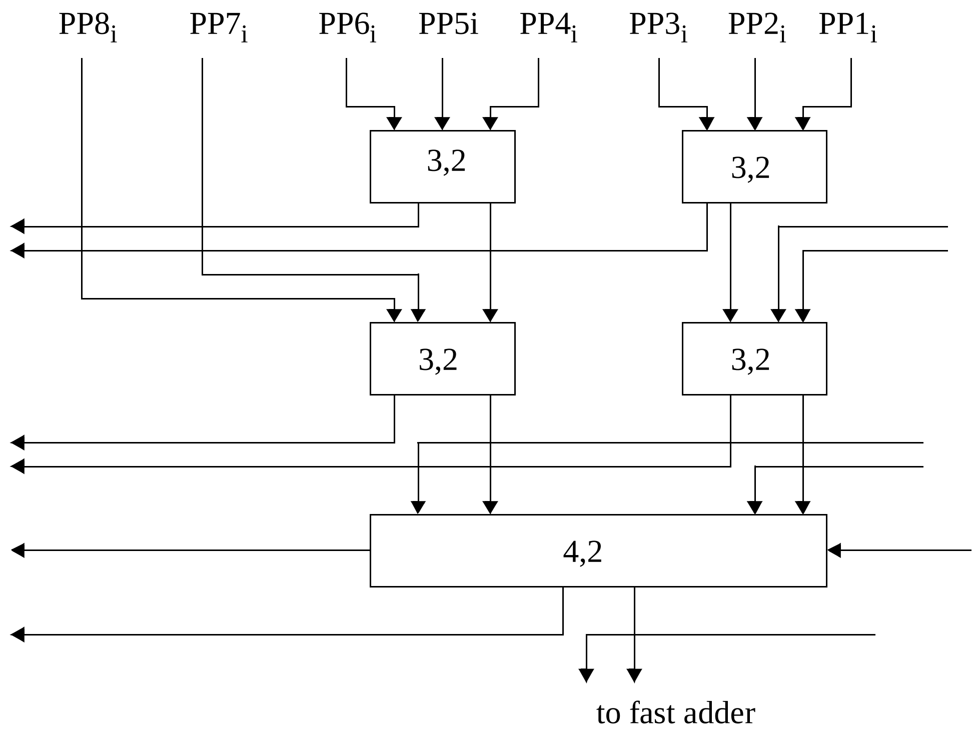
This makes the total delay going through the carry save proportional to N, the number of bits in the data words (there are N rows or N/2 rows if Booth recoding is used.) This delay is too long if we use a fast adder for the last row which as delay ~ log N.

The easiest way to reduce the CSA delay is to make a tree structure instead of the linearly connected naive design. In general, we want an efficient way to add N (or N/2) partial products. Let us examine the problem of adding 1 bit of each of the N products (this is called a “bit-slice”.

Example 4 partial products (8x8 Booth)

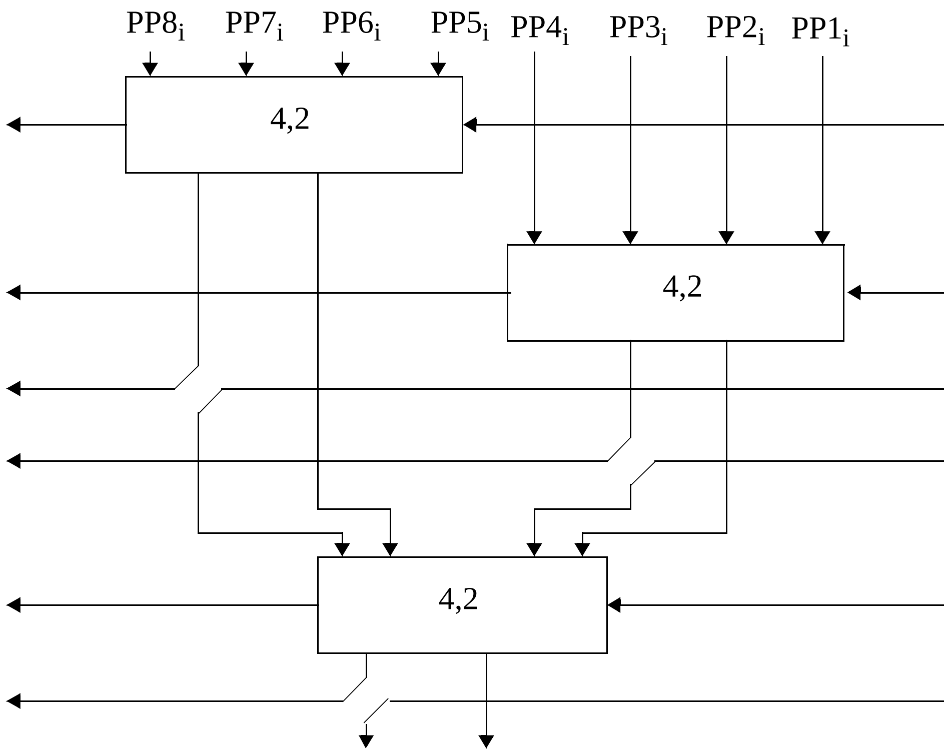


The horizontal lines indicate that the carry out must be passed to the next most significant bit slice. Since we make each bit slice the same, there will also be a corresponding carry-in. A tree of carry save adders is called a "Wallace tree". Note that only 2 levels of CSA' are needed in the Wallace tree for 4 partial products whereas the linear structure requires 3 levels. There is an even greater savings for larger multipliers. For example 8 partial products (16x16 Booth)

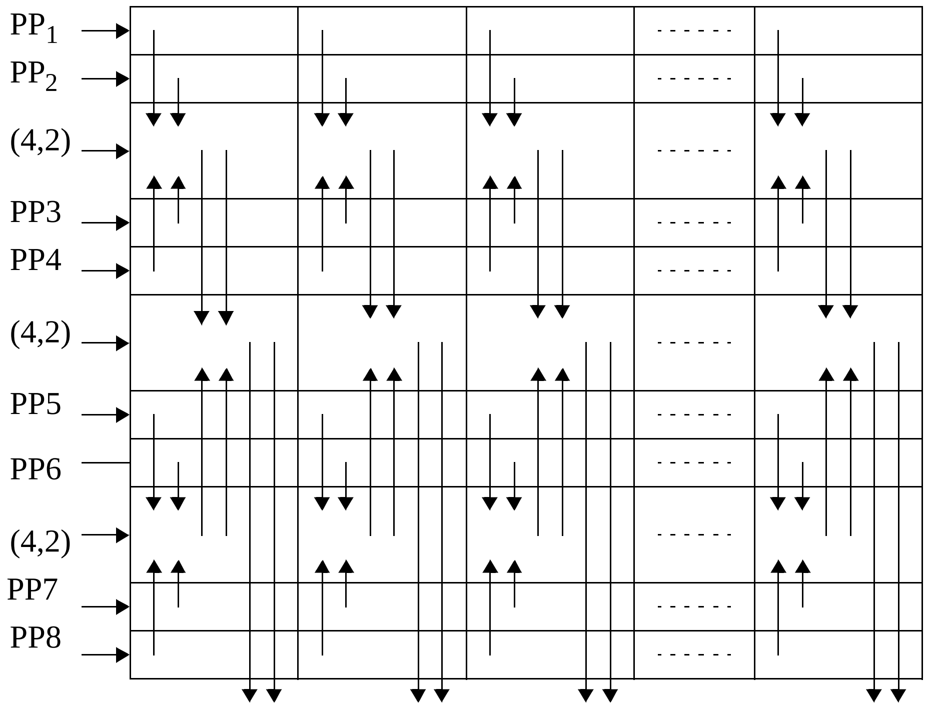


We have now used a notation where the CSA is regarded as a general element adding three inputs and putting out two binary weighted outputs. Thus, the (4,2) element is simply the design we already did for the 4 partial product example with four equally weighted inputs and two binary weighted outputs. The (4,2) element has recently become a popular building block for CSA trees. This is because there are 1/2 as many outputs as inputs to the (4,2) element. Thus, binary trees may be constructed which make the layout problem somewhat easier.

Example (8,2) out of (4,2) blocks



If we decompose each (4,2) block into (3,2) elements, there are 2 stages of (3,2) elements in each (4,2) block. This gives a total of 4 stages for 8 partial products in both designs (same delay) which is much better than the 7 required for the linear structure. The layout for the tree with (4,2) building blocks is probably more straightforward than using (3,2) elements directly as building blocks. Remember, each bit slice of the tree must be repeated N times. Thus, the layout of the (8,2) structure from (4,2) building blocks might look like:



Note that the tree interconnect must be provided for each bit slice. Note also that the carries do not ripple through the (4,2) blocks in the horizontal direction (the horizontal carry out is independent of the horizontal carry-in). Rather, the critical delay path follows the vertical tree structure interconnect. Finally, note that although there are 3 (4,2) blocks, the delay from any PP to an output goes through only $\underline{\underline{2}}$ of the (4,2) blocks. If we extend this design to larger numbers or partial products, the delay is proportional to $\log_2$ (no. of PP's). This is the basis for the high speed of tree-based designs.

Higher Order Building Blocks

The carry save adder can be regarded as the most general circuit that takes some number of inputs and represents the sum of the inputs in $\underline{\underline{2}}$ binary weighted output bits. We can extend this idea into larger building blocks that represent binary weighted sums in 3, 4 or more bits.

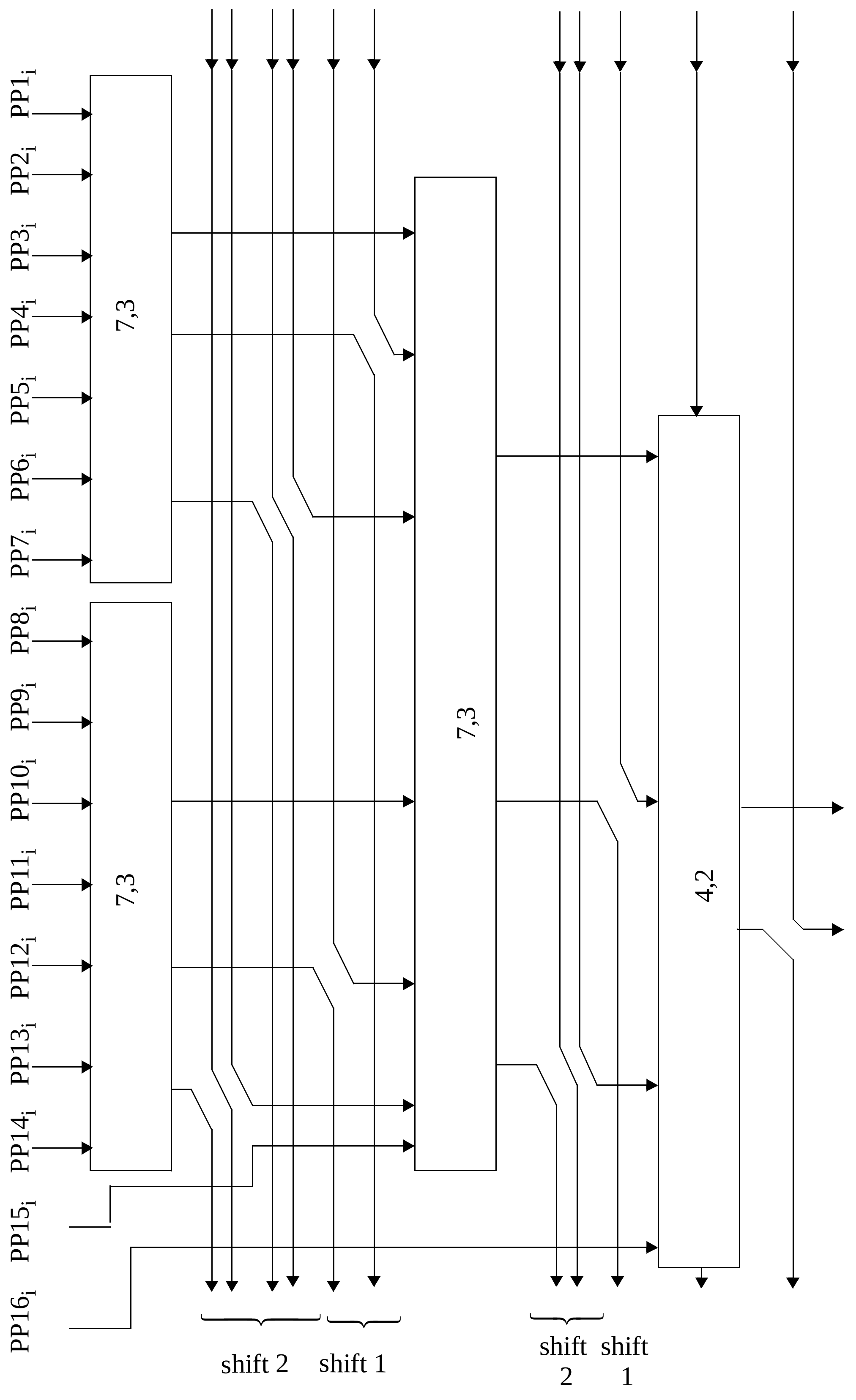
(3,2): most inputs in 2 output bits

(7,3): most inputs in 3 output bits

(15,4): most inputs in 4 output bits

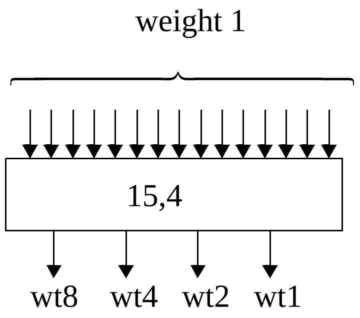
These high order building blocks can greatly simplify global routing.

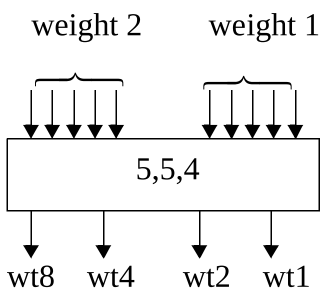
For example (16,2) out of (7,3) blocks.



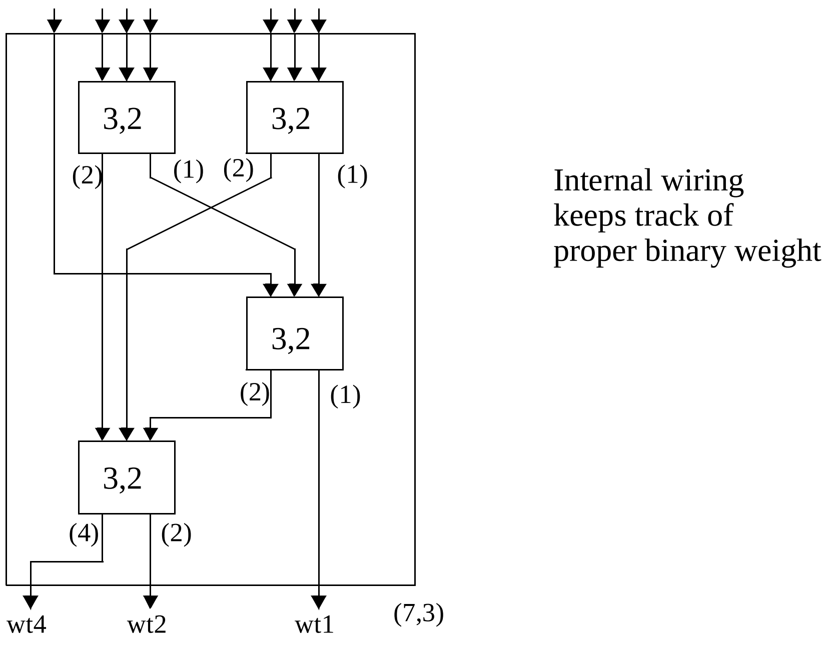
Such a diagram is much harder to draw out of (3,2) blocks, and the routing is correspondingly more complicated. Use of the (7,3) block organizes the routing and makes it more efficient even if the (7,3) block is constructed from (3,2) blocks! (More on this later).

A similar organization is obtained using (15,4) as a building block. However, it is difficult to easily route the $\underline{\underline{15}}$ inputs to this single block (it requires numerous feed throughs in the partial product generator!). The number of inputs can be reduced by making some of the inputs have higher binary weight. For example, a (5,5,4) block is an alternative to the (15,4) block. The (5,5,4) block requires 50% less feed throughs than the (15,4) block.





One might try to implement the large building blocks by developing logic equations for each output and synthesizing the corresponding logic gates. This results is complex logic functions with a high fan out from the circuit inputs (the same input occurs in many different terms of the logic equation). This causes the implementation to be too slow and take up too much area. The estimated the no. of FET's ~ (no. of inputs)^4! It seems more efficient to implement the larger building blocks out of smaller building blocks. If this process is repeated for the smaller blocks, then ultimately all the blocks can be made from the (3,2) element. For example

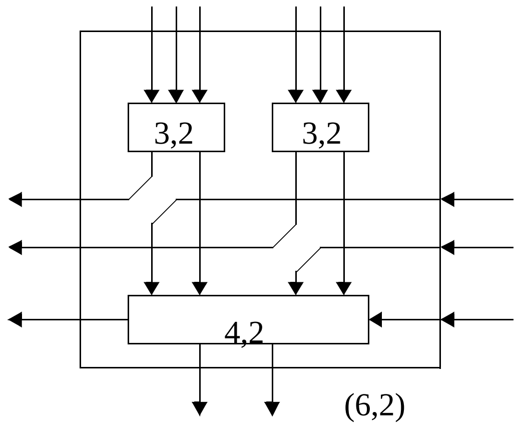


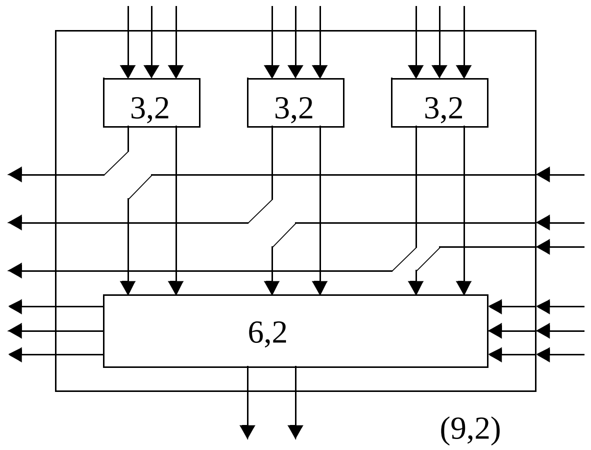
We still gain efficiency in the design by using the (7,3) block since it

1. simplifies global routing
2. allows re-use of the layout and routing already done for the (7,3) when multiple (7,3) are used.

The higher order building blocks discussed so far do not have any “hidden” carries like the (4,2) block discussed earlier. It is also possible to make higher order blocks with “hidden” carry-outs that do not depend on the “hidden” carry-ins.

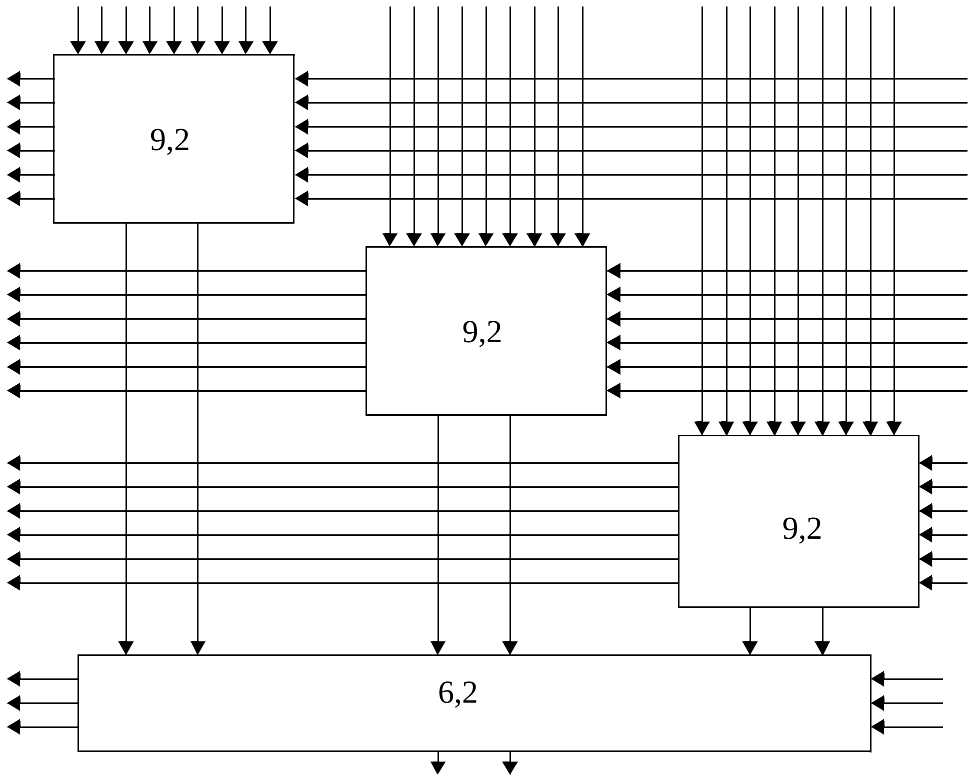
A (6,2) block and a (9,2) block can be made.





This idea can be extended to arbitrarily large blocks. Note that none of the carries “ripple” across from right to left. Rather the carries propagate down and to the left.

One should try to choose a block size or hierarchy of block sizes that fits the intended design. For example, consider a design for the 53-bit by 53-bit multiplier for the mantissa of the IEEE 744 floating point standard. First, Booth recording is used to reduce the number of partial products to 27. The needed (27,2) tree can be implemented as follows.



Since 27 is a multiple of 9, the (9,2) building block gives a very simple global routing structure for this multiplier. But it would not be appropriate for a 16 or 32 bit multiplier for example.