BINARY MULTIPLICATION USING PARTIALLY REDUNDANT MULTIPLES

Gary Bewick
Michael J. Flynn

Technical Report No. CSL-TR-92-528

June 1992

The work described by this report was supported by NSF under contract MIP88-22961

BINARY MULTIPLICATION USING PARTIALLY REDUNDANT MULTIPLES

by

Gary Bewick

Michael J. Flynn

Technical Report No. CSL-TR-92-528

June 1992

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

This report presents an extension to Booth's algorithm for binary multiplication. Most implementations that utilize Booth's algorithm use the 2 bit version, which reduces the number of partial products required to half that required by a simple add and shift method. Further reduction in the number of partial products can be obtained by using higher order versions of Booth's algorithm, but it is necessary to generate multiples of one of the operands (such as 3 times an operand) by the use of a carry propagate adder. This carry propagate addition introduces significant delay and additional hardware. The algorithm described in this report produces such difficult multiples in a partially redundant form, using a series of small length adders. These adders operate in parallel with no carries propagating between them. As a result, the delay introduced by multiple generation is minimized and the hardware needed for the multiple generation is also reduced, due to the elimination of expensive carry lookahead logic.

Key Words and Phrases: multiplication, Booth's algorithm, redundant multiples, computer arithmetic

Contents

List of Figures

1 Introduction

Multiplication is a basic arithmetic operation that is important in microprocessors, digital signal processing, and other modern electronic machines. VLSI Designers have recognized this importance by dedicating significant resources and area to integer and floating point multipliers. As a result, it is desirable to reduce the cost of these multipliers by using efficient algorithms that do not compromise performance. This report describes an extension to Booth's algorithm for binary multiplication that reduces the cost of such high performance multipliers.

To explain this scheme, background material on existing algorithms is presented. These algorithms are then extended to produce the new method. Since the method is somewhat complex, this report will attempt to stay away from implementation details, but instead concentrate on the algorithms in a hardware independent manner.

In order that the basic algorithms are not obscured with small details, unsigned multiplication only will be considered here, but the algorithms presented are easily generalized to deal with signed numbers.

2 Background

2.1 Add and Shift

The first and simplest method of multiplication to be addressed is the *add and shift* multiplication algorithm. This algorithm conditionally adds together copies of the multiplicand (partial products) to produce the final product, and is illustrated in Figure 1, for a 16 x 16 multiply. Each dot is

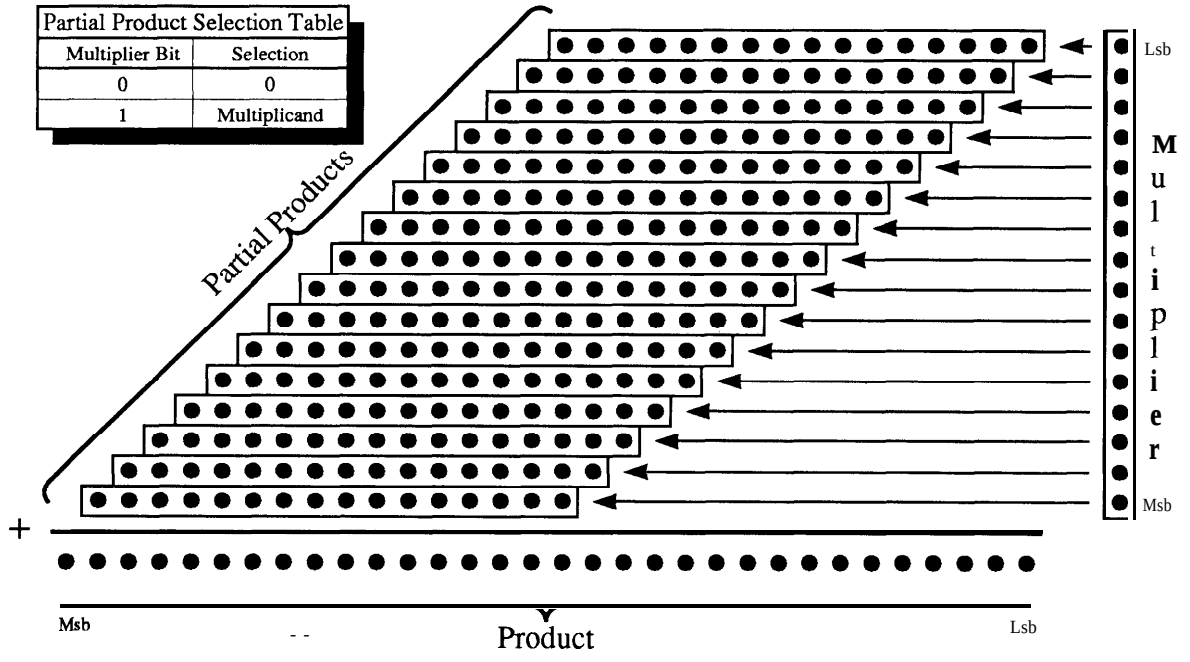


Figure 1: 16 bit add and shift multiply

a placeholder for a single bit which can be a zero or one. Each horizontal row of dots represents a single copy of the multiplicand, M, (i.e. one partial product), which is conditioned upon a particular bit of the multiplier . The conditioning algorithm is shown in the selection table in the upper left hand corner of the figure. The multiplier itself is represented on the right edge of the figure, with the least significant bit at the top. The final product is represented by the row of 32 horizontal dots at the bottom of the figure.

An example of the add and shift algorithm using actual numbers is shown in Figure 2.

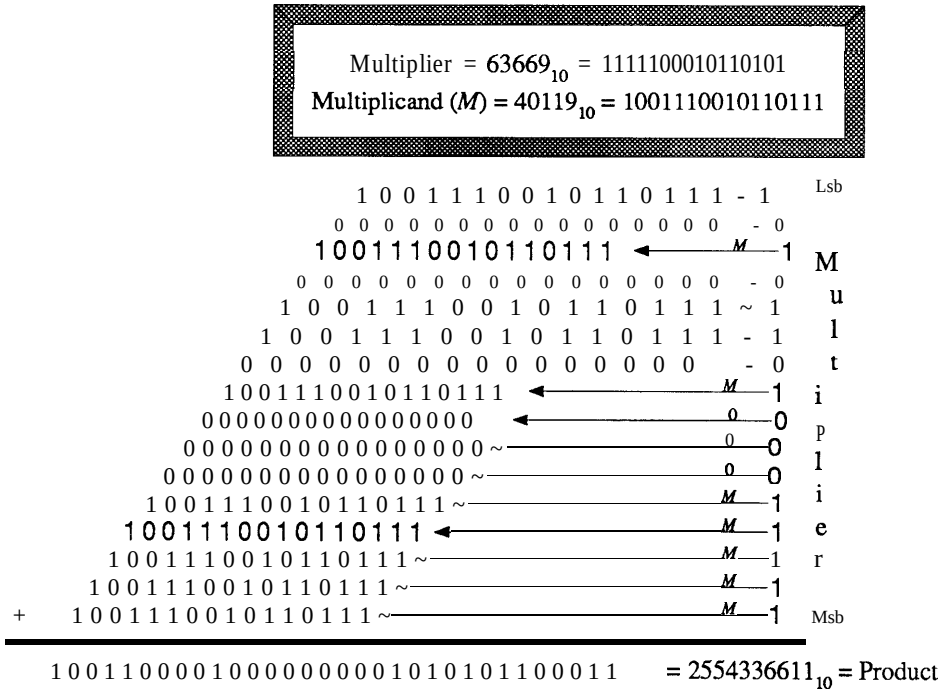


Figure 2: **16** bit add and shift example

2.1.1 Dot Diagrams

The *dot diagram,* as Figure 1 is referred to, can provide information which can be used as a guide for examining various multiplication algorithms. Roughly speaking, the number of dots (256 for Figure 1) in the partial product section of the dot diagram is proportional to the amount of hardware required (time multiplexing can reduce the hardware requirement, at the cost of slower operation [4]) to sum the partial products and form the final product.

The latency of an implementation of a particular algorithm is also related to the height of the partial product section (i.e the maximum number of dots in any vertical column) of the dot diagram. This relationship can vary from logarithmic (tree implementation where interconnect delays are insignificant) to linear (array implementation where interconnect delays are constant) to something in between (tree implementations where interconnect delays are significant). Implementations are not being considered here, so the only conclusion is that a smaller height ought to be faster.

Finally, the logic which selects the partial products can be deduced from the partial product selection table. For the add and shift algorithm, the logic is particularly simple and is shown in

Figure 3. This figure shows the selection logic for a single partial product (a single row of

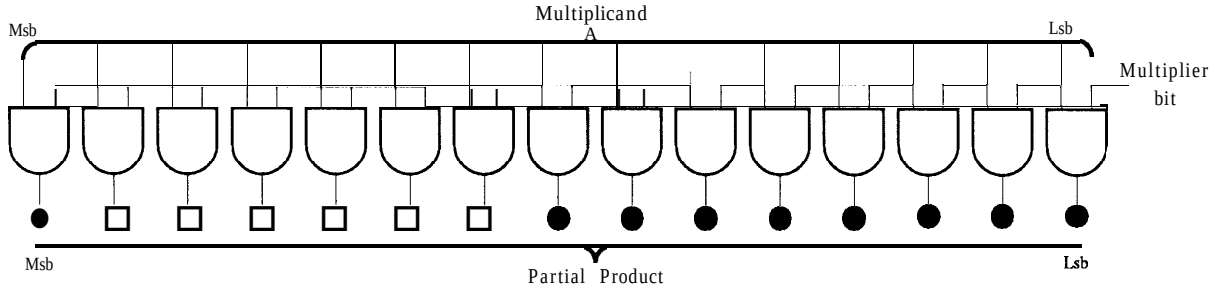


Figure 3: Partial product selection logic for add and shift

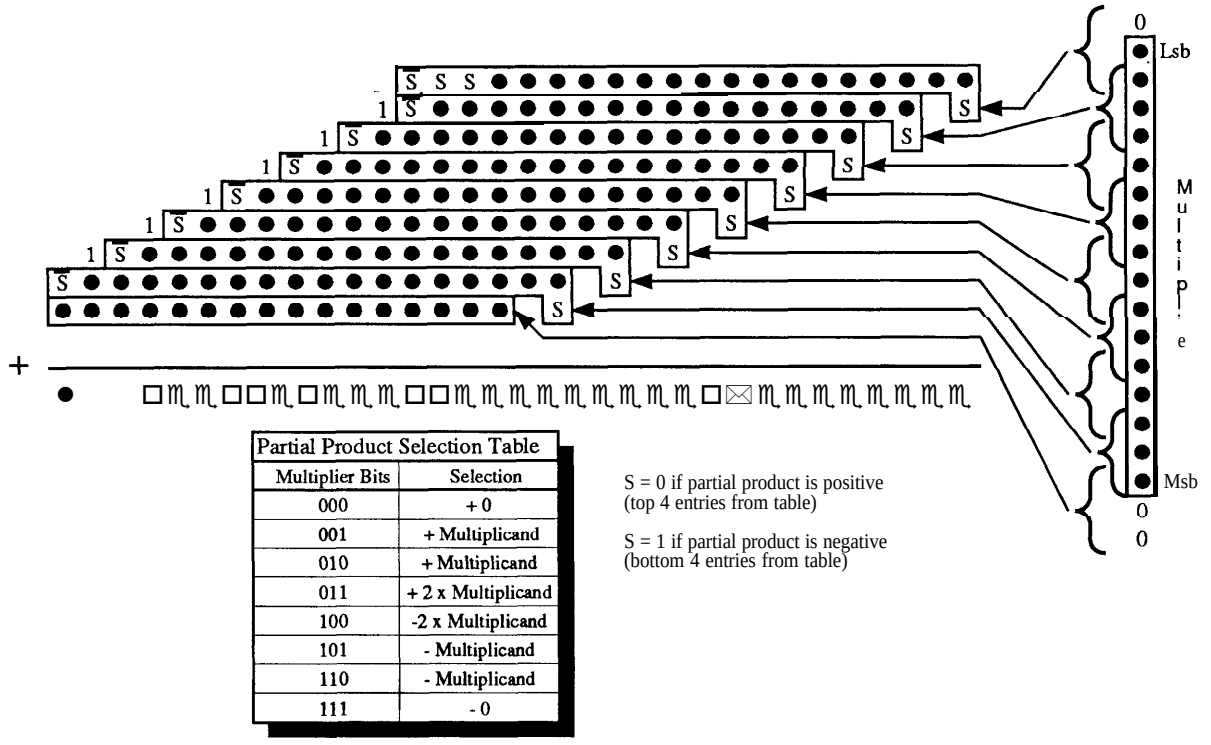
dots). Frequently this logic can be merged directly into whatever hardware is being used to sum the partial products. This can reduce the delay of the logic elements to the point where the extra time due to the selection elements can be ignored. However, **a** real implementation will still have interconnect delay due to the physical separation of the common inputs of each AND gate, and distribution of the multiplicand to the selection elements.

2.2 Booth's Algorithm

Since fewer dots can be faster and require less hardware, how can the number (and height) of dots be reduced? A common method is to use *Booth's Algorithm* [1]. Hardware implementations commonly use a slightly modified version of Booth's algorithm, referred to appropriately as *Modified Booth's Algorithm* [2]. Figure 4 shows the dot diagram for a 16 x 16 multiply using the 2 bit version of this algorithm (Booth 2). The multiplier is partitioned into overlapping groups of 3 bits, and each group is decoded to select a single partial product as per the selection table. Each partial product is shifted 2 bit positions with respect to it's neighbors. The number of partial products has been reduced from 16 to 9. In general the there will be $\left\lfloor \frac{n+2}{2} \right\rfloor$ partial products, where n is the operand length. The various required multiples can be obtained by a simple shift of the multiplicand (these are *easy* multiples). Negative multiples, in 2's complement form, can be obtained using a bit by bit complement of the corresponding positive multiple, with a 1 added in at the least significant position of the partial product (the S bits along the right side of the partial products). An example multiply is shown in Figure 5. The number of partial products to be added has been reduced from 16 to 9 vs. the add and shift algorithm. In addition, the number of dots has decreased from 256 to 177 (this includes sign extension and constants). This reduction in dot count is not a complete saving – the partial product selection logic is more complex (Figure S). In fact, depending on actual implementation details, the extra cost and delay due to the more complex partial product selection logic may overwhelm the savings due to the reduction in the number of dots [3].

2.3 Booth 3

Shift amounts between adjacent partial products of greater than 2 are also possible [2], with a corresponding reduction in the height and number of dots in the dot diagram. A 3 bit Booth dot diagram is shown in Figure 7, and an example is shown in Figure 8. Each partial product could be from the set $\{\pm 0, \pm M, \pm 2M, \pm 3M, \pm 4M\}$. All multiples with the exception of 3M are



Partial Product Selection Table

Multiplier Bits	Selection
000	+ 0
001	+ Multiplicand
010	+ Multiplicand
011	+ 2 x Multiplicand
100	-2 x Multiplicand
101	- Multiplicand
110	- Multiplicand
111	- 0

Figure 4: 16 bit Booth 2 multiply

easily obtained by simple shifting and complementing of the multiplicand. The number of dots, constants, and sign bits to be added is now 126 (for the 16 x 16 example) and the height of the partial product section is now 6.

Generation of the multiple 3M (referred to as a *hard* multiple, since it cannot be obtained via simple shifting and complementing of the multiplicand) generally requires some kind of carry propagate adder to produce. This carry propagate adder may increase the latency, mainly due to the long wires that are required for propagating carries from the less significant to more significant bits. Sometimes the generation of this multiple can be overlapped with an operation which sets up the multiply (for example the fetching of the multiplier).

Another drawback to this algorithm is the complexity of the partial product selection logic, an example of which is shown in Figure 9, along with the extra wiring needed for routing the 3M multiple.

2.4 Booth *4* and higher

A further reduction in the number and height in the dot diagram can be made, but the number of hard multiples required goes up exponentially with the amount of reduction. For example the Booth 4 algorithm (Figure 10) requires the generation of the multiples $\{\pm 0, \pm M, \pm 2M, \pm 3M, \pm 4M, \pm 5M, \pm 6M, \pm 7M, \pm 8M\}$. The hard multiples are 3M (6M can be obtained by shifting 3M), 5M and 7M. The formation of the multiples can take place in parallel, so the extra cost mainly involves the adders for producing the multiples, and the additional wires that are needed to route the various multiples around.

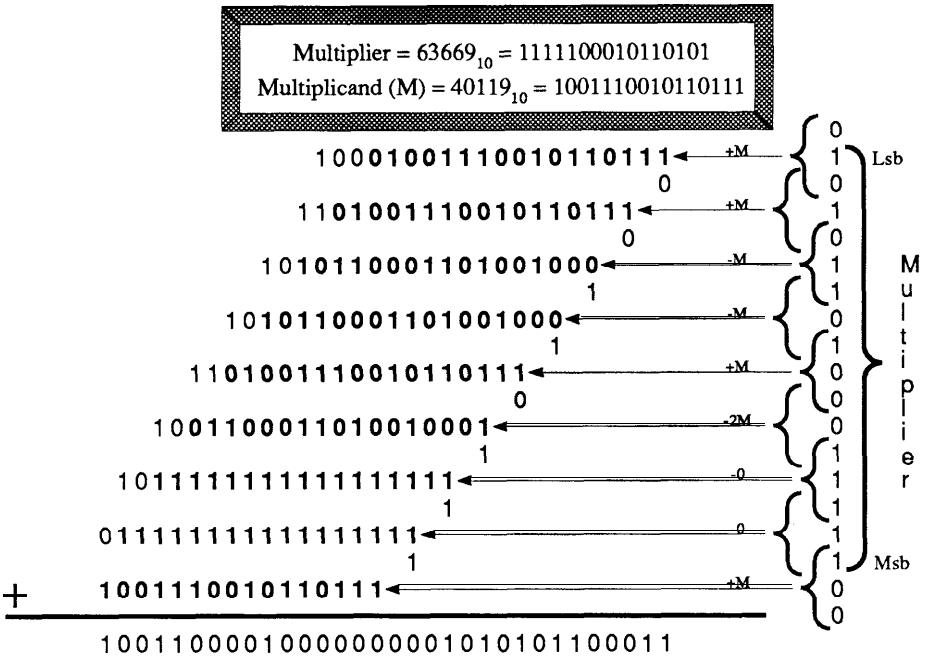


Figure 5: 16 bit Booth 2 Example

3 Redundant Booth

This section presents a new variation on the Booth 3 algorithm, which eliminates much of the delay and part of the hardware associated with the multiple generation, yet produces a dot diagram which can be made to approach that of the conventional Booth 3 algorithm. Before introducing this variation, a simple and similar method (but is not particularly hardware efficient) is explained. This method is then extended to produce the new variation. Methods of further generalizing to a Booth 4 algorithm are then discussed.

3.1 Booth 3 with fully redundant partial products

The time consuming carry propagate addition that is required for the higher Booth algorithms can be eliminated by representing the partial products in a fully redundant form. This method is illustrated by examining the Booth 3 algorithm, since it requires the fewest multiples. A fully redundant form represents an n bit number by two n – 1 bit numbers whose sum equals the number it is desired to represent (there are other possible redundant forms see [5]). For example the decimal number 14568 can be represented in redundant form as the pair (14568,0), or (14567,1), etc. Using this representation, it is trivial to generate the 3M multiple required by the Booth 3 algorithm, since 3M = 2M + 1M, and 2M and 1M are easy multiples. The dot diagram for a 16 bit Booth 3 multiply using this redundant form for the partial products is shown in Figure 11 (an example appears in Figure 12). The dot diagram is the same as that of the conventional Booth 3 dot diagram, but each of the partial products is twice as high, giving roughly twice the number of dots and twice the height. Negative multiples are obtained by the same method as the previous Booth algorithms – bit by bit complementation of the corresponding positive multiple with a 1 added at the lsb. Since every partial product now consists of two numbers, there will be two 1s added at the

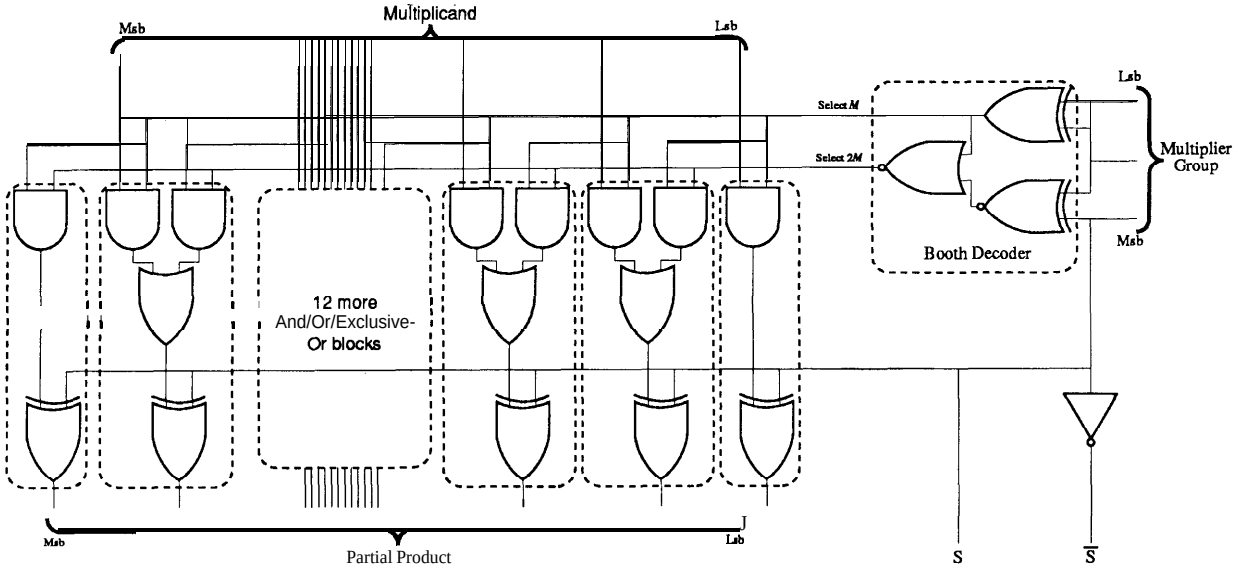
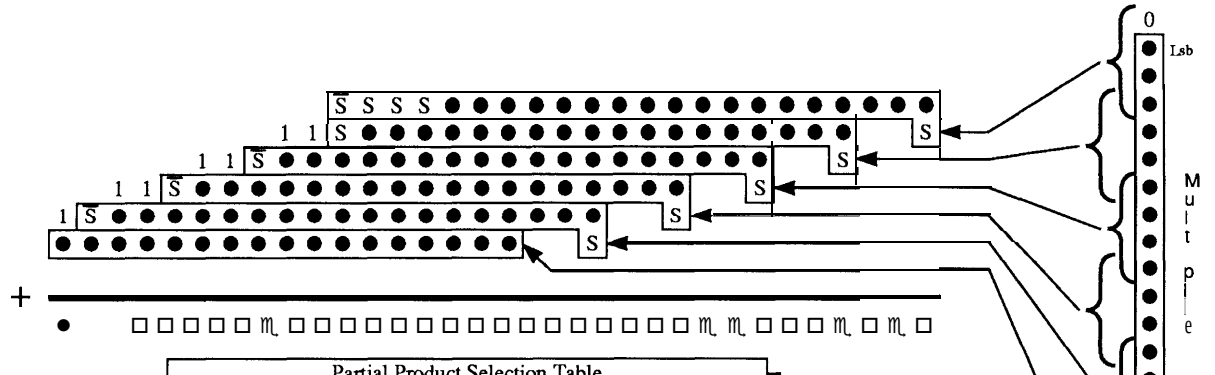


Figure 6: 16 bit Booth 2 Partial Product Selector Logic



Partial Product Selection Table

Multiplier Bits	Selection	Multiplier Bits	Selection
0000	+ 0	1000	-4 x Multiplicand
0001	+ Multiplicand	1001	-3 x Multiplicand
0010	+ Multiplicand	1010	-3 x Multiplicand
0011	+2 x Multiplicand	1011	-2 x Multiplicand
0100	+2 x Multiplicand	1100	-2 x Multiplicand
0101	+3 x Multiplicand	1101	- Multiplicand
0110	+3 x Multiplicand	1110	- Multiplicand
0111	+4 x Multiplicand	1111	- 0

Figure 7: 16 bit Booth 3 multiply

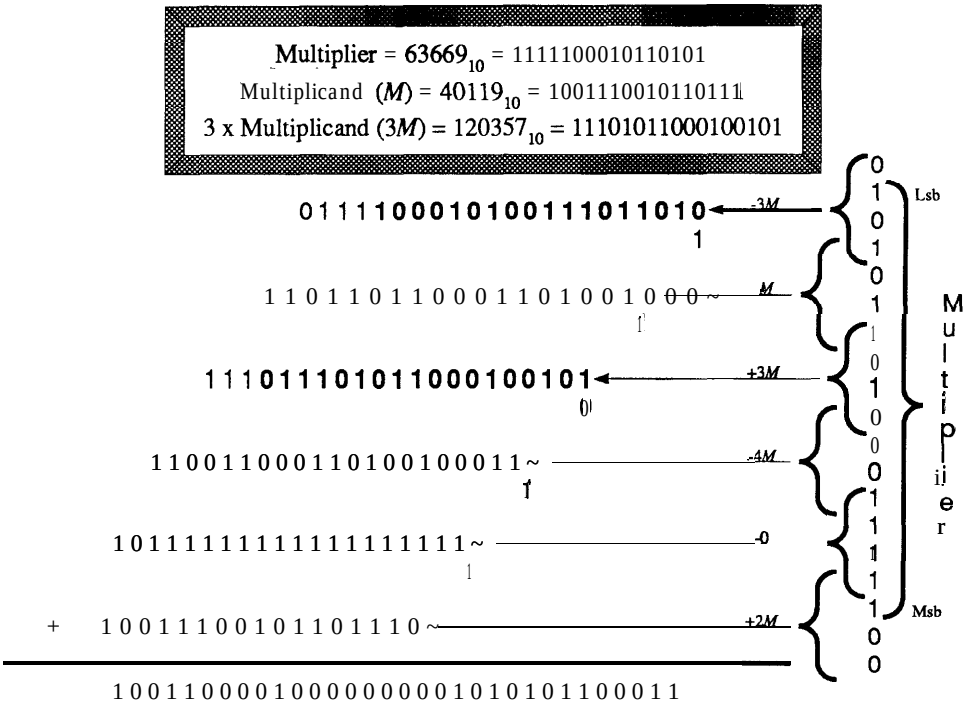


Figure 8: 16 bit Booth 3 Example

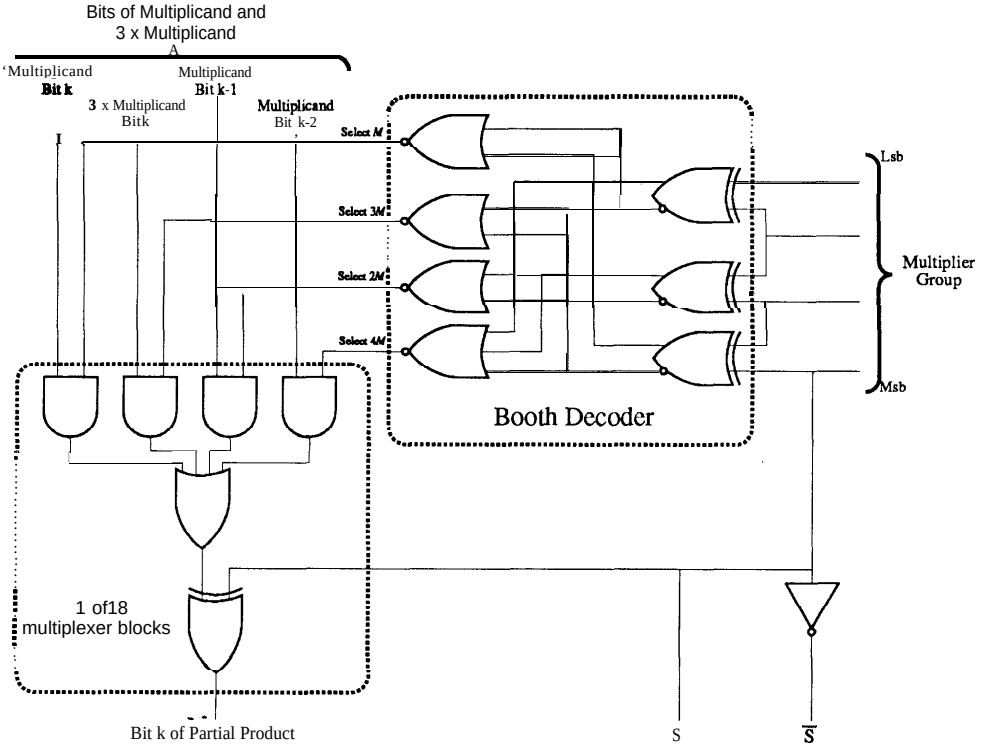


Figure 9: 16 bit Booth 3 Partial Product Selector Logic

Partial Product Selection Table							
Multiplier Bits	Selection	Multiplier Bits	Selection	Multiplier Bits	Selection	Multiplier Bits	Selection
00000	+ 0	01000	+4 x Multiplicand	10000	-8 x Multiplicand	11000	-4 x Multiplicand
00001	+ Multiplicand	01001	+5 x Multiplicand	10001	-7 x Multiplicand	11001	-3 x Multiplicand
00010	+ Multiplicand	01010	+5 x Multiplicand	10010	-7 x Multiplicand	11010	-3 x Multiplicand
00011	+2 x Multiplicand	01011	+6 x Multiplicand	10011	-6 x Multiplicand	11011	-2 x Multiplicand
00100	+2 x Multiplicand	01100	+6 x Multiplicand	10100	-6 x Multiplicand	11100	-2 x Multiplicand
00101	+3 x Multiplicand	01101	+7 x Multiplicand	10101	-5 x Multiplicand	11101	- Multiplicand
00110	+3 x Multiplicand	01110	+7 x Multiplicand	10110	-5 x Multiplicand	11110	- Multiplicand
00111	+4 x Multiplicand	01111	+8 x Multiplicand	10111	-4 x Multiplicand	11111	- 0

Figure 10: Booth 4 Partial Product Selection Table

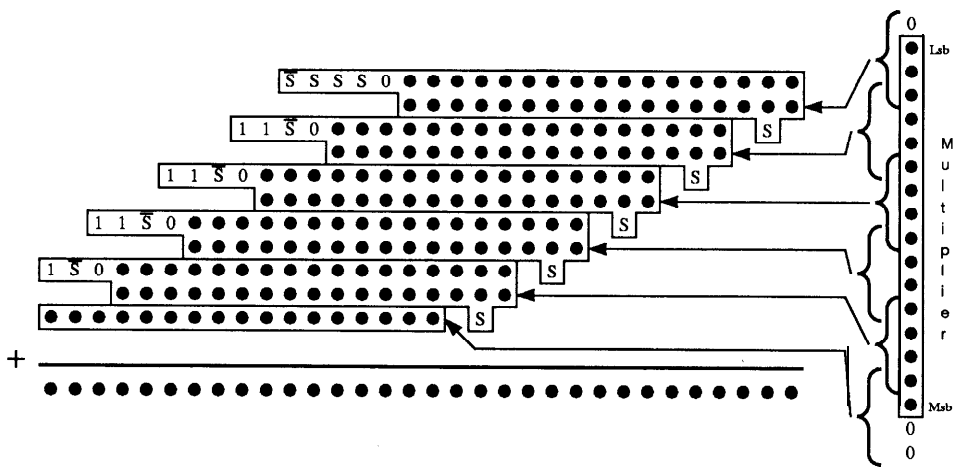


Figure 11: 16 x 16 Booth 3 multiply with fully redundant partial products

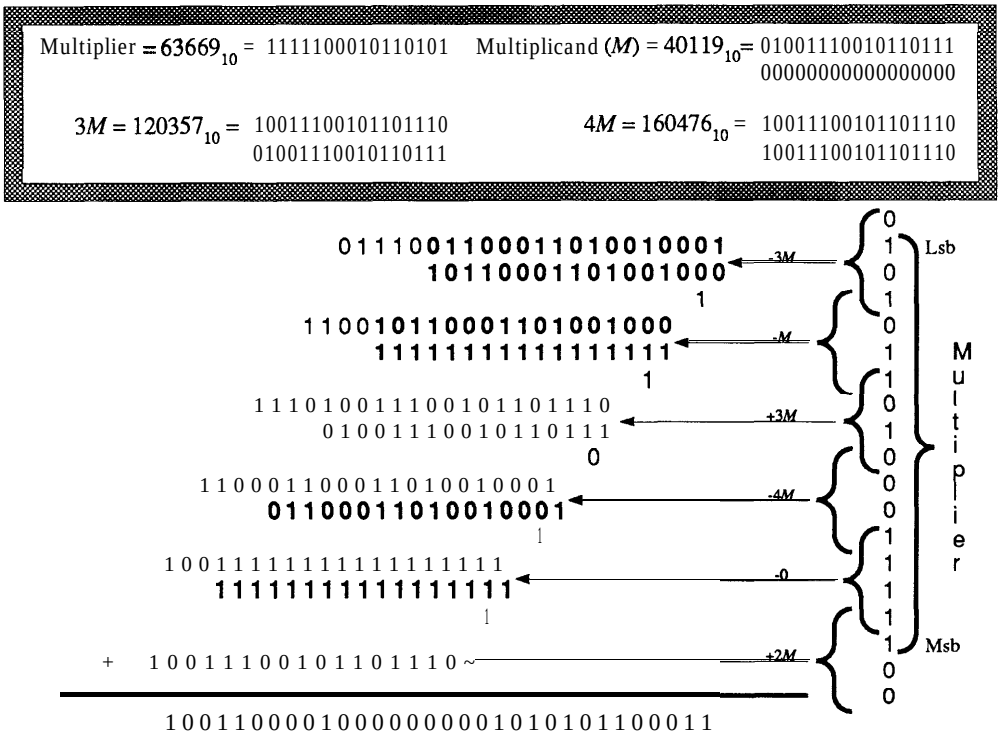


Figure 12: 16 bit fully redundant Booth 3 example

lsb, which can be combined into a single 1 which is shifted to the left one position.

Although this algorithm is not particularly attractive, due to the doubling of the number of dots in each partial product, it provides a stepping stone to a related, efficient algorithm.

3.2 **Booth 3 with partially redundant partial products**

The conventional Booth 3 algorithm assumes that the 3M multiple is available in non-redundant form. Before the partial products can be summed, a time consuming carry propagate addition is needed to produce this multiple. The Booth 3 algorithm with fully redundant partial products avoids the carry propagate addition, but has the equivalent of twice the number of partial products to sum. The new scheme tries to combine the smaller dot diagram of the conventional Booth 3 algorithm, with the ease of the hard multiple generation of the fully redundant Booth 3 algorithm.

The idea is to form the 3M multiple in a *partially redundant* form by using a series of small length adders, with no carry propagation between the adders (Figure 13). If the adders are of sufficient length, the number of dots per partial product can approach the number in the non-redundant representation. This reduces the number of dots needing summation. If the adders are small enough, then carries are not propagated across large distances, and are faster than a full carry propagate adder. Also, less hardware is required due to the elimination of the logic which propagates carries between the small adders. There is a design tradeoff which must be resolved here.

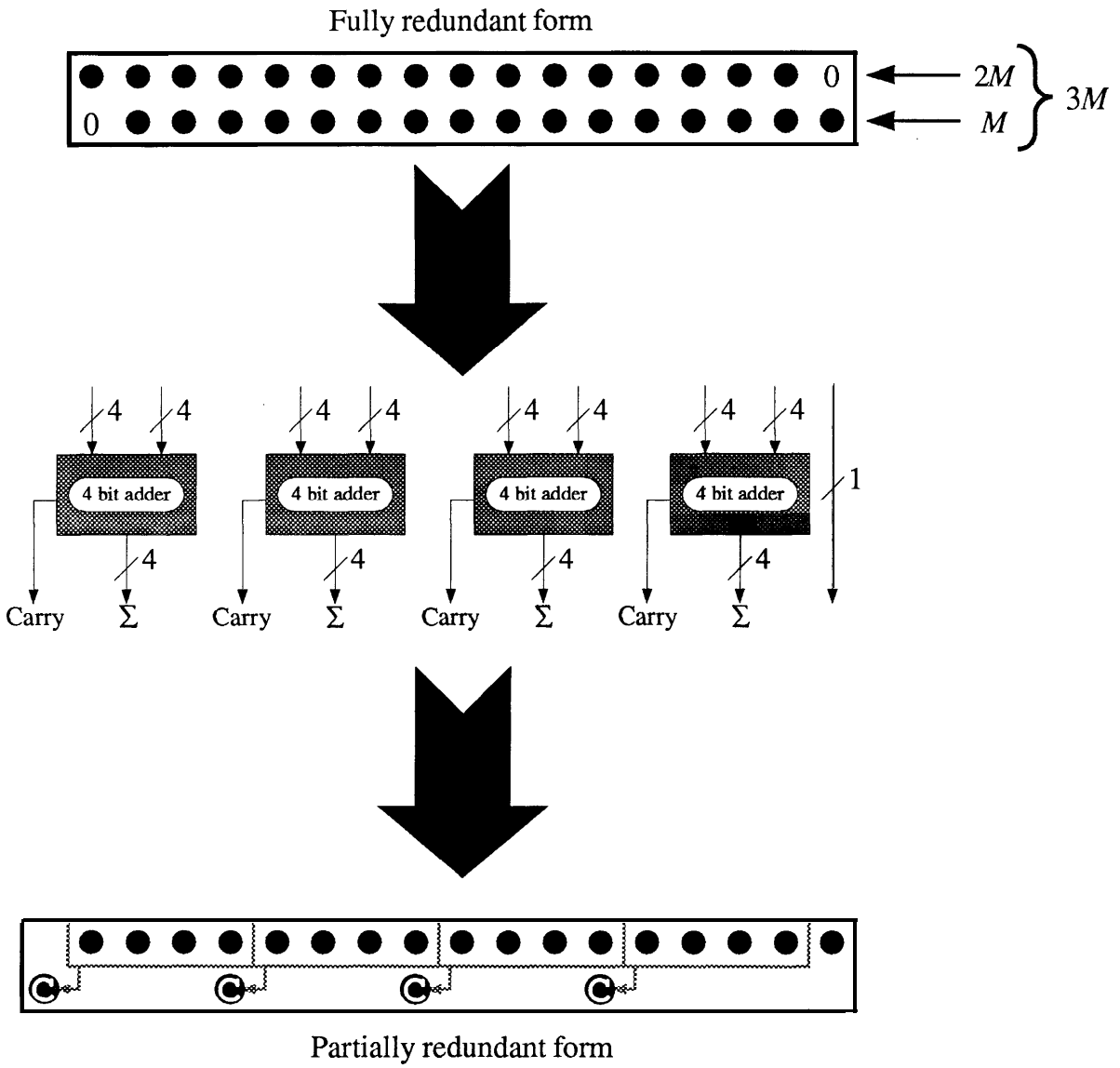


Figure 13: Computing 3M in a partially redundant form

3.2.1 Dealing with negative partial products

There is a difficulty with the partially redundant representation described by Figure 13. Recall that Booth's algorithm requires the negative of all multiples. The negative (2's complement) can normally be produced by a bit by bit complement, with a 1 added in at the lsb of the partial product. If this procedure is done to a multiple in partially redundant form, then the large gaps of zeros in the positive multiple become large gaps of ones in the negative multiple (see Figure 14). In the worst case (all partial products negative), summing the partially redundant partial products

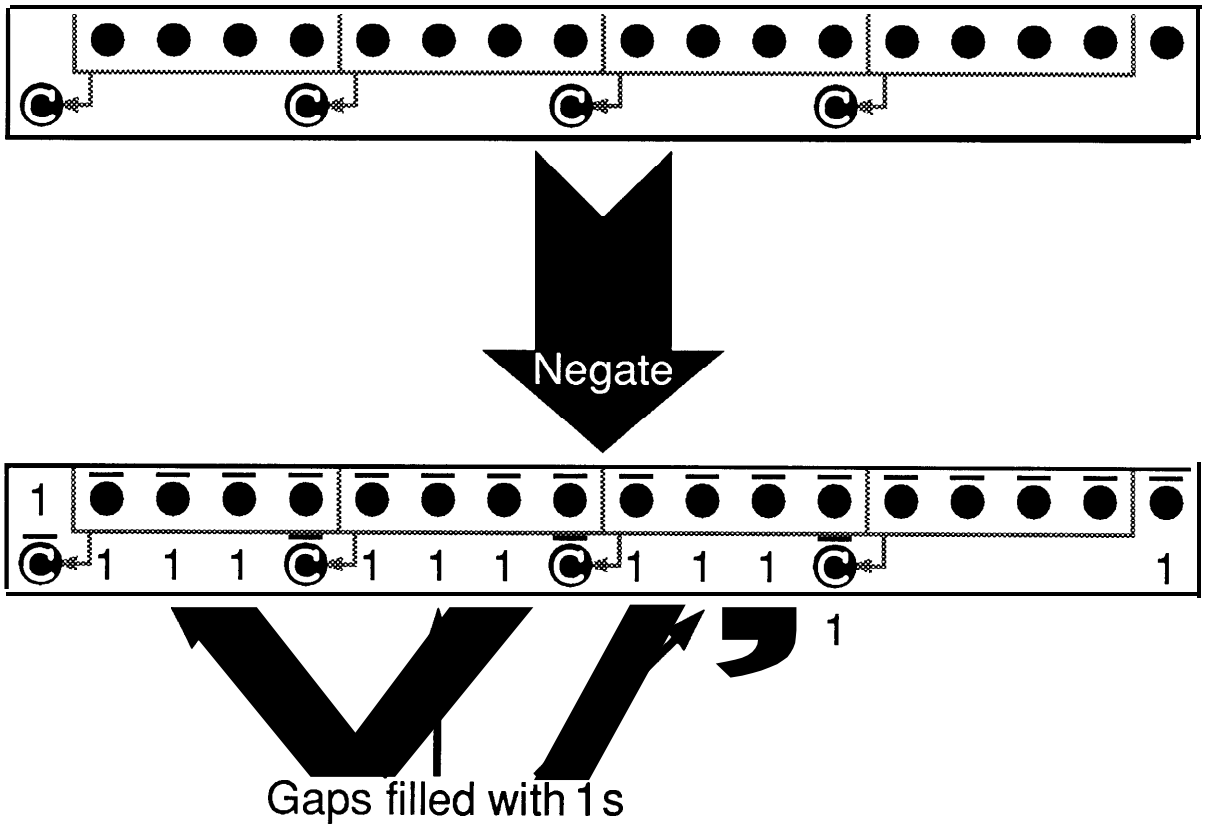
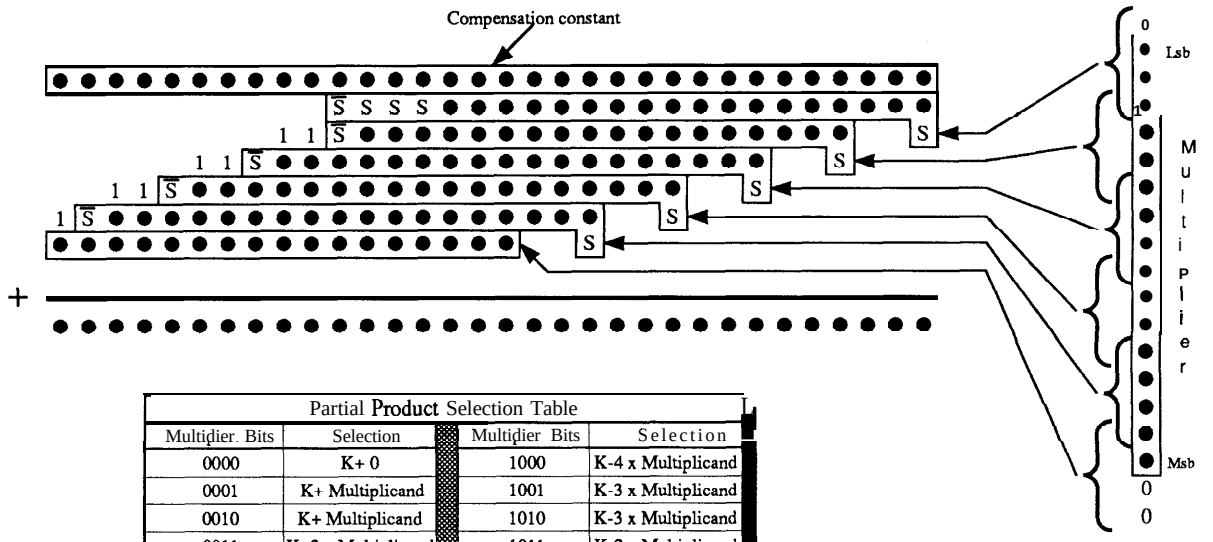


Figure 14: Negating a number in partially redundant form

requires as much hardware as representing them in the fully redundant form. The problem then is to find a partially redundant representation which has the same form for both positive and negative multiples, and allows easy generation of the negative multiple from the positive multiple (or vice versa). The simple form used in Figure 13 cannot meet both of these conditions simultaneously.

3.3 Booth with bias

In order to produce multiples in the proper form, Booth's algorithm needs to be modified slightly. This modification is shown in Figure 15. Each partial product has a bias constant added to it before being summed to form the final product. The bias constant (K) is the same for both positive and negative multiples. The bias constants may be different for each partial product. The only restriction is that K, for a given partial product, cannot depend on the which particular multiple



Partial Product Selection Table

Multiplier Bits	Selection	Multiplier Bits	Selection
0000	K+ 0	1000	K-4 x Multiplicand
0001	K+ Multiplicand	1001	K-3 x Multiplicand
0010	K+ Multiplicand	1010	K-3 x Multiplicand
0011	K+2 x Multiplicand	1011	K-2 x Multiplicand
0100	K+2 x Multiplicand	1100	K-2 x Multiplicand
0101	K+3 x Multiplicand	1101	K- Multiplicand
0110	K+3 x Multiplicand	1110	K- Multiplicand
0111	K+4 x Multiplicand	1111	K- 0

Figure 15: Booth 3 with bias

is selected for use in producing the partial product. With this assumption, the constants for each partial product can be added (at design time!) and the negative of this sum added to the partial products (the *Compensation constant).* The net result is that zero has been added to the partial products, so the final product is unchanged.

3.3.1 Choosing the right constant

Now consider a multiple in the partially redundant form of Figure 13 and choose a value for K such that there is a 1 in the positions where a "C" dot appears and zero elsewhere, as shown in the top part of Figure 16. Notice the topmost circled section enclosing 3 vertical items (two dots and the constant 1). These items can be summed as per the middle part of the figure, producing the dots "X" and "Y". The three items so summed can be replaced by the equivalent two dots, shown in the bottom part of the figure, to produce a redundant form for the sum of K and the multiple. This is very similar to the simple redundant form described earlier, in that there are large gaps of zeros in the multiple. The key advantage of this form is that the value for K – Multiple can be obtained very simply from the value of K + Multiple.

Figure 17 shows the sum of K+Multiple with a value Z which is formed by the bit by bit complement of the non-blank portions of K + Multiple and the constant 1 in the lsb. When these two values are summed toget her, the result is 2K (this assumes proper sign extension to however many bits are desired). That is :

$$K + Multiple + Z = 2K$$

$$Z = K - Multiple$$

In short, K – Multiple can be obtained from K + Multiple by complementing all of the non-blank

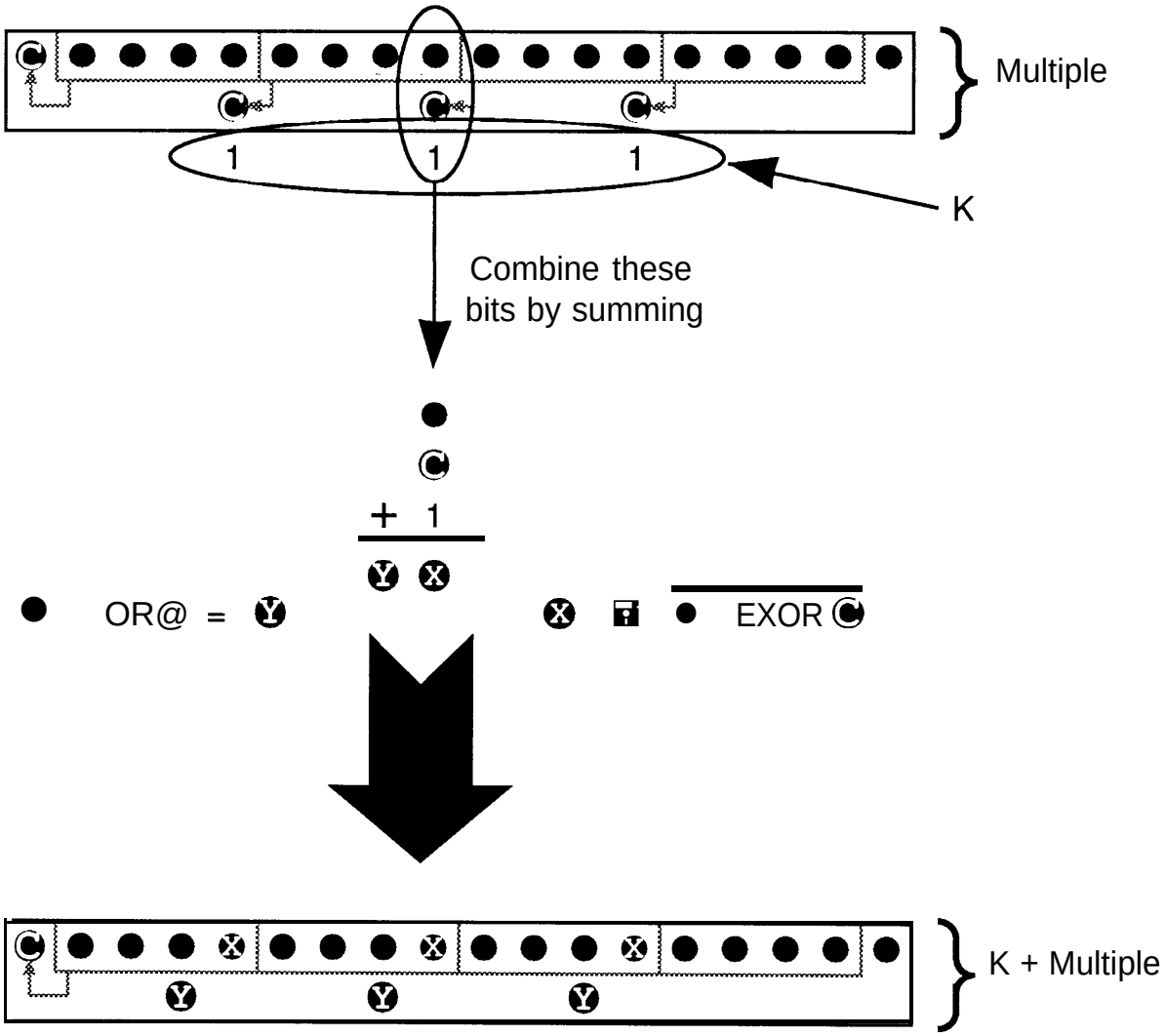


Figure 16: Transforming the simple redundant form

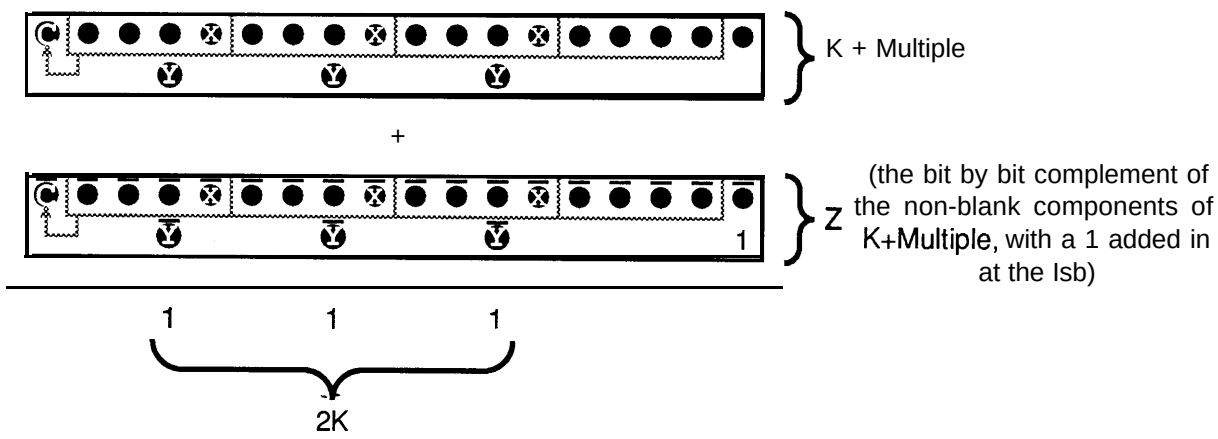


Figure 17: Summing K – Multiple and Z

bits of K + Multiple and adding 1. This is exactly the same procedure used to obtain the negative of a number when it is represented in its non-redundant form.

This partially redundant form satisfies the two conditions presented earlier, that is it has the same representation for both positive and negative multiples, and also it is easy to generate the negative given the positive form (the entries from the right side of the table in Figure 15 will continue to be considered as negative multiples).

3.3.2 Producing the multiples

Figure 18 shows in detail how the biased multiple K + 3M is produced from M and 2M using 4 bit adders and some simple logic gates. The simple logic gates will not increase the time needed to

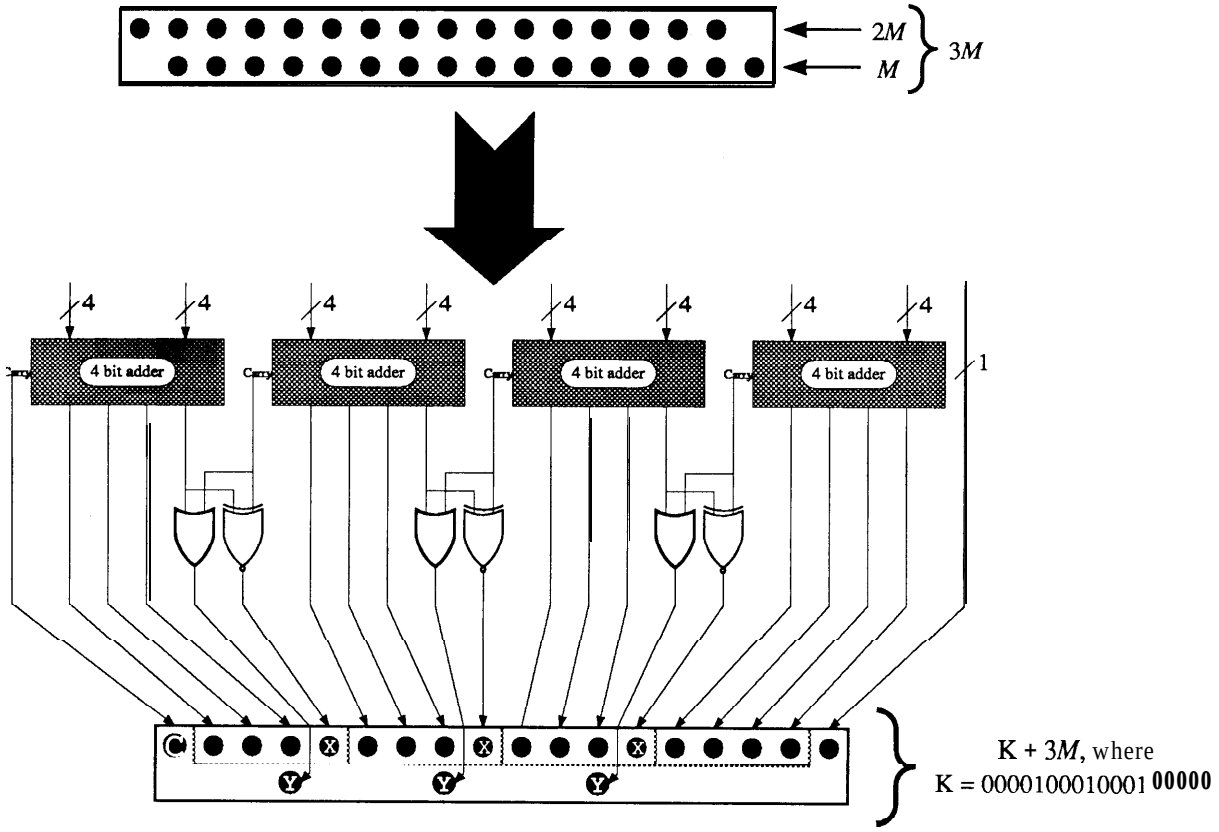


Figure 18: Producing K + 3M in partially redundant form

produce the biased multiple if the carry-out and the least significant bit from the small adder are available early. This is usually easy to assure. The other required biased multiples are produced by simple shifting and inverting of the multiplicand as shown in Figure 19. In this figure the bits of the multiplicand (M) are numbered (lsb = 0) so that the source of each bit in each multiple can be easily seen.

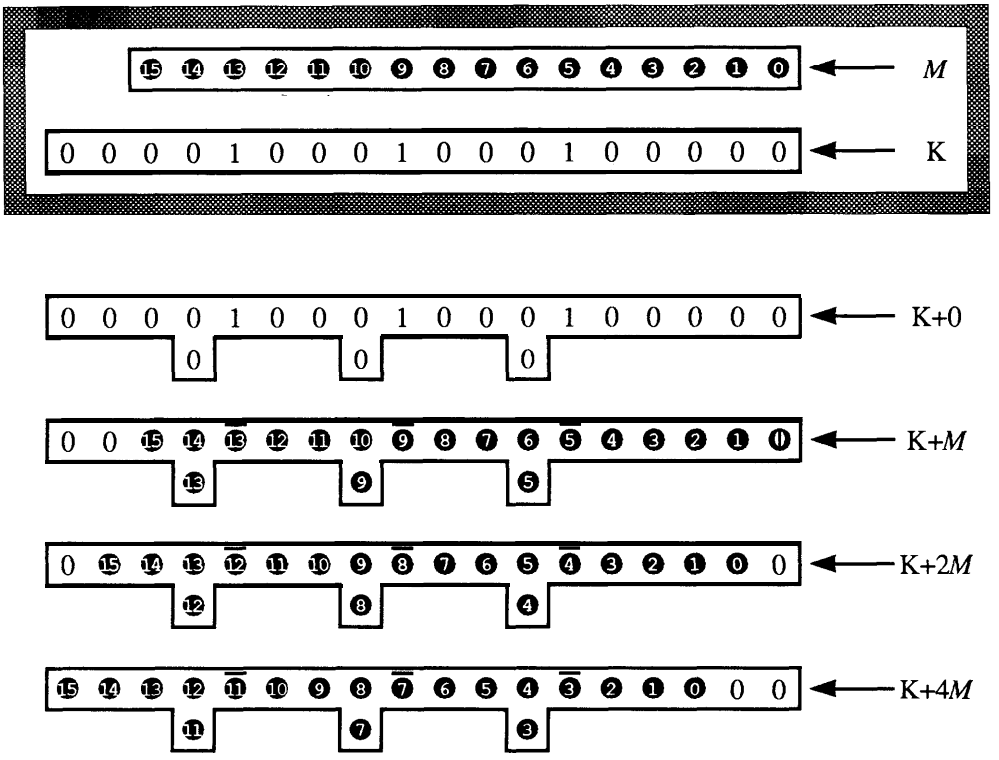


Figure 19: Producing other multiples

3.4 Redundant Booth 3

Combining the partially redundant representation for the multiples with the biased Booth 3 algorithm provides a workable redundant Booth 3 algorithm. The dot diagram for the complete redundant Booth 3 algorithm is shown in Figure 20 for a 16 x 16 multiply. The compensation constant has been computed given the size of the adders used to compute the K + 3M multiple (4 bits in this case). There are places where more than a single constant is to be added (on the left hand diagonal). These constants could be merged into a single constant to save hardware. Ignoring this merging, the number of dots, constants and sign bits in the dot diagram is 155, which is slightly more than that for the non-redundant Booth 3 algorithm (previously given as 126). The height [1] is 7, which is one more than that for the Booth 3 algorithm. Each of these measures are less than that for the Booth 2 algorithm (although the cost of the small adders is not reflected in this count).

A detailed example for the redundant Booth 3 algorithm is shown in Figure 21. This example uses 4 bit adders as per Figure 18 to produce the multiple K + 3M. All of the multiples are shown in detail at the top of the figure.

The partial product selectors can be built out of a single multiplexer block, **as** shown in Figure 22. This figure shows how a single partial product is built out of the multiplicand and K + 3M generated by logic in Figure 18.

The diagram indicates a single column (20) with height 8, but this can be reduced to 7 by manipulation of the S bits and the compensation constant.

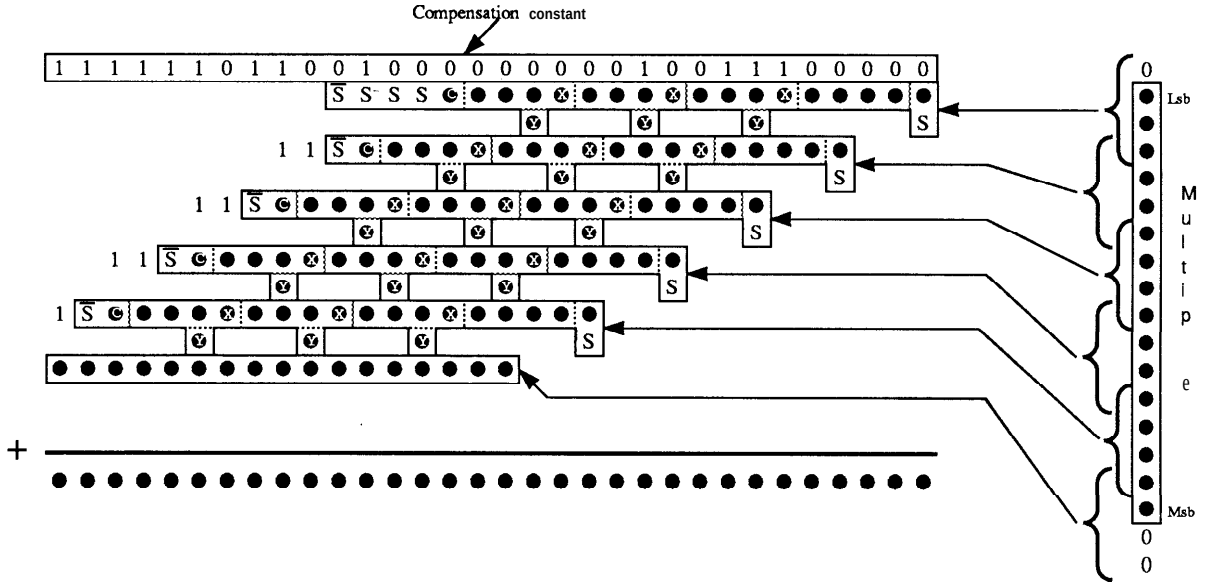


Figure 20: 16 x 16 Redundant Booth 3

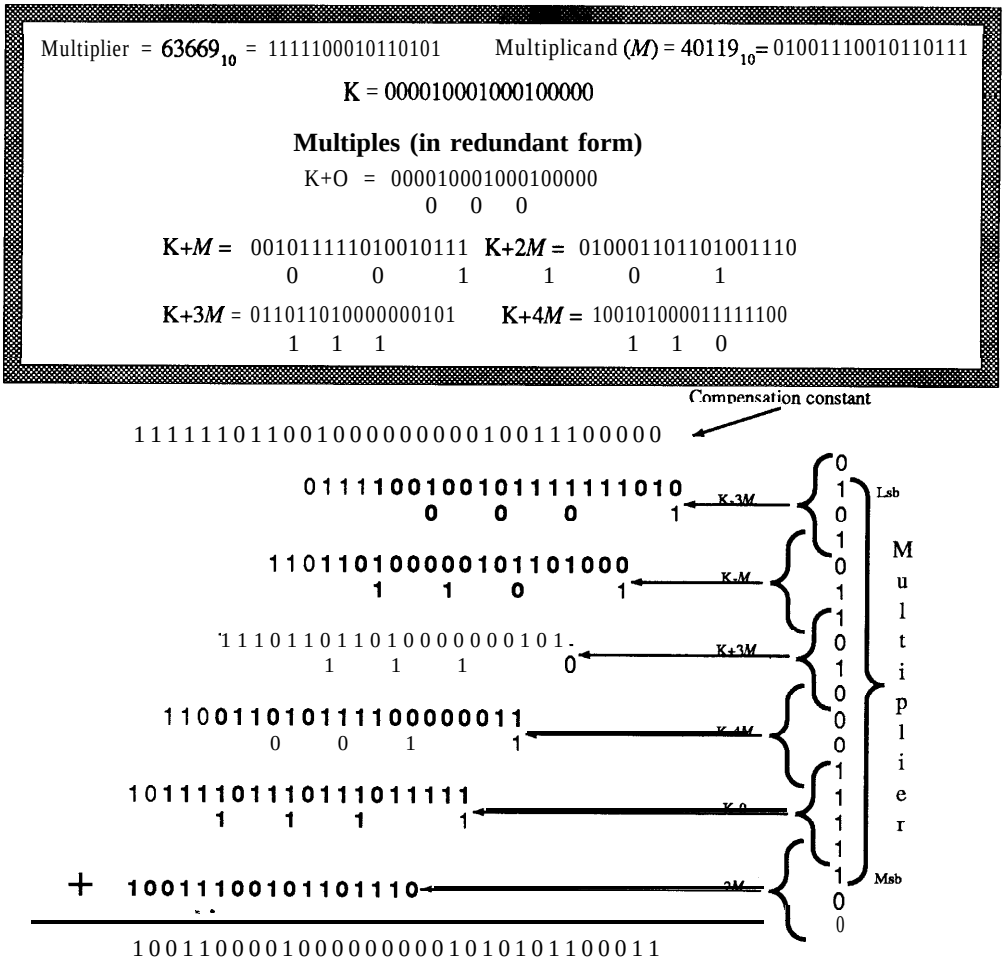


Figure 21: 16 bit partially redundant Booth 3 multiply

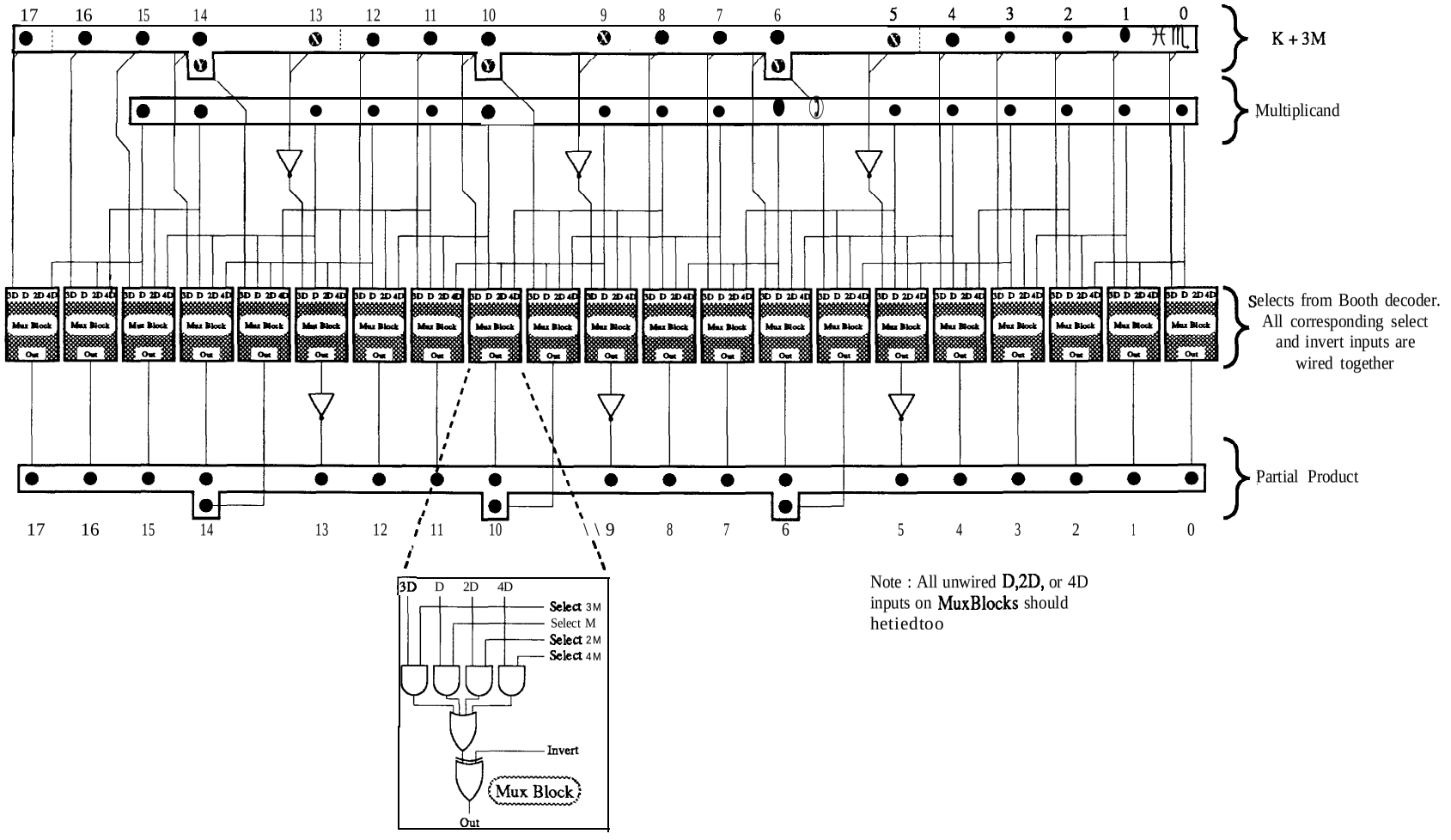


Figure 22: Partial product selector for redundant Booth 3

3.5 Redundant Booth 4

At this point, a possible question is "Can this scheme be adapted to the Booth 4 algorithm". The answer is yes, but it is not particularly efficient and probably is not viable. The difficulty is outlined in Figure 23 and is concerned with the biased multiples 3M and 6M. The left side of the figure

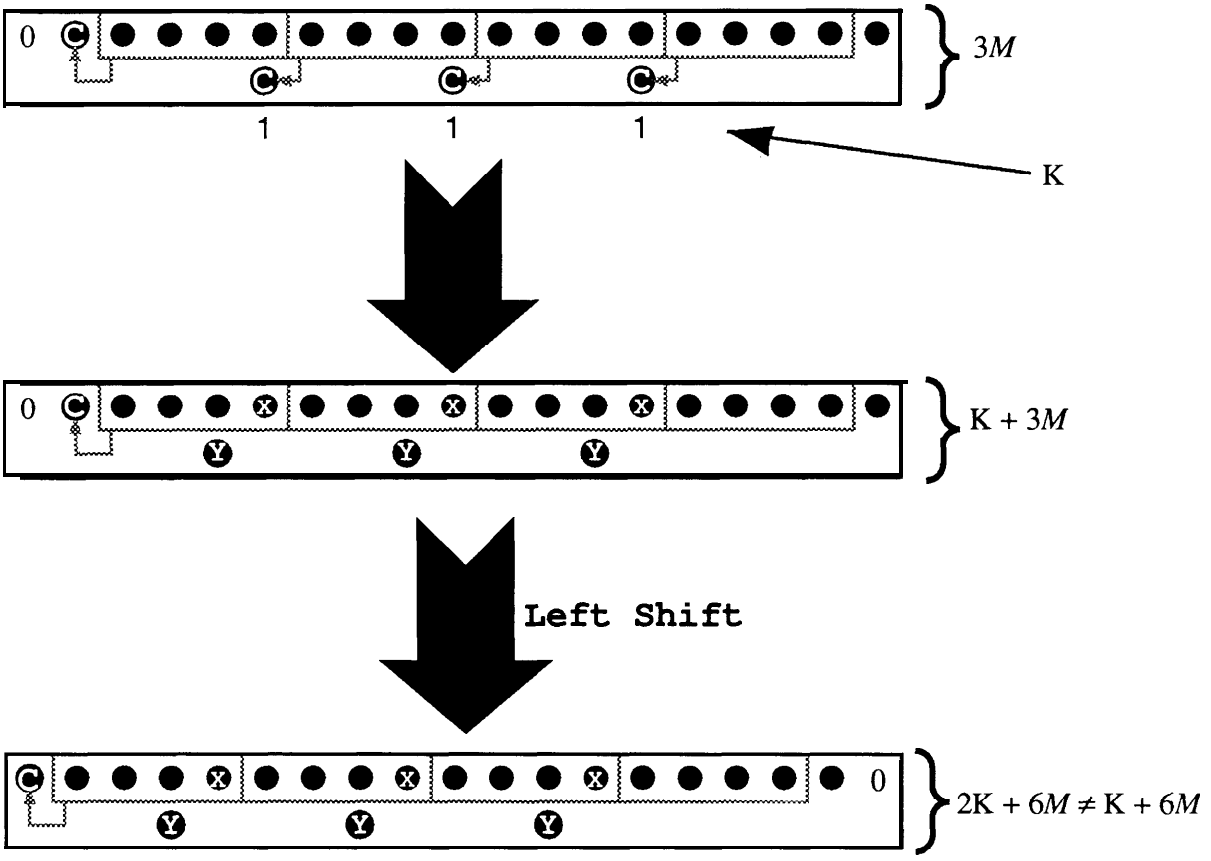


Figure 23: Producing K + 6M from K + 3M ?

shows the format of K + 3M. The problem arises when the biased multiple K + 6M is required. The normal (unbiased) Booth algorithms obtain 6M by a single left shift of 3M. If this is tried using the partially redundant biased representation, then the result is not K + 6M, but 2K + 6M. This violates one of the original premises, that the bias constant for each partial product is independent of the multiple being selected. In addition to this problem, the actual positions of the Y bits has shifted.

These problems can be overcome by choosing a different bias constant, as illustrated in Figure 24. The bias constant is selected to be non-zero only in bit positions corresponding to carries ***after*** shifting to create the 6M multiple. The three bits in the area of the non-zero part of K (circled in the figure) can be summed, but the summation is not the same for 3M (left side of the figure) as for 6M (right side of the figure). Extra signals must be routed to the Booth multiplexers, to simplify them as much as possible (there may be many of them if the multiply is fairly large). For example, to fully form the 3 dots labeled "X", "Y", and "Z" requires the routing of 5 signal wires. Creative use of hardware dependent circuit design (for example creating OR gates at the inputs of

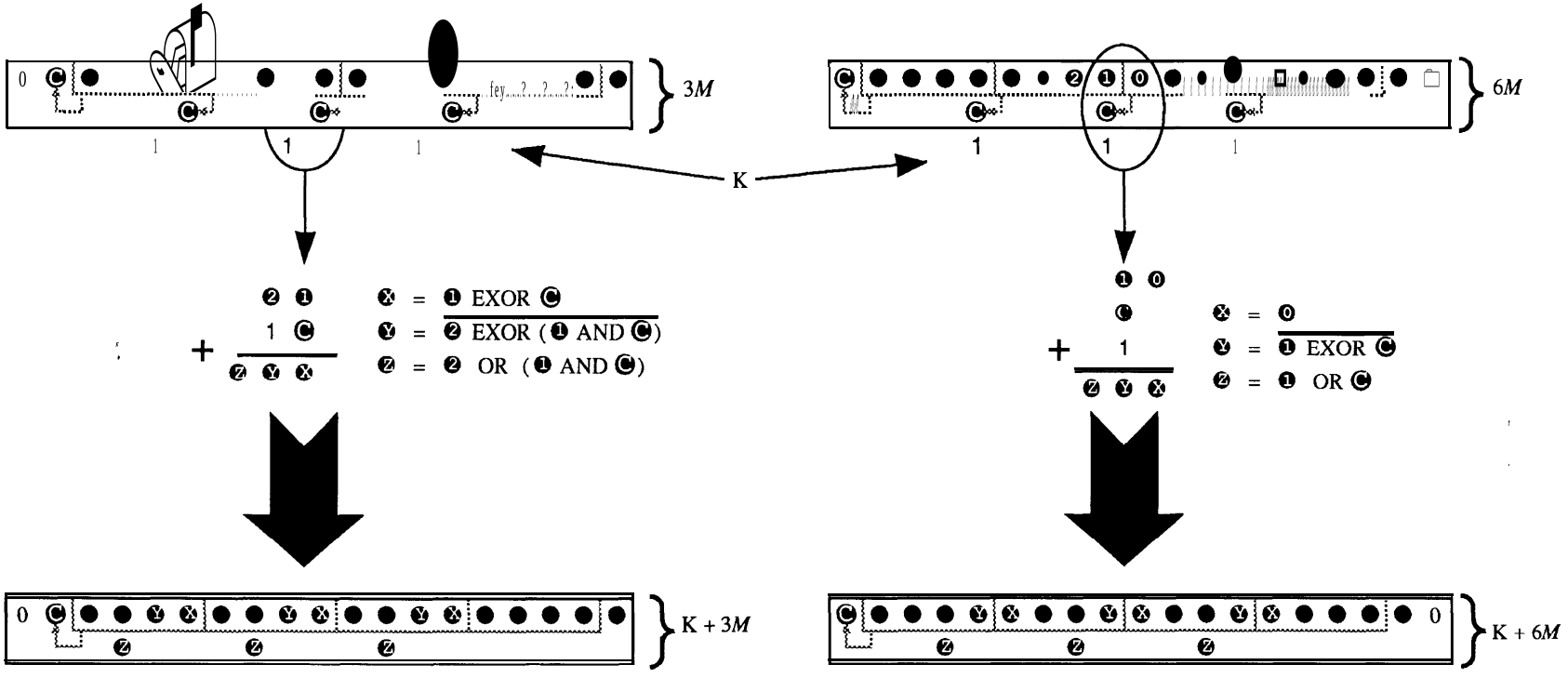


Figure 24: A different bias constant for 6M and 3M

the multiplexers) can reduce this to 4, but this still means that there are more routing wires for a multiple than there are dots in the multiple. Of course since there are now 3 multiples that must be routed (3M, 5M, and 7M), these few extra wires may not be significant.

There are many other problems, which are inherited from the non-redundant Booth 4 algorithm. Larger multiplexers – each multiplexer must choose from 8 possibilities, twice as many as for the Booth 3 algorithm – are required. There is also a smaller hardware reduction in going from Booth 3 to Booth 4 then there was in going from Booth 2 to Booth 3. Optimizations are also possible for generation of the 3M multiple. These optimizations are not possible for the 5M and 7M multiples, so the small adders that generate these multiples must be of a smaller length (for a given delay). This means more dots in the partial product section to be summed.

Thus a redundant Booth 4 algorithm is possible to construct, but probably has little speed or implementation advantage over the redundant Booth 3 algorithm. The hardware savings due to the reduced number of partial products is exceeded by the cost of the adders needed to produce the three hard multiples, and the increased complexity of the multiplexers required to select the partial products.

3.6 Choosing the adder length

By and large, the rule for choosing the length of the small adders necessary for is straightforward - The largest possible adder should be chosen. This will minimize the amount of hardware needed for summing the partial products. Since the multiple generation occurs in parallel with the Booth decoding, there is little point in reducing the adder lengths to the point where they are faster than the Booth decoder. The exact length is dependent on the actual technology used in the implementation, and must be determined empirically.

Certain lengths should be avoided, as illustrated in Figure 25. This figure assumes a redundant

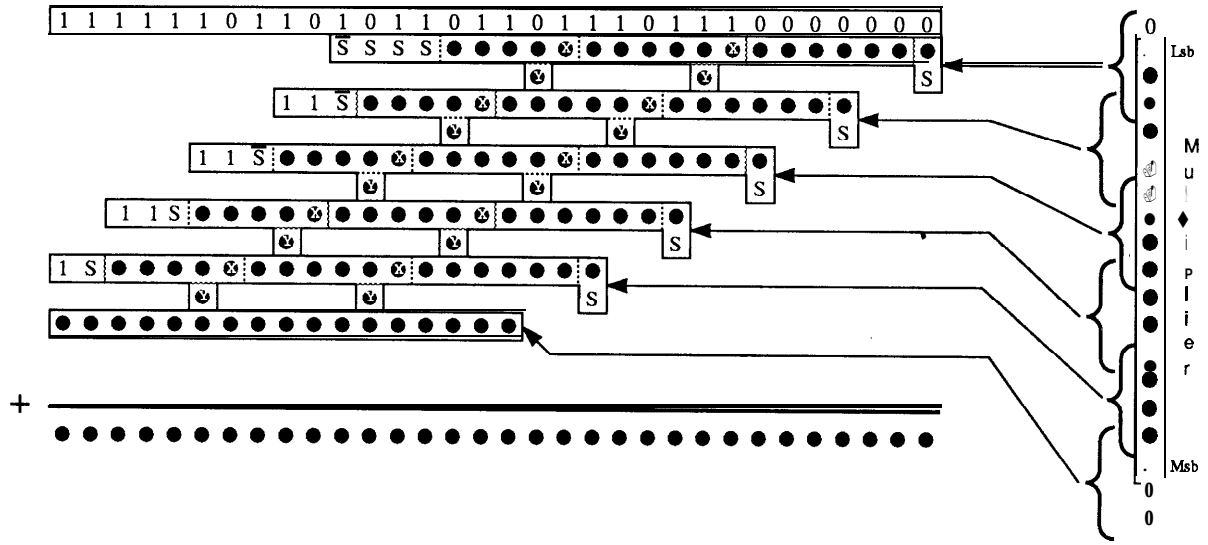


Figure 25: Redundant Booth 3 with 6 bit adders

Booth 3 algorithm, with a carry interval of 6 bits. Note the accumulation of dots at certain positions in the dot diagram. In particular, the column forming bit 15 of the product is now 8 high (vs 7

for a 4 bit carry interval). This accumulation can be avoided by choosing adder lengths which are relatively prime to the shift amount between neighboring partial products (in this case, 3). This spreads the Y bits out so that accumulation won't occur in any particular column.

4 Summary

This report has described a new variation on conventional Booth multiplication algorithms. By representing partial products in a partially redundant form, hard multiples can be computed without a slow, full length carry propagate addition. With such hard multiples available, a reduction in the amount of hardware needed for summing partial products is then possible using the Booth 3 multiplication method. A detailed evaluation of implementations using this algorithm is currently in progress, which will give precise answers to performance, area, and power improvements.

References

[1] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics,* 4(2):236–240, June 1951.

[2] 0. L. MacSorley. High-speed arithmetic in binary computers. *Proceedings of the IRE,* 49(1):67-91, Jan 1961.

[3] Mark Santoro. *Design and Clocking of VLSI Multipliers.* PhD thesis, Stanford University, Oct 1989.

[4] Mark Santoro and Mark Horowitz. Spim: A pipelined 64x64b iterative array multiplier. *IEEE International Solid State Circuits Conference,* pages 35-36, February 1988.

[5] Naofumi Takagi, Hiroto Yasuura, and Shuzo Yajima. High-speed VLSI multiplication algorithm with a redundant binary addition tree. *IEEE Transactions on Computers, C-34(9),* Sept 1985.

[6] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital Systems Designers.* Holt, Rinehart and Winston, 1982.