FAST MULTIPLICATION:
ALGORITHMS AND IMPLEMENTATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Gary W. Bewick
February 1994

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Michael J. Flynn
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark A. Horowitz

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Constance J. Chang-Hasnain

Approved for the University Committee on Graduate Studies:

Abstract

This thesis investigates methods of implementing binary multiplication with the smallest possible latency. The principle area of concentration is on multipliers with lengths of 53 bits, which makes the results suitable for IEEE-754 double precision multiplication.

Low latency demands high performance circuitry, and small physical size to limit propagation delays. VLSI implementations are the only available means for meeting these two requirements, but efficient algorithms are also crucial. An extension to Booth's algorithm for multiplication (redundant Booth) has been developed, which represents partial products in a partially redundant form. This redundant representation can reduce or eliminate the time required to produce "hard" multiples (multiples that require a carry propagate addition) required by the traditional higher order Booth algorithms. This extension reduces the area and power requirements of fully parallel implementations, but is also as fast as any multiplication method yet reported.

In order to evaluate various multiplication algorithms, a software tool has been developed which automates the layout and optimization of parallel multiplier trees. The tool takes into consideration wire and asymmetric input delays, as well as gate delays, as the tree is built. The tool is used to design multipliers based upon various algorithms, using both Booth encoded, non-Booth encoded and the new extended Booth algorithms. The designs are then compared on the basis of delay, power, and area.

For maximum speed, the designs are based upon a 0.6μ BiCMOS process using emitter coupled logic (ECL). The algorithms developed in this thesis make possible 53x53 multipliers with a latency of less than 2.6 nanoseconds @ 10.5 Watts and a layout area of $13mm^2$. Smaller and lower power designs are also possible, as illustrated by an example with a latency of 3.6 nanoseconds @ 5.8 W, and an area of $8.9mm^2$. The conclusions based

upon ECL designs are extended where possible to other technologies (CMOS).

Crucial to the performance of multipliers are high speed carry propagate adders. A number of high speed adder designs have been developed, and the algorithms and design of these adders are discussed.

The implementations developed for this study indicate that traditional Booth encoded multipliers are superior in layout area, power, and delay to non-Booth encoded multipliers. Redundant Booth encoding further reduces the area and power requirements. Finally, only half of the total multiplier delay was found to be due to the summation of the partial products. The remaining delay was due to wires and carry propagate adder delays.

Acknowledgements

The work presented in this thesis would not have been possible without the assistance and cooperation of many people and organizations. I would like to thank the people at Philips Research Laboratories - Sunnyvale, especially Peter Baltus and Uzi Bar-Gadda for their assistance and support during my early years here at Stanford. I am also grateful to the people at Sun Microsystems Inc., specifically George Taylor, Mark Santoro and the entire P200 gang. I would like to extend thanks to the members of my committee, Constance Chang-Hasnain, Giovanni De Micheli and Mark Horowitz for their time and patience. Mark, in particular, provided many helpful suggestions for this thesis.

Finally I would like to thank my advisor, colleague, and friend Michael Flynn for providing guidance and keeping me on track, but also allowing me the freedom to pursue areas in my own way and at my own pace. Mike was always there when I needed someone to bounce ideas off of, or needed support, or requested guidance. My years at Stanford were hard work, sometimes frustrating, but I always had fun.

The work presented in this thesis was supported by NSF under contract MIP88-22961.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the performance of processors has increased, the demand for high speed arithmetic blocks has also increased. With clock frequencies approaching 1 GHz, arithmetic blocks must keep pace with the continued demand for more computational power. The purpose of this thesis is to present methods of implementing high speed binary multiplication. In general, both the algorithms used to perform multiplication, and the actual implementation procedures are addressed. The emphasis of this thesis is on minimizing the latency, with the goal being the implementation of the fastest multiplication blocks possible.

1.1 Technology Options

Fast arithmetic requires fast circuits. Fast circuits require small size, to minimize the delay effects of wires. Small size implies a single chip implementation, to minimize wire delays, and to make it possible to implement these fast circuits as part of a larger single chip system to minimize input/output delays. Even for single chip implementations, a number of choices exist as to the implementation technology and architecture. A brief review of some of the options is presented in order to provide some motivation as to the choices that were made for this thesis.

1.1.1 CMOS

CMOS (Complementary Metal Oxide Semiconductor) is the primary technology in the semiconductor industry at the present time. Most high speed microprocessors are implemented using CMOS. Contemporary CMOS technology is characterized by :

- Small minimum sized transistors, allowing for dense layouts, although the interconnect limits the density.
- Low Quiescent Power - The power consumption of conventional CMOS circuits is largely determined by the AC power caused by the charge and discharge of capacitances :

  $$\mathrm{Power} \propto \mathrm{CV^2f} \tag{1.1}$$

  where f is the frequency at which a capacitance is charged and discharged. As the circuits get faster, the frequency goes up as does the power consumption.
- Relatively simple fabrication process.
- Large required transistors - In order to drive wires quickly, large width transistors are needed, since the time to drive a load is given by :

  $$\Delta\mathrm{t} = \mathrm{C}\frac{\Delta\mathrm{V}}{\mathrm{i}} \tag{1.2}$$

  where :

  Δt is the time to charge or discharge the load
  C is the capacitance associated with the load
  ΔV is the load voltage swing
  i is the average current provided by the load driver
- Large voltage swings - Typical voltage swings for contemporary CMOS are from 3.3 to 5 volts (with even smaller swings on the way). All other things being equal, equation 1.2 says that a smaller voltage swing will be proportionally faster.
- Good noise margins.

BiCMOS

BiCMOS generally refers to CMOS-BiCMOS where bipolar transistors are used to improve the driving capability of CMOS logic elements (Figure 1.1). In general this will improve

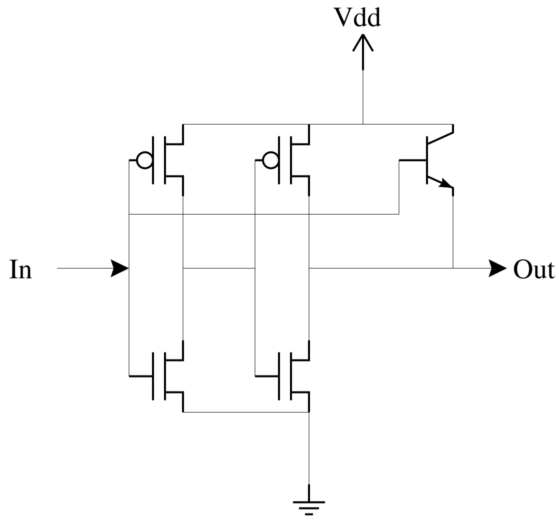


Figure 1.1: BiCMOS (BiNMOS) buffer.

the driving capability of relatively long wires by about a factor of two [2] [22]. A parallel multiplier does indeed have some long wires, and the long wires contribute significantly to the total delay, but the delay is not dominated by the long wires. A large number of short wires also contribute significantly to delay. The net effect is perhaps a 20 to 30% improvement in performance. The addition of the bipolar transistors increases the process complexity significantly and it is not clear that the additional complexity is worth this level of improvement.

1.1.2 ECL

ECL (emitter coupled logic) [20] uses bipolar transistors exclusively to produce various logic elements (Figure 1.2). The primary advantage of bipolar transistors is that they have an exponential turn-on characteristic, that is the current through the device is exponentially related to the base-emitter voltage. This allows extremely small voltage swings (0.5V) in logic elements. Referring back to Equation 1.2, this results in a proportional speed up

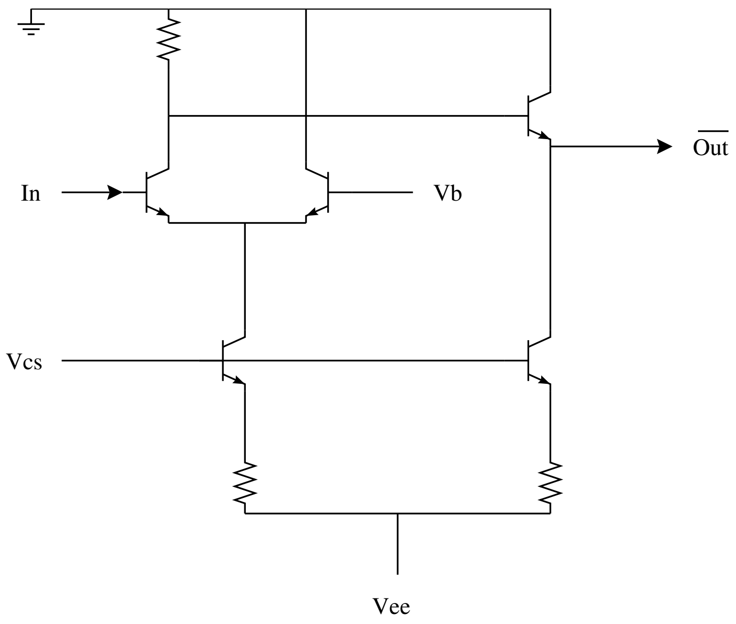


Figure 1.2: ECL inverter.

in the basic logic element. For highest speed the bipolar transistors must be kept from saturating, which means that they must be used in a current switching mode. Unlike CMOS or BiCMOS, logic elements dissipate power even if the element is not switching, resulting in a very high DC power consumption. The total power consumption is relatively independent of frequency, so even at extremely high frequencies the power consumption will be about the same as the DC power consumption. In contrast, CMOS or BiCMOS power increases with frequency. Even at high frequencies, CMOS probably has a better speed-power product than ECL, but this depends on the exact nature of the circuitry. A partial solution to the high power consumption problem of ECL is to build relatively complex gates, for example building a full adder directly rather than building it from NOR gates. Other methods of reducing power are described in Chapter 4.

Differential ECL

Differential ECL is a simple variation on regular ECL which uses two wires to represent a single logic signal, with each wire having 1/2 the voltage swing of normal. To first order, this means that differential ECL is approximately twice as fast as ECL (Equation 1.2), but more wires are needed and more power may be required.

1.2 Technology Choice

Historically, ECL has been the choice when the highest speed was desired, it's main drawback being high power consumption. Although CMOS has been closing the speed gap, at high speeds it too is a high power technology. At the present time ECL, as measured by loaded gate delays, is somewhere between $\frac{1}{2}$ and $\frac{1}{4}$ the delay of similar CMOS gates. Comparable designs in ECL also take about the same layout area as a CMOS design, primarily because the metal interconnect limits the circuit densities. Because ECL seems to still maintain a speed advantage, the technology used as a basis for this thesis will be ECL, supplemented with differential ECL where possible. Most conclusions will apply primarily to implementations using ECL, but wherever possible, the results will be generalized to other implementation technologies, principally CMOS.

1.3 Multiplication Architectures

Chapter 2 presents partial product generation in detail, but all multiplication methods share the same basic procedure - addition of a number of partial products. A number of different methods can be used to add the partial products. The simple methods are easy to implement, but the more complex methods are needed to obtain the fastest possible speed.

1.3.1 Iterative

The simplest method of adding a series of partial products is shown in Figure 1.3. It is based upon an adder-accumulator, along with a partial product generator and a hard wired shifter. This is relatively slow, because adding N partial products requires N clock cycles. The easiest clocking scheme is to make use of the system clock, if the multiplier is embedded in a larger system. The system clock is normally much slower than the maximum speed at which the simple iterative multiplier can be clocked, so if the delay is to be minimized an expensive and tricky clock multiplier is needed, or the hardware must be self-clocking.

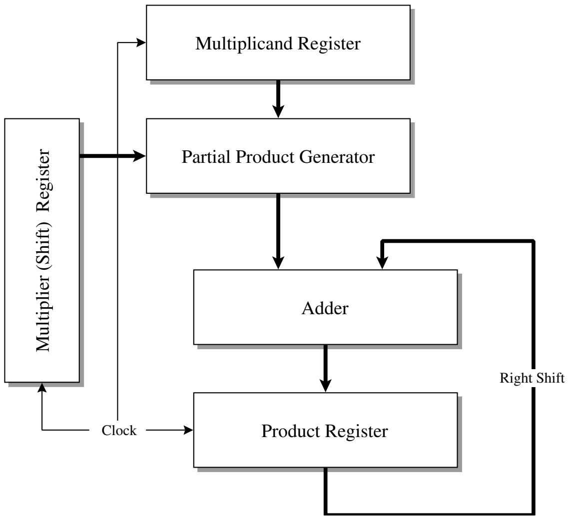


Figure 1.3: Simple iterative multiplier.

1.3.2 Linear Arrays

A faster version of the basic iterative multiplier adds more than one operand per clock cycle by having multiple adders and partial product generators connected in series (Figure 1.4). This is the equivalent of "unrolling" the simple iterative method. The degree to which the loop is unrolled determines the number of partial products that can be reduced in each clock cycle, but also increases the hardware requirements. Typically, the loop is unrolled only to the point where the system clock matches the clocking rate of this multiplier. Alternately, the loop can be unrolled completely, producing a completely combinatorial multiplier (a full linear array). When contrasted with the simple iterative scheme, it will match the system clock speed better, making the clocking much simpler. There is also less overhead associated with clock skew and register delay per partial product reduced.

1.3.3 Parallel Addition (Trees)

When a number of partial products are to be added, the adders need not be connected in series, but instead can be connected to maximize parallelism, as shown in Figure 1.5. This requires no more hardware than a linear array, but does have more complex interconnections. The time required to add N partial products is now proportional to log N, so this can be much

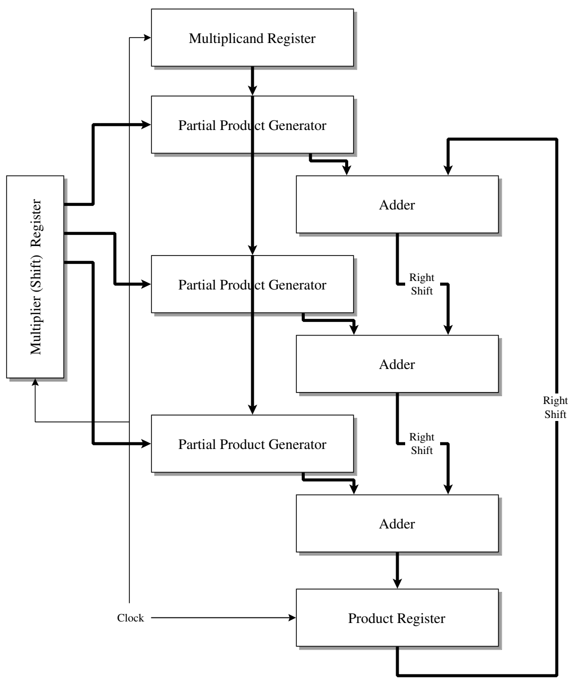


Figure 1.4: Linear array multiplier. Reduces 3 partial products per clock.

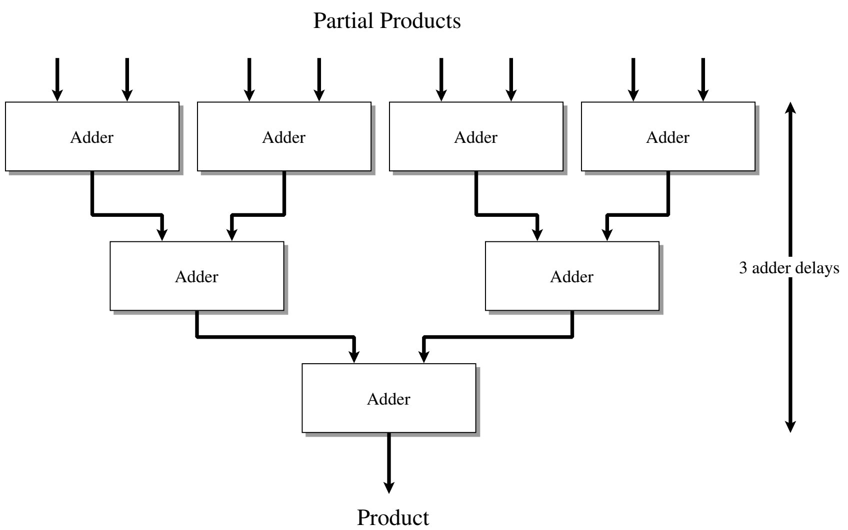


Figure 1.5: Adding 8 partial products in parallel.

faster for larger values of N. On the down side, the extra complexity in the interconnection of the adders may contribute to additional size and delay.

1.3.4 Wallace Trees

The performance of the above schemes are limited by the time to do a carry propagate addition. Carry propagate adds are relatively slow, because of the long wires needed to propagate carries from low order bits to high order bits. Probably the single most important advance in improving the speed of multipliers, pioneered by Wallace [35], is the use of carry save adders (CSAs also known as full adders or 3-2 counters [7]), to add three or more numbers in a redundant and carry propagate free manner. The method is illustrated in Figure 1.6. By applying the basic three input adder in a recursive manner, any number of

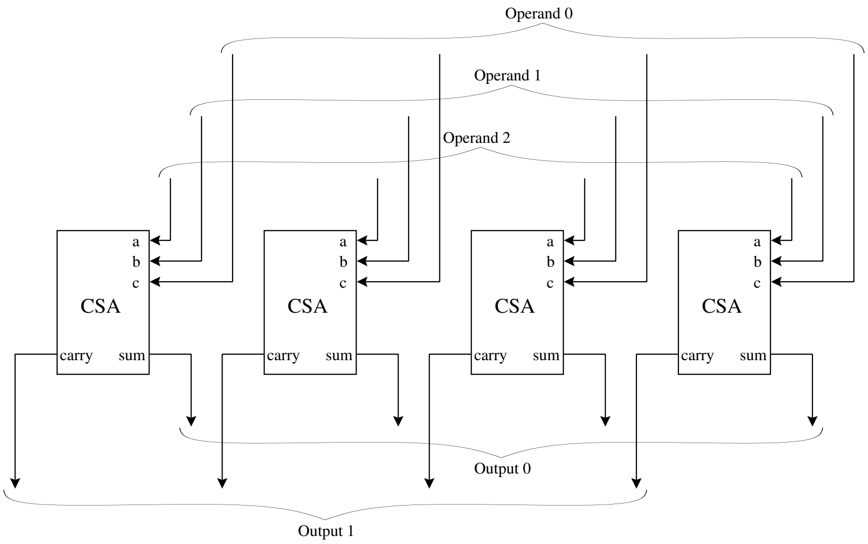


Figure 1.6: Reducing 3 operands to 2 using CSAs.

partial products can be added and reduced to 2 numbers without a carry propagate adder. A single carry propagate addition is only needed in the final step to reduce the 2 numbers to a single, final product. The general method can be applied to trees and linear arrays alike to improve the performance.

Binary Trees

The tree structure described by Wallace suffers from irregular interconnections and is difficult to layout. A more regular tree structure is described by [24], [37], and [30], all of which are based upon binary trees. A binary tree can be constructed by using a row of 4-2 counters [1], which accepts 4 numbers and sums them to produce 2 numbers. Although this improves the layout problem, there are still irregularities, an example of which is shown in Figure 1.7. This figure shows the reduction of 8 partial products in two levels of 4-2 counters to two numbers, which would then be reduced to a final product by a carry propagate adder. The shifting of the partial products introduce zeros at various places in the reduction. These zeros represent either hardware inefficiency, if the zeros are actually added, or irregularities in the tree if special counters are built to explicitly exclude the zeros from the summation. The figure shows bits that jump levels (gray dots), and more counters in the row making up the second level of counters (12), than there are in the rows making up the first level of counters (9). All of these effects contribute to irregularities in the layout, although it is still more regular than a Wallace tree.

1.4 Architectural Choices

With the choice of ECL as an implementation technology, many of the architectural choices are determined. Registers are extremely expensive, both in layout area and in power requirements. Because of the high potential speed and minimum amount of overhead circuitry (such as registers, clock distribution and skew), a fully parallel, tree implementation seems to promise the highest possible speed. Implementations and comparisons will be based upon this assumption, although smaller tree or array structures will be noted when appropriate.

ECL allows the efficient implementation of CSAs. Two tail (gate) currents are necessary per CSA. The most efficient implementations of 4-2 counters, or higher order blocks (such as 5-5-4 or 7-3 counters) appear to offer no advantage in area or power consumption. For

[1] 4-2 adders, as used by Santoro[24] and Weinberger[37], are easily constructed from two CSAs, however in some technologies a more direct method may be faster.

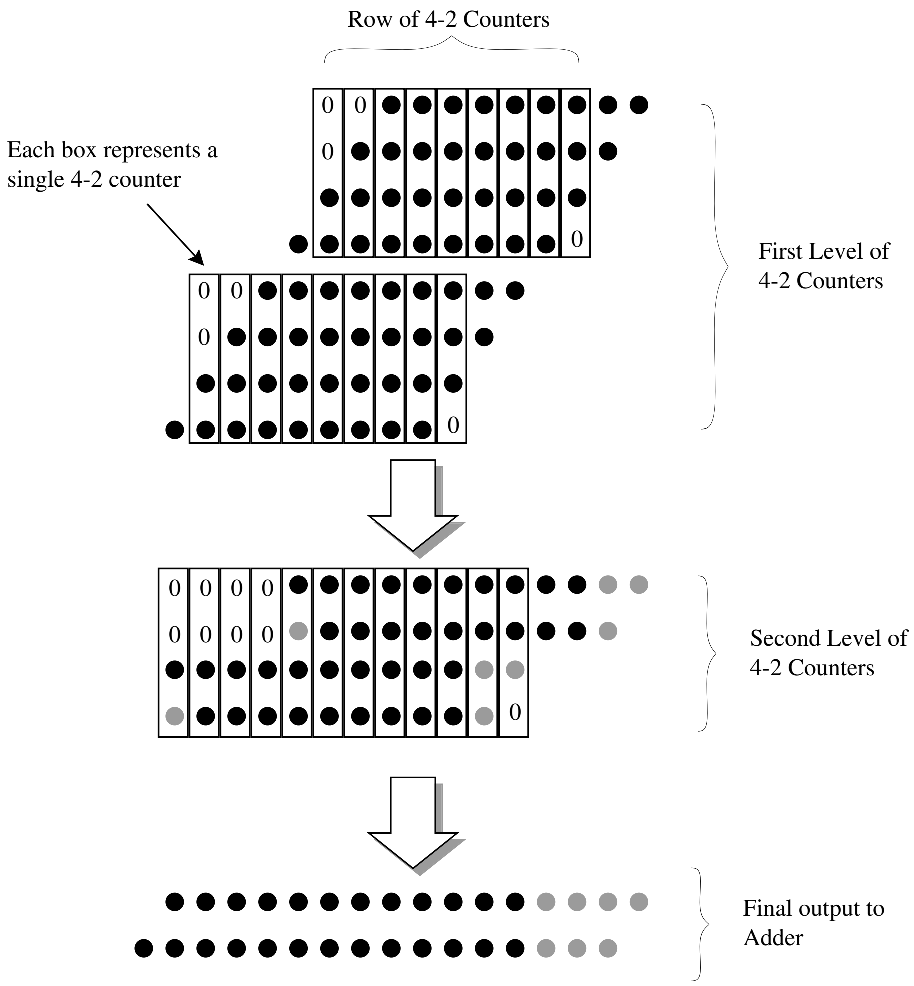


Figure 1.7: Reduction of 8 partial products with 4-2 counters.

this reason architectures based upon CSAs will be considered exclusively. To overcome the wiring complexity of the direct usage of CSAs, an automated tool will be used to implement multiplier trees. This tool is described in detail in later chapters, and is responsible for placement, wiring, and optimization of multiplier tree structures.

1.5 Thesis Structure

The remaining portion of this thesis is structured as follows :

- Chapter 2 - Begins the main contribution of this thesis, by reviewing existing partial product generation algorithms. A new class of algorithms, (Redundant Booth) which is a variation on more conventional algorithms, is described.

- Chapter 3 - Presents the design of various carry propagate adders and multiple generators. Carry propagate adders play a crucial role in the design of high speed multipliers. After the partial products are reduced as far as possible in a redundant form, a carry propagate addition is needed to produce the final product. This addition consumes a significant fraction of the total multiply time.

- Chapter 4 - Describes a software tool that has been developed for this thesis, which automatically produces the layout and wiring of multiplier trees of various sizes and algorithms. The tool also performs a number of optimizations to reduce the layout area and increase the speed.

- Chapter 5 - Combines the results of Chapters 2, 3 and 4 to compare implementations using various partial product generation algorithms on the basis of speed, power, and layout area. All of the multipliers perform a 53 by 53 bit unsigned multiply, which is suitable for IEEE-754 [12] double precision multiplication. Some interesting and unique variations on conventional algorithms are also presented. Implementations based upon the redundant Booth algorithm are also included in the analysis. The designs are also compared to other designs described in the literature.

- Chapter 6 - Closes the main body of this thesis by noting that the delay of all pieces of a multiplier are important. In particular long control wire delays, multiple distribution, and carry propagate adder delays are at least as important in determining the overall performance as the partial product summing delay.

Chapter 2

Generating Partial Products

Chapter 1 briefly described a number of different methods of implementing integer multipliers. The methods all reduce to two basic steps – create a group of partial products, then add them up to produce the final product. Different ways of adding the partial products were mentioned, but little was said about how to generate the partial products to be summed. This chapter presents a number of different methods for producing partial products. The simplest partial product generator produces N partial products, where N is the length of the input operands. A recoding scheme introduced by Booth [5] reduces the number of partial products by about a factor of two. Since the amount of hardware and the delay depends on the number of partial products to be added, this may reduce the hardware cost and improve performance. Straightforward extensions of the Booth recoding scheme can further reduce the number of partial products, but require a time consuming N bit carry propagate addition before any partial product generation can take place. The final sections of this chapter will present a new variation on Booth's algorithm which reduces the number of partial products by nearly a factor of three, but does not require an N bit carry propagate add for partial product generation.

This chapter attempts to stay away from implementation details, but concentrates on the partial product generation in a hardware independent manner. Unsigned multiplication only will be considered here, in order that that the basic methods are not obscured with small details. Multiplication of unsigned numbers is also important because most floating point formats represent numbers in a sign magnitude form, completely separating the mantissa multiplication from the sign handling. The methods are all easily extended to deal with signed numbers, an example of which is presented in Appendix A.

2.1 Background

2.1.1 Dot Diagrams

The partial product generation process is illustrated by the use of a *dot diagram*. Figure 2.1 shows the dot diagram for the partial products of a 16x16 bit *Simple Multiplication*. Each

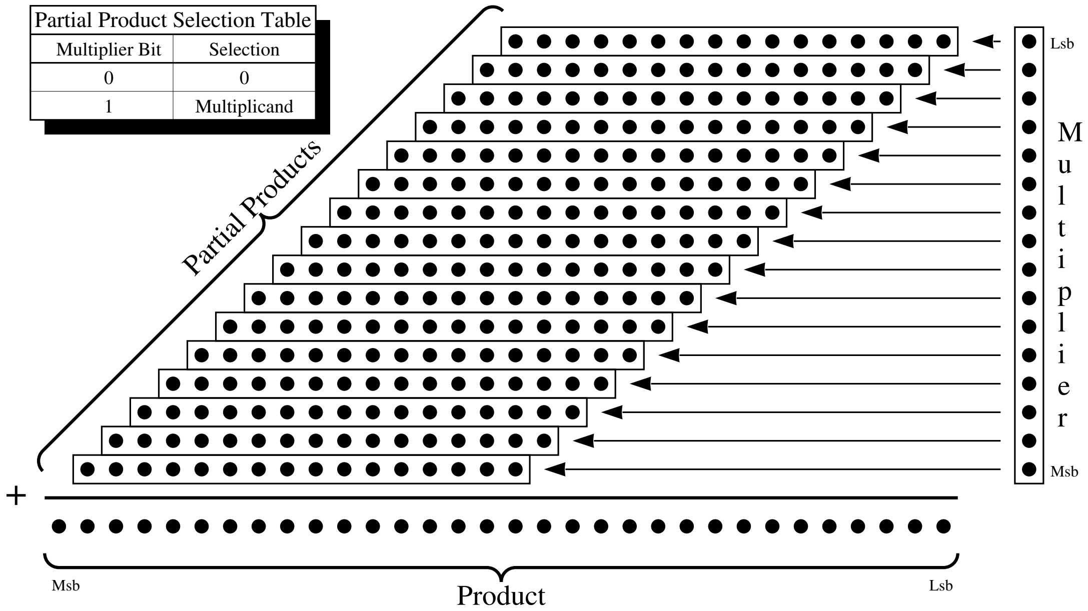


Figure 2.1: 16 bit simple multiplication.

dot in the diagram is a place holder for a single bit which can be a zero or one. The partial products are represented by a horizontal row of dots, and the selection method used in producing each partial product is shown by the table in the upper left corner. The partial products are shifted to account for the differing arithmetic weight of the bits in the multiplier, aligning dots of the same arithmetic weight vertically. The final product is represented by the double length row of dots at the bottom. To further illustrate simple multiplication, an example using real numbers is shown in Figure 2.2.

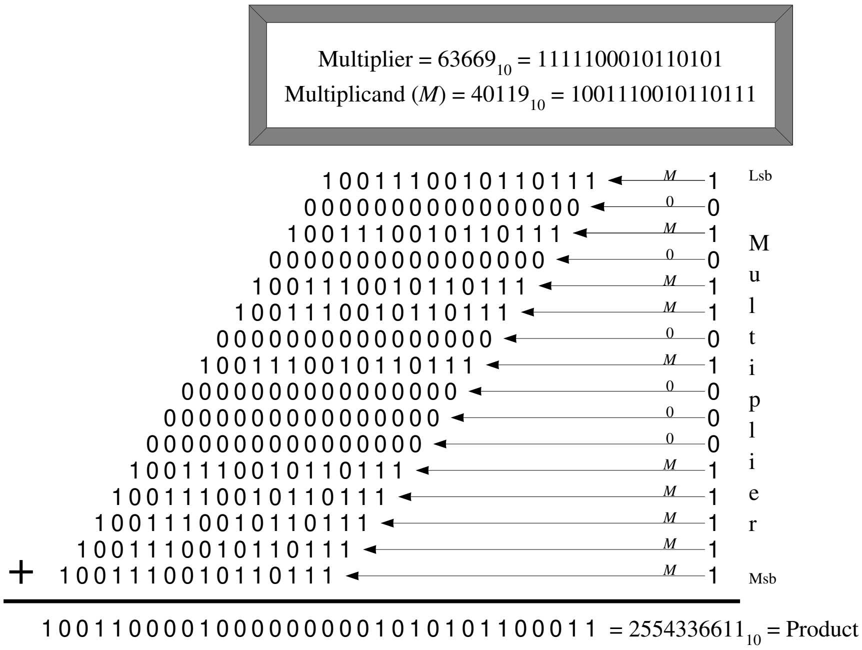


Figure 2.2: 16 bit simple multiplication example.

Roughly speaking, the number of dots (256 for Figure 2.1) 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 [25]) 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). But independent of the implementation, adding fewer partial products is always better.

Finally, the logic which selects the partial products can be deduced from the partial product selection table. For the simple multiplication algorithm, the logic consists of a single AND gate per bit as shown in Figure 2.3. This figure shows the selection logic for a single partial product (a single row of dots). Frequently this logic can be merged directly into whatever hardware is being used to sum the partial products. This merging can reduce the delay of the logic elements to the point where the extra time due to the selection elements

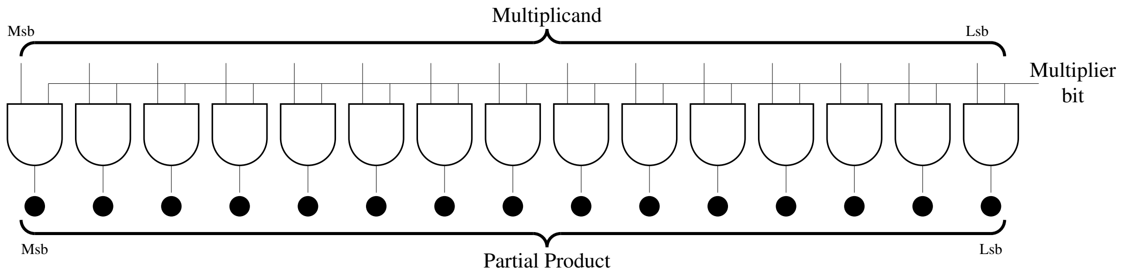
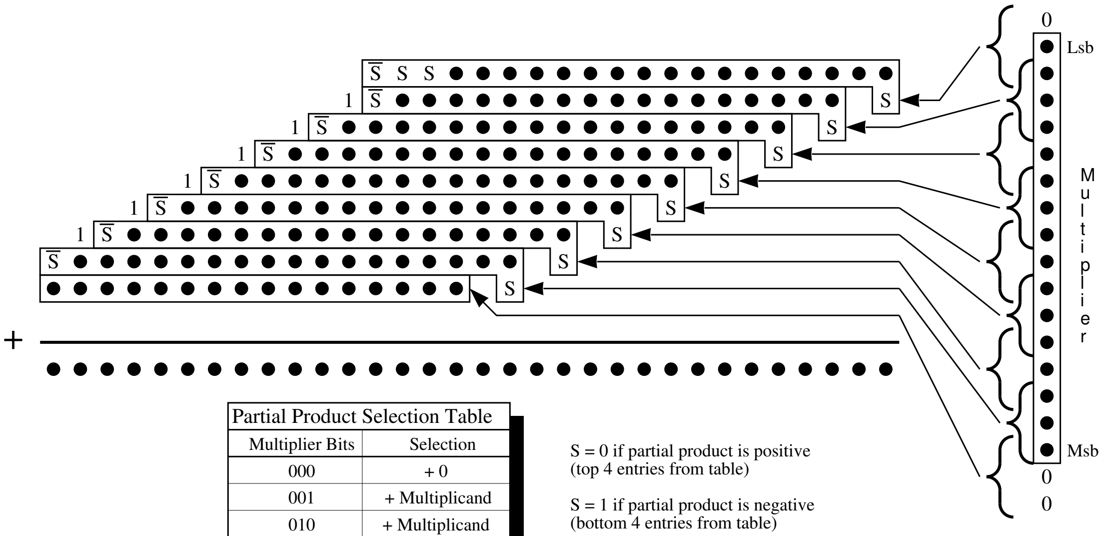


Figure 2.3: Partial product selection logic for simple multiplication.

can be ignored. However, in a real implementation there will still be interconnect delay due to the physical separation of the common inputs of each AND gate, and distribution of the multiplicand to the selection elements.

2.1.2 Booth's Algorithm

A generator that creates a smaller number of partial products will allow the partial product summation to be faster and use less hardware. The simple multiplication generator can be extended to reduce the number of partial products by grouping the bits of the multiplier into pairs, and selecting the partial products from the set {0,M,2M,3M}, where M is the multiplicand. This reduces the number of partial products by half, but requires a carry propagate add to produce the 3M multiple, before any partial products can be generated. Instead, a method known as *Modified Booth's Algorithm* [5] [17] reduces the number of partial products by about a factor of two, without requiring a preadd to produce the partial products. The general idea is to do a little more work when decoding the multiplier, such that the multiples required come from the set {0,M,2M,4M + -M}. All of the multiples from this set can be produced using simple shifting and complementing. The scheme works by changing any use of the 3M multiple into 4M - M. Depending on the adjacent multiplier groups, either 4M is pushed into the next most significant group (becoming M because of the different arithmetic weight of the group), or -M is pushed into the next least significant group (becoming -4M). Figure 2.4 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 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 2.4: 16 bit Booth 2 multiply.

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 referred to as *easy* multiples). Negative multiples, in two'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 2.5. In this case Booth's algorithm has reduced the total number of dots from 256 to 177 (this includes sign extension and constants – see Appendix A for a discussion of sign extension). This reduction in dot count is not a complete saving – the partial product selection logic is more complex (Figure 2.6). 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 [24] (more on this in Chapter 5).

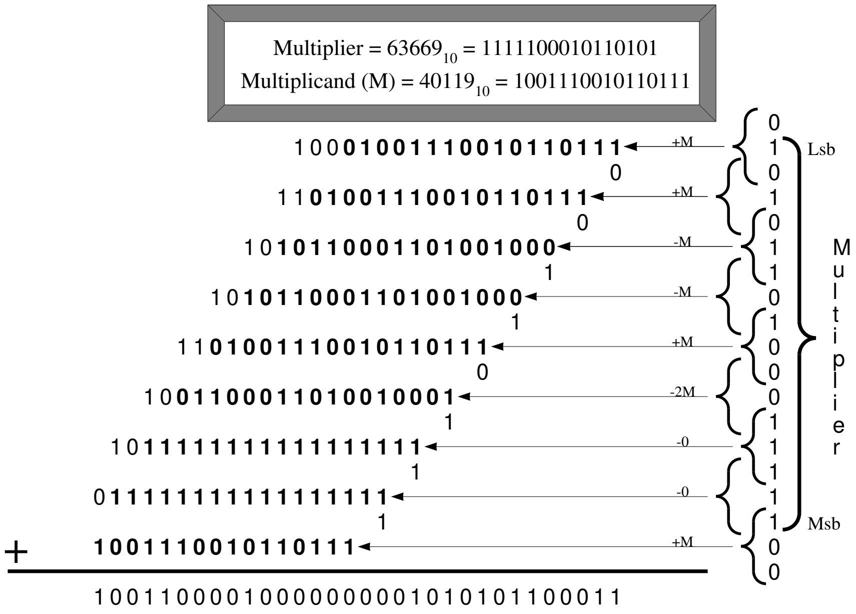


Figure 2.5: 16 bit Booth 2 example.

2.1.3 Booth 3

Actually, Booth's algorithm can produce shift amounts between adjacent partial products of greater than 2 [17], with a corresponding reduction in the height and number of dots in the dot diagram. A 3 bit Booth (Booth 3) dot diagram is shown in Figure 2.7, and an example is shown in Figure 2.8. Each partial product could be from the set $\{\pm 0, \pm\mathrm{M}, \pm 2\mathrm{M}, \pm 3\mathrm{M}, \pm 4\mathrm{M}\ \}$. All multiples with the exception of 3M are 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

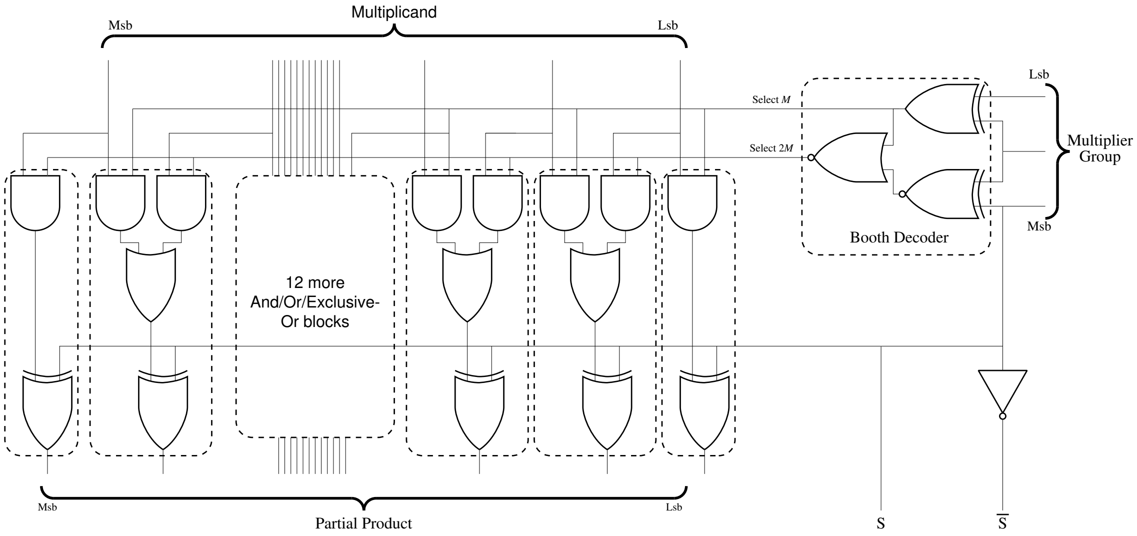
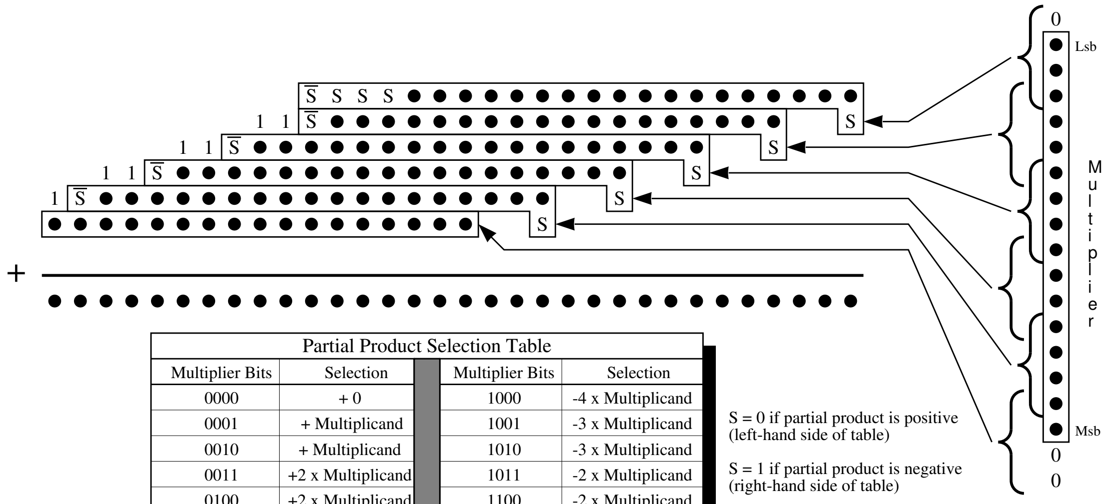


Figure 2.6: 16 bit Booth 2 partial product selector logic.



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

S = 0 if partial product is positive (left-hand side of table)

S = 1 if partial product is negative (right-hand side of table)

Figure 2.7: 16 bit Booth 3 multiply.

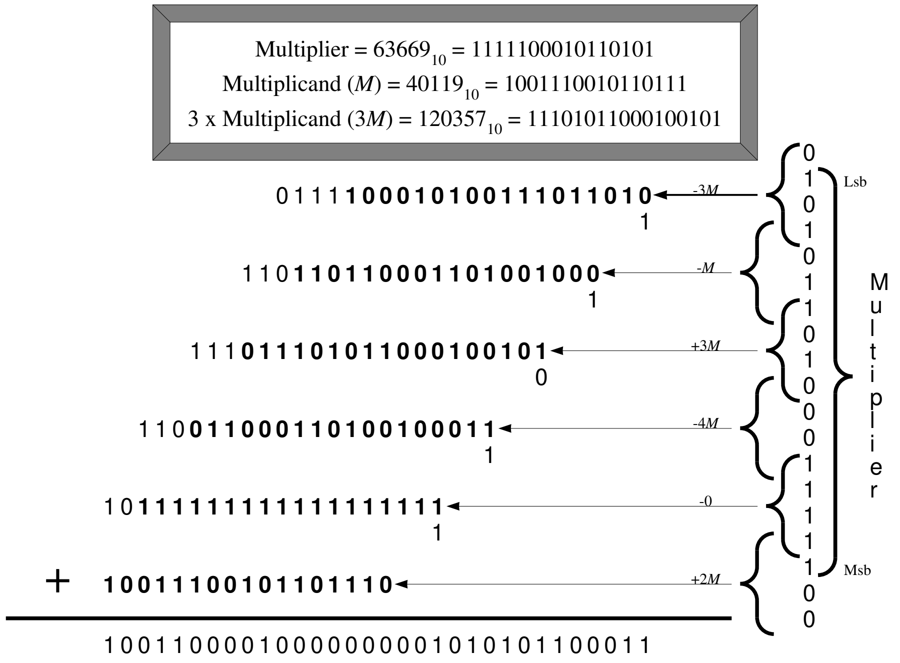


Figure 2.8: 16 bit Booth 3 example.

logic, an example of which is shown in Figure 2.9, along with the extra wiring needed for routing the 3M multiple.

2.1.4 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 2.10) requires the generation of the multiples {±0, ±M, ±2M, ±3M, ±4M,±5M,±6M,±7M,±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, larger partial product selection multiplexers, and the additional wires that are needed to route the various multiples around.

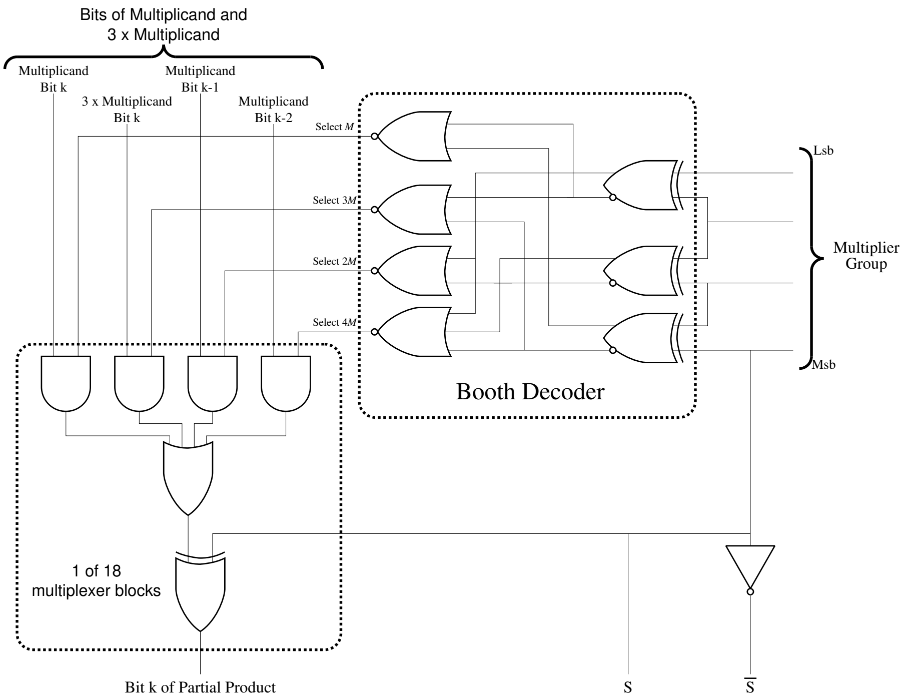


Figure 2.9: 16 bit Booth 3 partial product selector logic.

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 2.10: Booth 4 partial product selection table.

2.2 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 hard multiple generation, yet produces a dot diagram which can be made to approach that of the conventional Booth 3 algorithm. To motivate this variation a similar, but slightly simpler is explained. Improving the hardware efficiency of this method produces the new variation. Methods of further generalizing to a Booth 4 algorithm are then discussed.

2.2.1 Booth 3 with Fully Redundant Partial Products

The time consuming carry propagate addition that is required to generate the "hard multiples" 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 [30]). 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 2.11 (an example appears in Figure 2.12). 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 (in 2's complement form) 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, two 1s are added at the lsb to complete the 2's complement negation. These two 1s can be preadded 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 suggests that a partially redundant representation of the partial products might lead to a more efficient variant.

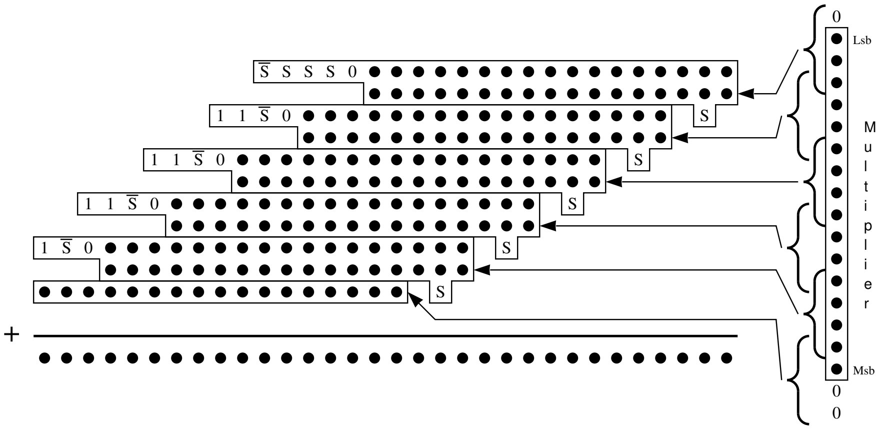


Figure 2.11: 16 x 16 Booth 3 multiply with fully redundant partial products.

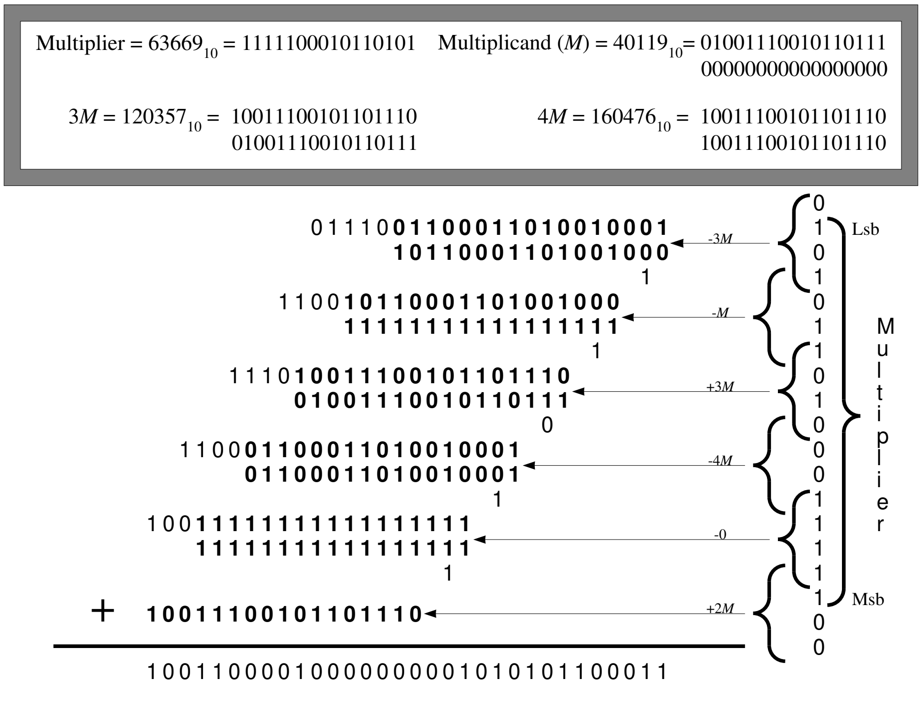


Figure 2.12: 16 bit fully redundant Booth 3 example.

2.2.2 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 2.13). 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, carries will not be propagated across large distances, and the small adders will be 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.

A difficulty with the partially redundant representation shown in Figure 2.13 is that negative partial products do not preserve the proper redundant form. To illustrate the problem, the top of Figure 2.14 shows a number in the proposed redundant form. The negative (two's complement) can be formed by treating the redundant number as two separate numbers and forming the negative of each in the conventional manner by complementing and adding a 1 at the least significant bit. If this procedure is done, then the large gaps of zeros in the positive multiple become large gaps of ones in the negative multiple (the bottom of Figure 2.14). In the worst case (all partial products negative), summing the partially redundant partial products requires as much hardware as representing them in the fully redundant form. It would have been better to just stick with the fully redundant form in the first place, rather than require small adders to make the partially 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 2.13 cannot meet both of these

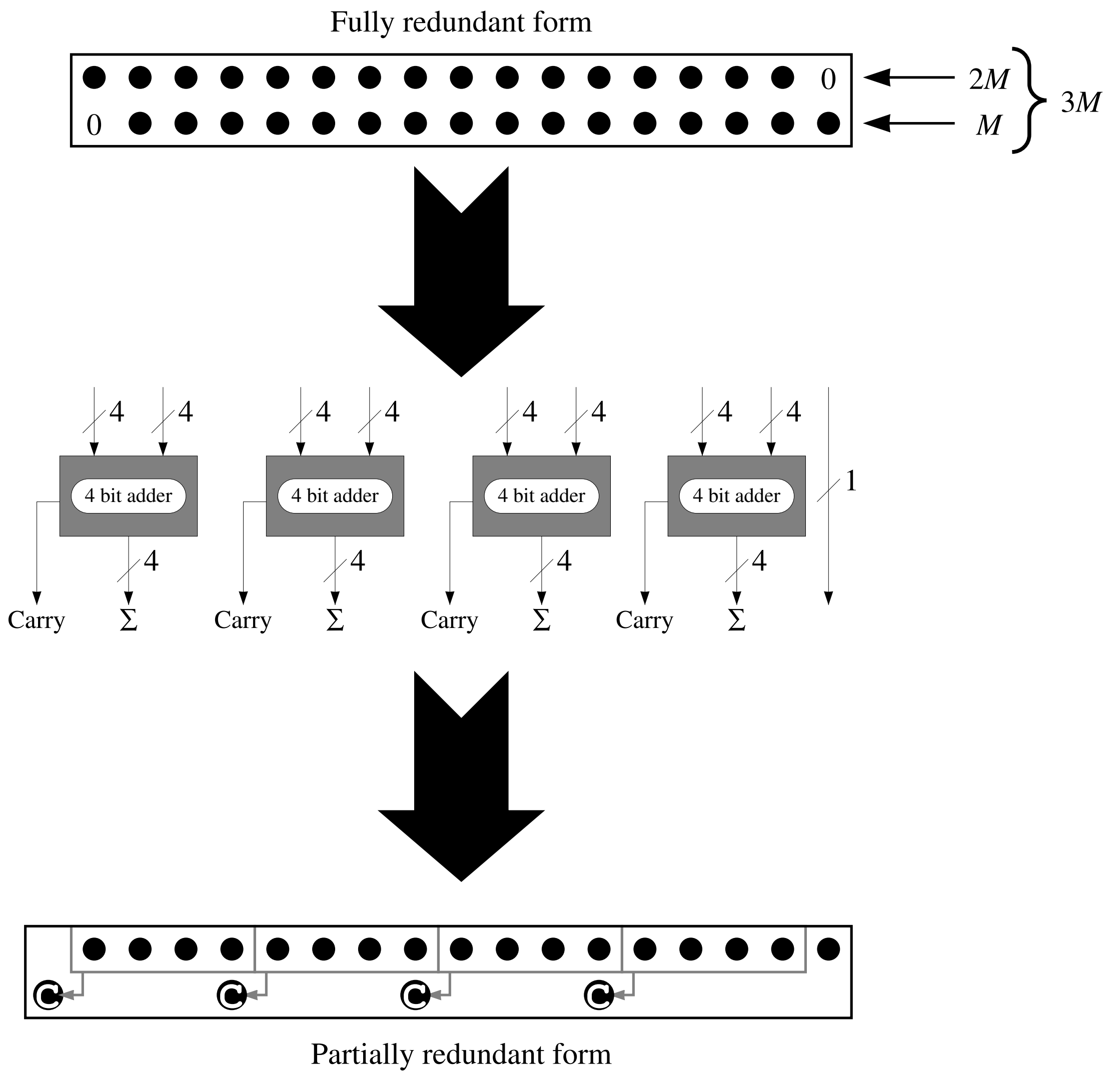


Figure 2.13: Computing 3M in a partially redundant form.

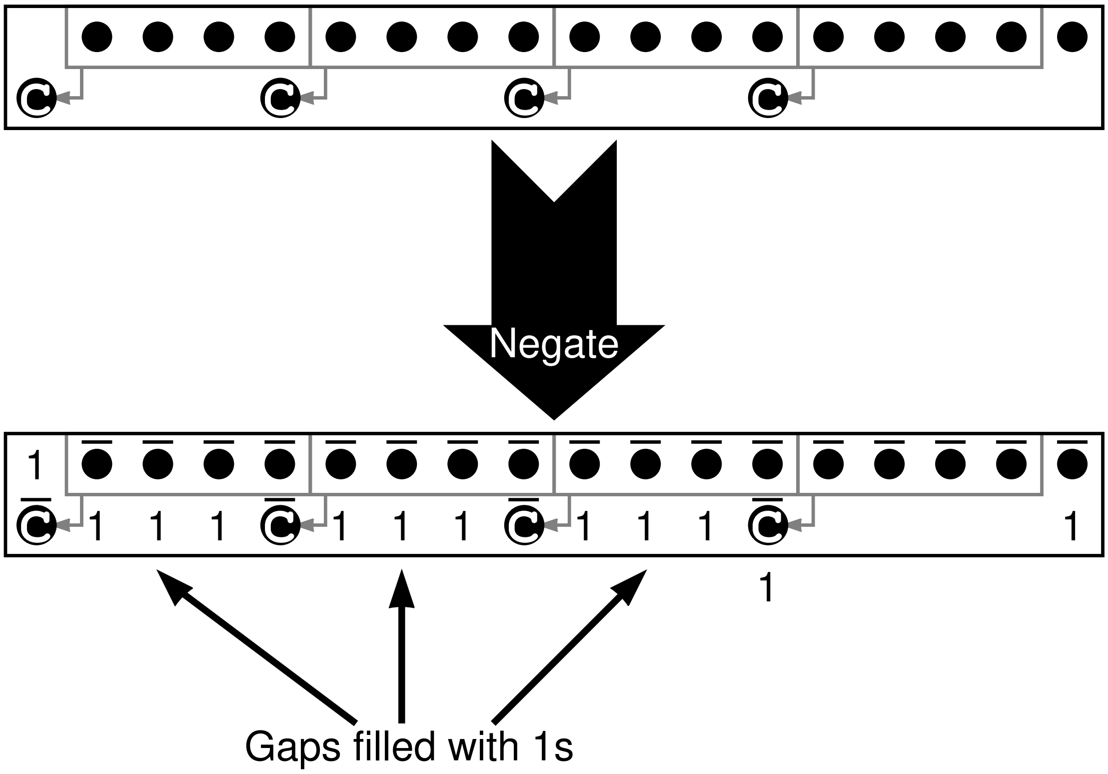
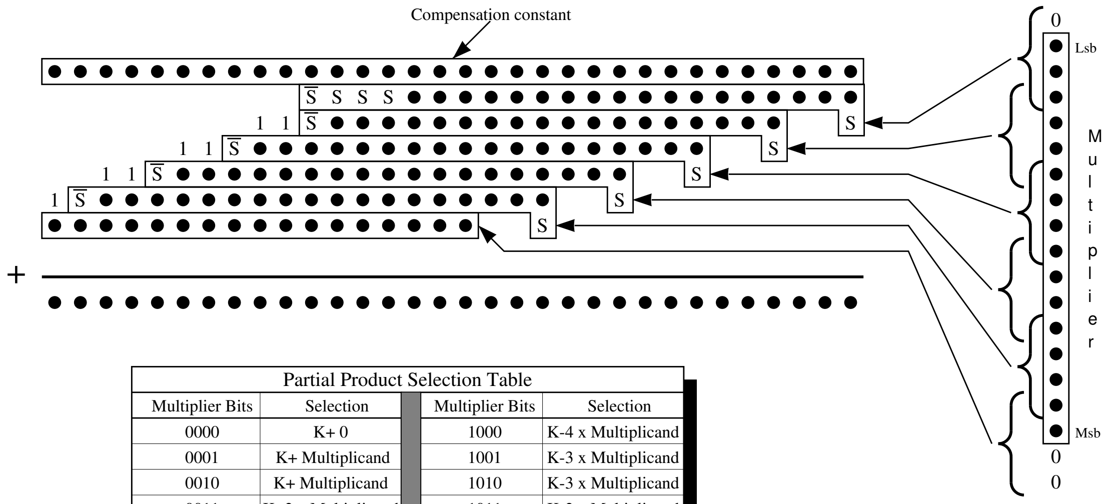


Figure 2.14: Negating a number in partially redundant form.

conditions simultaneously.

2.2.3 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 2.15. Each partial product has a bias constant



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 2.15: Booth 3 with bias.

added to it before being summed to form the final product. The bias constant (K) is the same for both positive and negative multiples[1] of a single partial product, but different partial products can have different bias constants. The only restriction is that K, for a given partial product, cannot depend on the particular multiple 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.

[1]the entries from the right side of the table in Figure 2.15 will continue to be considered as negative multiples

The value of the bias constant K is chosen in such a manner that the creation of negative partial products is a simple operation, as it is for the conventional Booth algorithms. To find an appropriate value for this constant, consider a multiple in the partially redundant form of Figure 2.13 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 2.16. The topmost circled

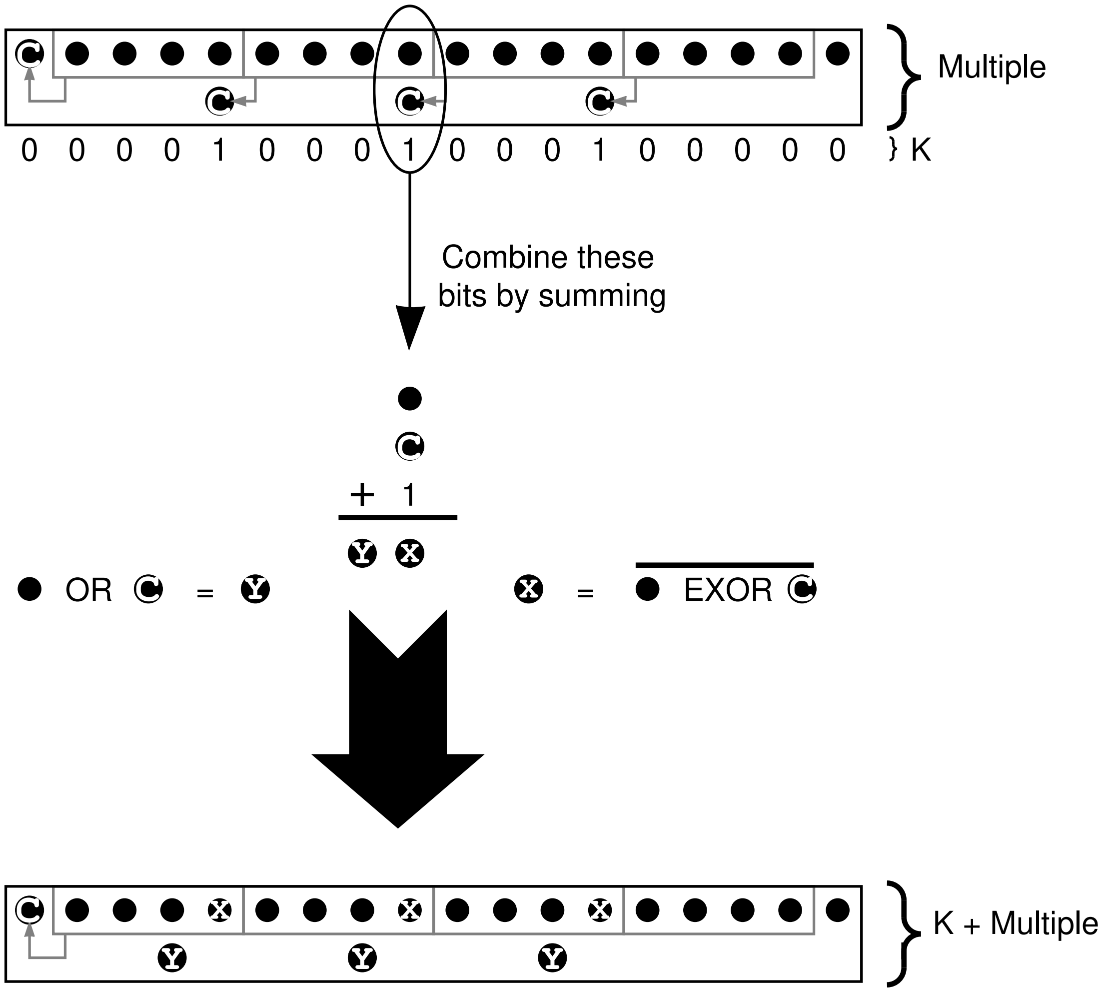


Figure 2.16: Transforming the simple redundant form.

section enclosing 3 vertical items (two dots and the constant 1) 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 $\mathrm{K} - \mathrm{Multiple}$ can be obtained very simply from the value of $\mathrm{K} + \mathrm{Multiple}$.

Figure 2.17 shows the sum of $\mathrm{K} + \mathrm{Multiple}$ with a value Z which is formed by the bit by bit complement of the non-zero portions of $\mathrm{K} + \mathrm{Multiple}$ and the constant 1 in the lsb. When these two values are summed together, the result is 2K (this assumes proper sign

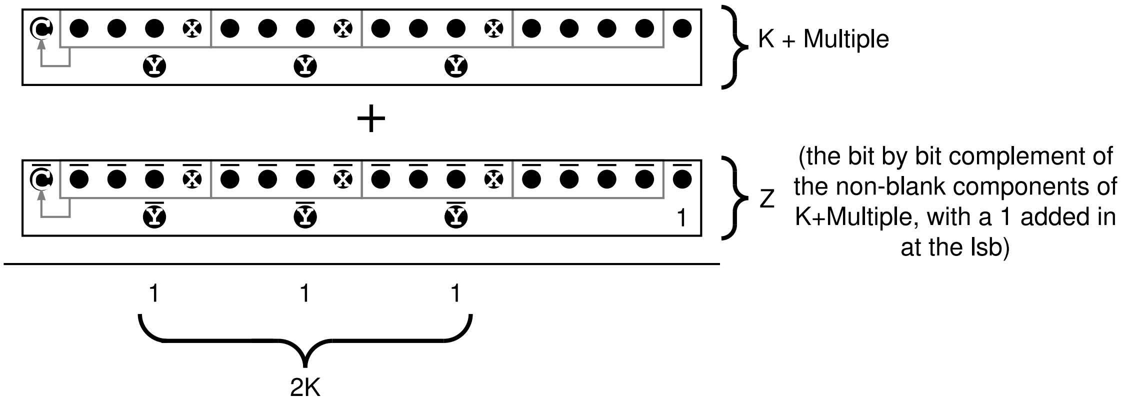


Figure 2.17: Summing $\mathrm{K} - \mathrm{Multiple}$ and Z.

extension to however many bits are desired). That is :

$$\begin{aligned} \mathrm{K} + \mathrm{Multiple} + \mathrm{Z} &= 2\mathrm{K} \\ \mathrm{Z} &= \mathrm{K} - \mathrm{Multiple} \end{aligned}$$

In short, $\mathrm{K} - \mathrm{Multiple}$ can be obtained from $\mathrm{K} + \mathrm{Multiple}$ by complementing all of the non-blank bits of $\mathrm{K} + \mathrm{Multiple}$ 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.

The process behind the determination of the proper value for K can be understood by deducing the same result in a slightly different method. First, assume that a partial product, PP, is to be represented in a partially redundant form using the numbers X and Y, with Y having mostly zeroes in it's binary representation. Let PP be equal to the sum of the three numbers A,B, and the bias constant K. That is :

$$\mathrm{PP} = \mathrm{A} + \mathrm{B} + \mathrm{K}$$

The partially redundant form can be written in binary format as :

$$
\begin{aligned}
\mathrm{PP} &= \mathrm{A}+\mathrm{B}+\mathrm{K} \\
&= \mathrm{X}+\mathrm{Y} \\
&= \left\{ \begin{array}{llllllll}
\mathrm{X}_{n-1} & \mathrm{X}_{n-2}\ldots & \mathrm{X}_{k}\ldots & \mathrm{X}_{i}\ldots & \mathrm{X}_{1} & \mathrm{X}_{0} & + \\
0 & 0\ldots & \mathrm{Y}_{k}0\ldots & \mathrm{Y}_{i}0\ldots & 0 & 0 &
\end{array} \right.
\end{aligned}
$$

The desired behaviour is to be able to "negate" the partial product P, by complementing all the bits of X and the non-zero components of Y, and then adding 1. It is not really negation, because the bias constant K, must be the same in both the positive and "negative" forms. That is :

$$
\begin{aligned}
\text{"negative" of PP} &= -(\mathrm{A}+\mathrm{B})+\mathrm{K} \\
&= \left\{ \begin{array}{lllllll}
\overline{\mathrm{X}}_{n-1} & \overline{\mathrm{X}}_{n-2}\cdots & \overline{\mathrm{X}}_{k}\cdots & \overline{\mathrm{X}}_{i}\cdots & \overline{\mathrm{X}}_{1} & \overline{\mathrm{X}}_{0} & +1+ \\
0 & 0\cdots & \overline{\mathrm{Y}}_{k}0\cdots & \overline{\mathrm{Y}}_{i}0\cdots & 0 & 0 &
\end{array} \right.
\end{aligned}
\tag{2.1}
$$

Now if PP is actually negated in 2's complement form it gives :

$$
\begin{aligned}
-\mathrm{PP} &= -(\mathrm{A}+\mathrm{B}+\mathrm{K}) \\
&= \left\{ \begin{array}{lllllll}
\overline{\mathrm{X}}_{n-1} & \overline{\mathrm{X}}_{n-2}\cdots & \overline{\mathrm{X}}_{k}\cdots & \overline{\mathrm{X}}_{i}\cdots & \overline{\mathrm{X}}_{1} & \overline{\mathrm{X}}_{0} & +1+ \\
1 & 1\cdots & \overline{\mathrm{Y}}_{k}1\cdots & \overline{\mathrm{Y}}_{i}1\cdots & 1 & 1 & +1
\end{array} \right.
\end{aligned}
\tag{2.2}
$$

So all the long strings of 0's in Y have become long strings of 1's, as mentioned previously. The undesirable strings of 1's can be pulled out and assembled into a separate constant, and the "negative" of PP can be substituted :

$$
\begin{aligned}
-\mathrm{PP} &= \left\{ \begin{array}{lllllll}
\overline{\mathrm{X}}_{n-1} & \overline{\mathrm{X}}_{n-2}\cdots & \overline{\mathrm{X}}_{k}\cdots & \overline{\mathrm{X}}_{i}\cdots & \overline{\mathrm{X}}_{1} & \overline{\mathrm{X}}_{0} & +1+ \\
0 & 0\cdots & \overline{\mathrm{Y}}_{k}0\cdots & \overline{\mathrm{Y}}_{i}0\cdots & 0 & 0 & + \\
1 & 1\cdots & 01\cdots & 01\cdots & 1 & 1 & +1
\end{array} \right. \\
&= \left\{ \begin{array}{l}
\text{"negative" of PP} + \\
1 \quad 1\cdots \quad 01\cdots \quad 01\cdots \quad 1 \quad 1 \quad +1
\end{array} \right.
\end{aligned}
$$

Finally, substituting Equations 2.2 and 2.1 and simplifying :

$$
-(\mathrm{A}+\mathrm{B}+\mathrm{K}) = -(\mathrm{A}+\mathrm{B})+\mathrm{K}+
$$

$$
\begin{array}{rcllllllll}
 & & 1 & 1\cdots & 01\cdots & 01\cdots & 1 & 1 & +1 \\
-2\mathrm{K} & = & 1 & 1\cdots & 01\cdots & 01\cdots & 1 & 1 & +1 \\
2\mathrm{K} & = & 0 & 0\cdots & 10\cdots & 10\cdots & 0 & 0 &
\end{array}
$$

which again gives the same value for K. The partially redundant form described above 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.

Producing the multiples

Figure 2.18 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

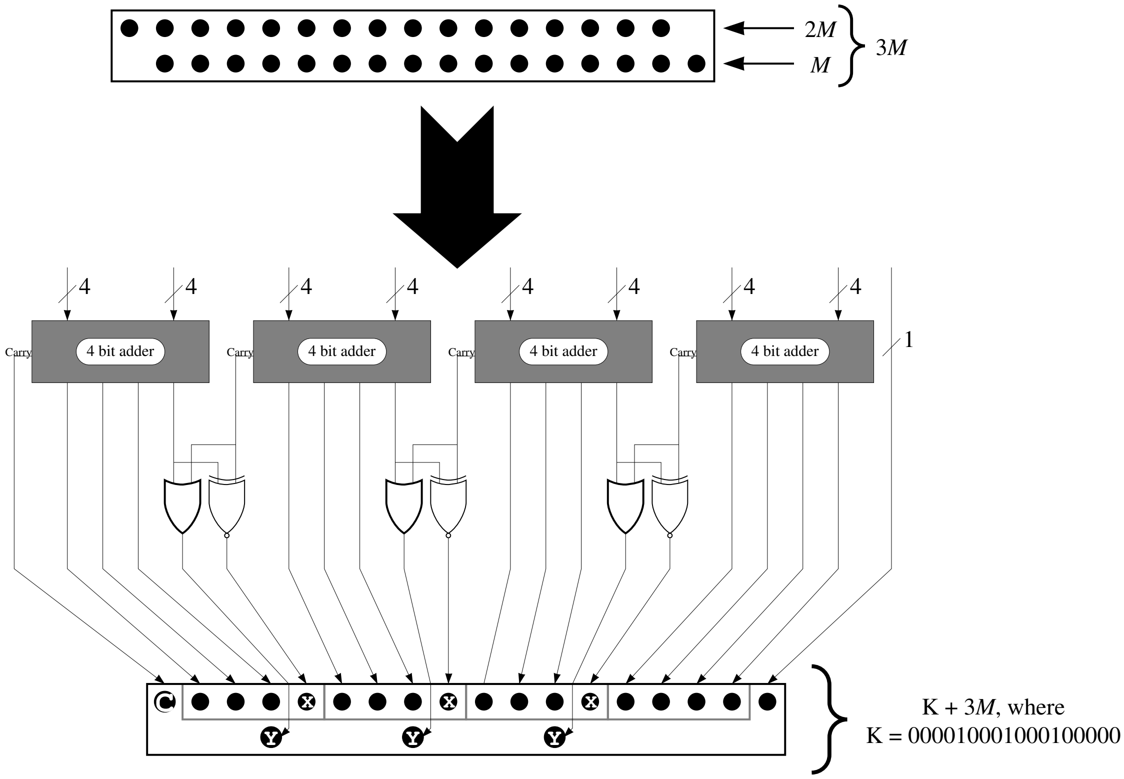


Figure 2.18: Producing K + 3M in partially redundant form.

the time needed to 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 2.19. In this figure the bits of the multiplicand (M) are numbered (lsb = 0) so that

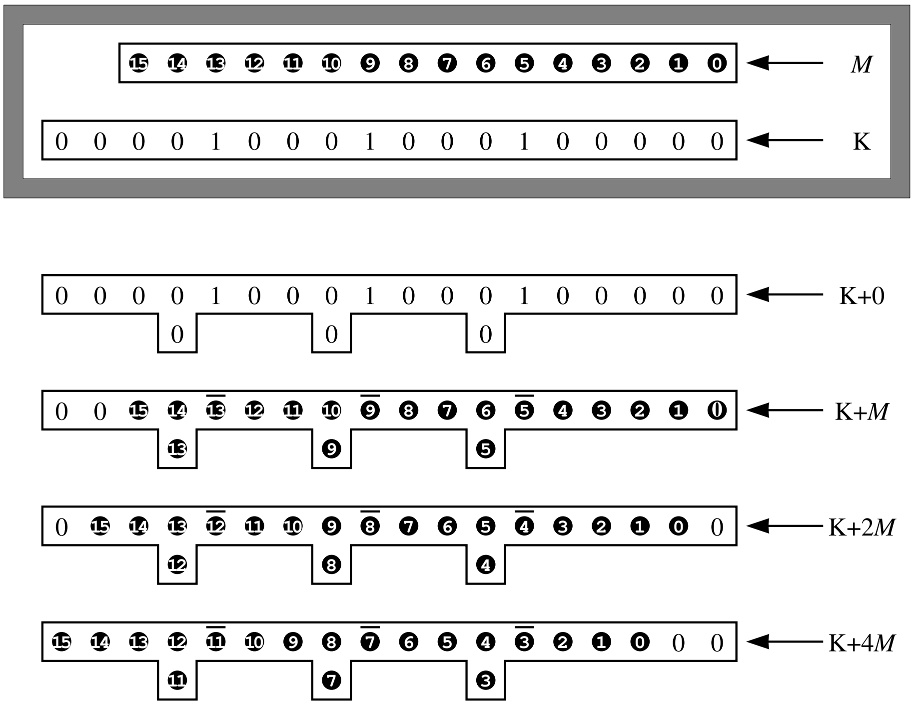


Figure 2.19: Producing other multiples.

the source of each bit in each multiple can be easily seen.

2.2.4 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 2.20 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

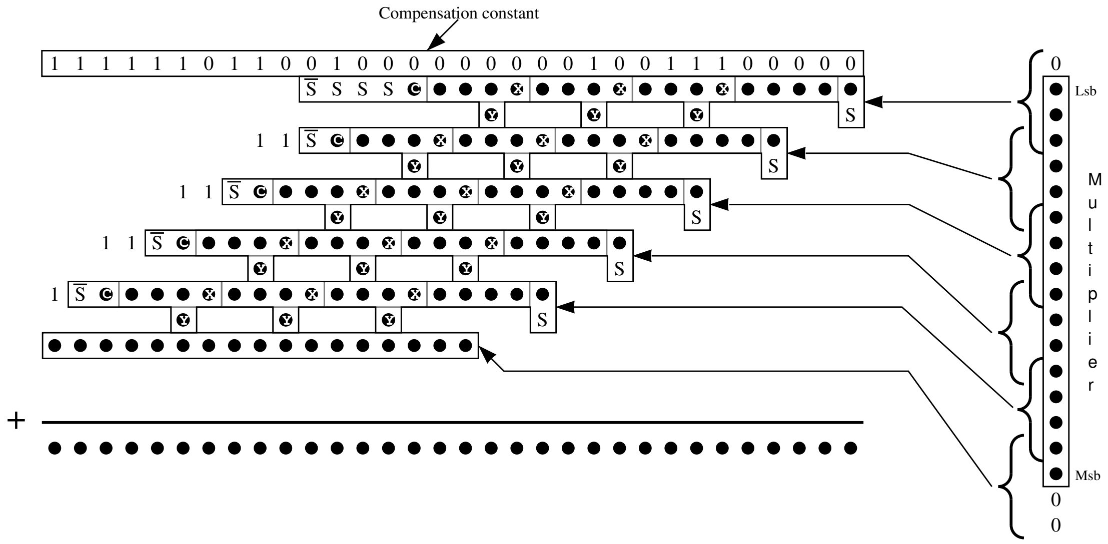


Figure 2.20: 16 x 16 redundant Booth 3.

3 algorithm (previously given as 126). The height [2] 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 2.21. This example uses 4 bit adders as per Figure 2.18 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 2.22. This figure shows how a single partial product is built out of the multiplicand and K + 3M generated by logic in Figure 2.18.

2.2.5 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 2.23 and is concerned with the biased multiples 3M and 6M. The left side of the figure shows the format of $K + 3M$. The problem arises when the biased multiple

[2] 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.

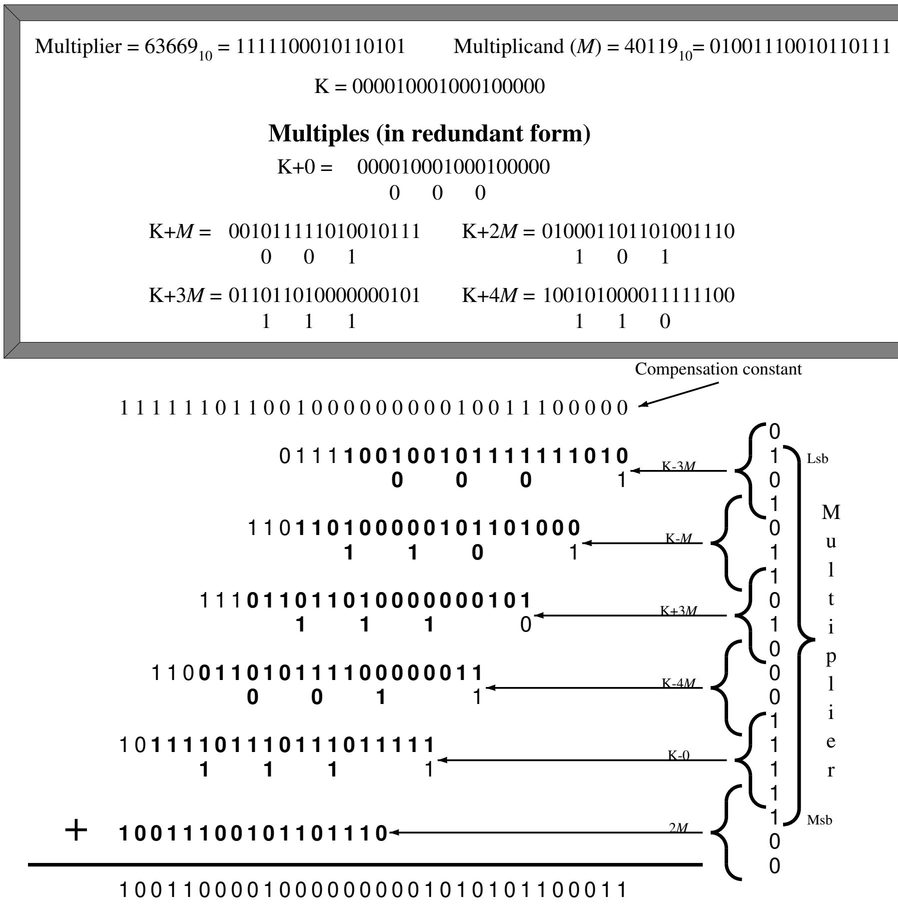


Figure 2.21: 16 bit partially redundant Booth 3 multiply.

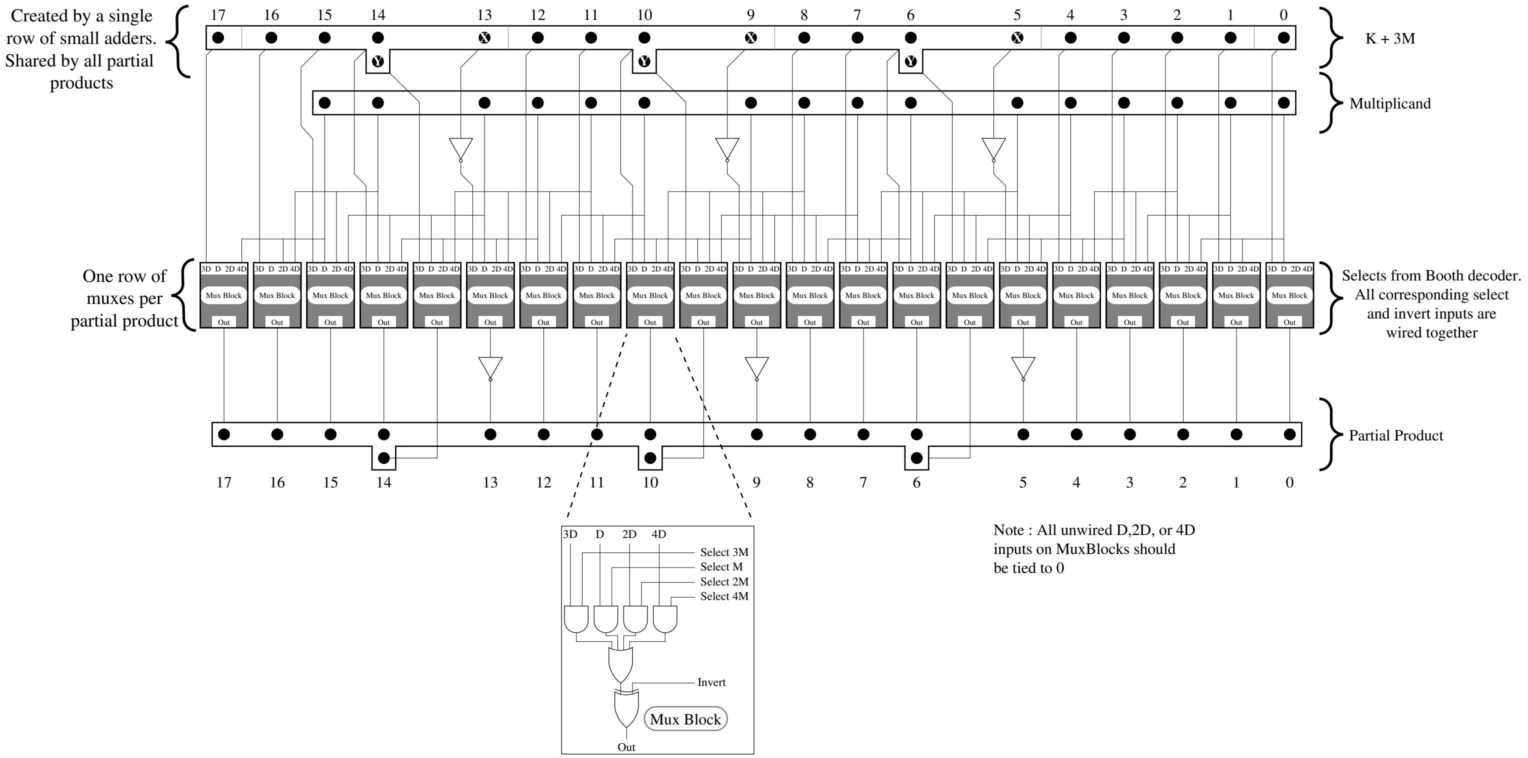


Figure 2.22: Partial product selector for redundant Booth 3.

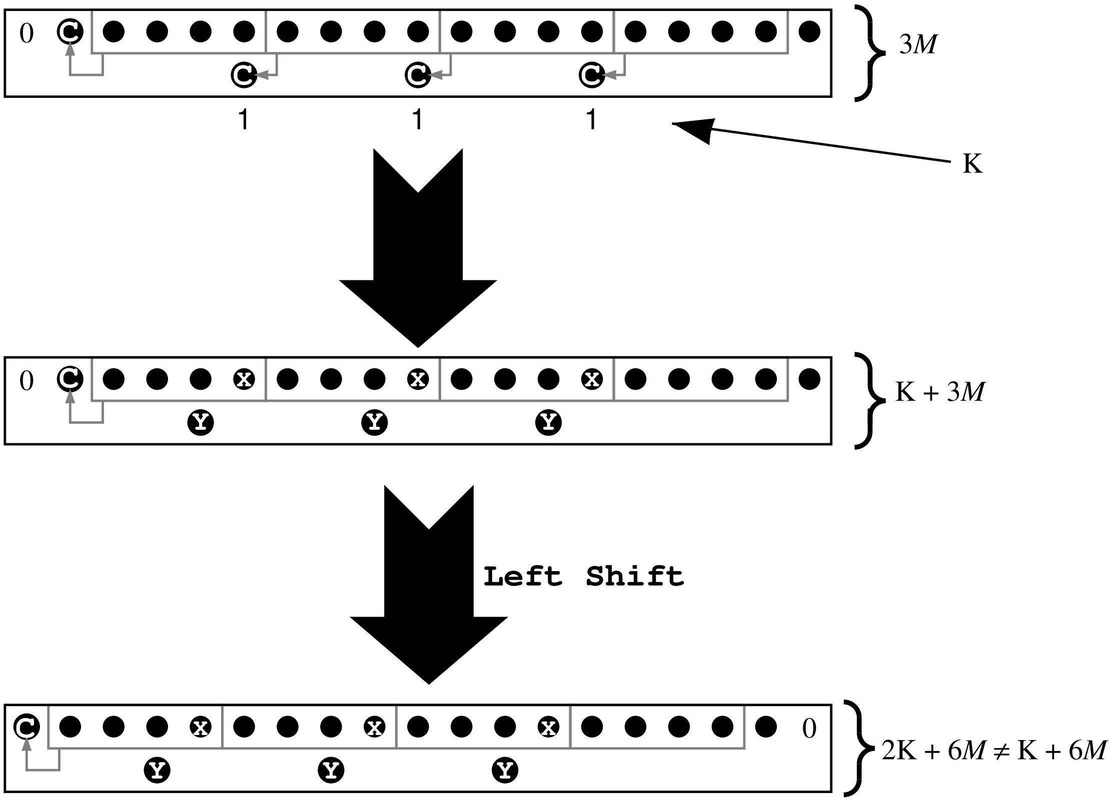


Figure 2.23: Producing K + 6M from K + 3M ?

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 2.24. 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 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 Chapter 5 will show that the non-redundant Booth 4 algorithm offers no performance, area, or power advantages over the Booth 3 algorithm for reasonable ($\leq$ 64 bits) length algorithms. As a result the redundant Booth 4 algorithm is not very interesting. 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, the extra wires (long) needed to distribute the multiples

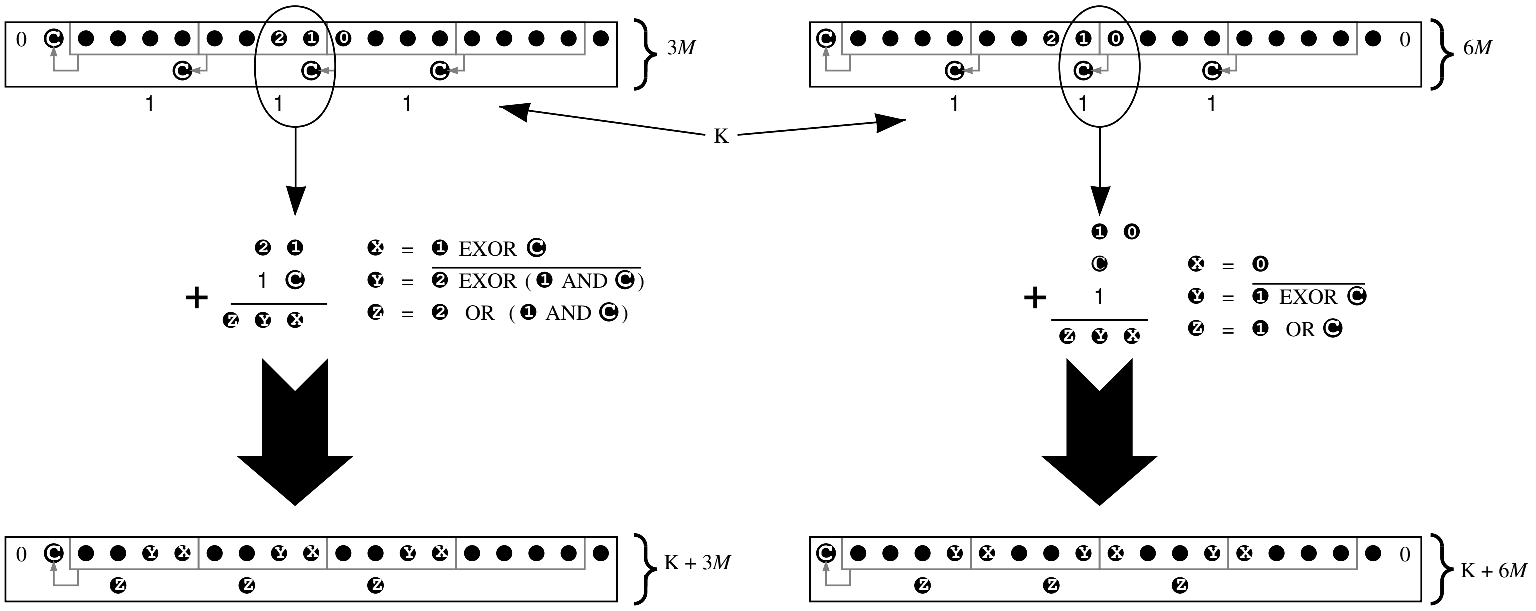


Figure 2.24: A different bias constant for 6M and 3M.

to the partial product multiplexers, and the increased complexity of the partial product multiplexers themselves.

2.2.6 Choosing the Adder Length

By and large, the rule for choosing the length of the small adders necessary for is straightforward - use largest possible adder that does not increase the latency of multiply. 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 2.25. This figure assumes

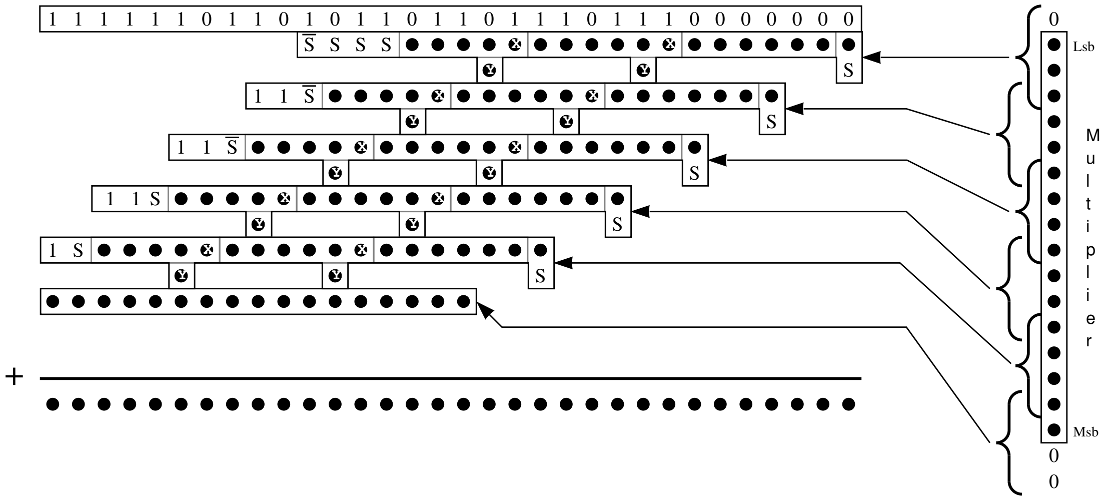


Figure 2.25: Redundant Booth 3 with 6 bit adders.

a redundant 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.

2.3 Summary

This chapter 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. Since Booth's algorithm requires negative partial products, the key idea in using the partially redundant representation is to add a carefully chosen constant to each partial product, which allows the partial product to be easily negated. A detailed evaluation of implementations using this algorithm is presented in Chapter 5, including comparisons with implementations using more conventional algorithms.

Chapter 3

Adders for Multiplication

Fast carry propagate adders are important to high performance multiplier design in two ways. First, an efficient and fast adder is needed to make any "hard" multiples that are needed in partial product generation. Second, after the partial products have been summed in a redundant form, a carry propagate adder is needed to produce the final non-redundant product. Chapter 5 will show that the delay of this final carry propagate sum is a substantial portion of the total delay through the multiplier, so minimizing the adder delay can make a significant contribution to improving the performance of the multiplier. This chapter presents the design of several high performance adders, both general purpose and specialized. These adders will then be used in Chapter 5 to evaluate overall multiplier designs.

3.1 Definitions and Terminology

The operands to be added are n bit binary numbers, A and B, with resultant binary sum S (also n bits long). The single bit carry-in to the summation will be denoted by c_0 and the carry-out by c_n. A,B, and S can be expanded directly in binary representation. For example, the binary representation representation for A is :

$$A = \sum_{k=0}^{n-1} a_k \cdot 2^k \qquad\qquad a_k \in (0, 1)$$

with similar expansions for B, and S.

The following notation for various boolean operators will be used :

$$
\begin{aligned}
\mathrm{a\,b} &\mapsto \text{boolean AND of a,b} \\
\mathrm{a + b} &\mapsto \text{boolean OR of a,b} \\
\mathrm{a \oplus b} &\mapsto \text{EXCLUSIVE OR of a,b} \\
\overline{\mathrm{a}} &\mapsto \text{boolean NOT of a} \\
\overline{\mathrm{A}} &\mapsto \sum_{k=0}^{n-1} \overline{a_k} \cdot 2^k \quad \text{the bit by bit complement of the binary number A}
\end{aligned}
$$

To avoid ambiguity, the symbol $\overset{\text{sum}}{+}$ will be used to signify actual addition of binary numbers.

The defining equations for the binary addition of A, B, and c_0, giving sum S and c_n will be taken as :

$$s_k = a_k \oplus b_k \oplus c_k \tag{3.1}$$

$$c_{k+1} = a_k\, b_k + a_k\, c_k + b_k\, c_k \tag{3.2}$$

$$k = 0, 1, \ldots, n-1$$

In developing the algebra of adders, the auxiliary functions p (carry propagate) and g (carry generate) will be needed, and are defined by a modified version of equation 3.2:

$$c_{k+1} = g_k + p_k\, c_k \tag{3.3}$$

Combining equations 3.3 and 3.2 gives the definition of g and two possible definitions for p

$$g_k = a_k\, b_k \tag{3.4}$$

$$p_k = a_k + b_k \tag{3.5}$$

$$ = a_k \oplus b_k \tag{3.6}$$

In general, the two definitions of p_k are interchangeable. Where it is necessary to distinguish between the two possible p definitions (most importantly in the Ling adder), the first form is referred to as ${p^+}_k$ and the second form as ${p^\oplus}_k$.

Equation 3.3 gives the carry out from a given bit position in terms of the carry-in to that position. This equation can also be applied recursively to give c_{k+1} in terms of a lower order carry. For example, applying (3.3) three times gives c_{k+1} in terms of c_{k-2} :

$$c_{k+1} = g_k + p_k\, g_{k-1} + p_k\, p_{k-1}\, g_{k-2} + p_k\, p_{k-1}\, p_{k-2}\, c_{k-2} \tag{3.7}$$

This leads to two additional functions which can be defined :

$$g_k^j = g_j + p_j\, g_{j-1} + p_j\, p_{j-1}\, g_{j-2} + \cdots + p_j\, p_{j-1} \cdots p_{k+1} g_k \tag{3.8}$$

$$p_k^j = p_j\, p_{j-1}\, p_{j-2} \cdots p_{k+1}\, p_k \tag{3.9}$$

Equations 3.8 and 3.9 give the carry generate and propagate for the range of bits from k to j. These equations form the basis for the conventional carry lookahead adder [38].

3.1.1 Positive and Negative Logic

Before presenting the design examples, a simple theorem relating positive logic adders (where a "1" is represented by a high voltage) and negative logic adders (a "1" is represented by a low voltage) will be stated. The proof for this theorem is presented in Appendix C. This theorem is important because it allows transformation of inputs or outputs to better fit the inverting nature of implementations of most conventional logic, and to avoid the use of inefficient logic functions. For example, ECL can provide efficient and fast NOR/OR gates, but NAND/AND gates are slower, larger and consume more power. Replacement of NAND/AND gates with NOR/OR gates will produce better ECL implementations.

Theorem 1 *Let A and B be positive logic binary numbers, each n bits long, and* c_0 *be a single carry bit. Let S be the n bit sum of A, B, and* c_0*, and let* c_n *be the carry out from the summation. That is :*

$$2^n \cdot c_n \overset{\text{sum}}{+} S = A \overset{\text{sum}}{+} B \overset{\text{sum}}{+} c_0$$

Then :

$$2^n \cdot \overline{c_n} \overset{\text{sum}}{+} \overline{S} = \overline{A} \overset{\text{sum}}{+} \overline{B} \overset{\text{sum}}{+} \overline{c_0}$$

Theorem 1 is simply stating that a positive adder is also a negative logic adder. Or in other words, an adder designed to function with positive logic inputs and outputs will also be an adder if the inputs and outputs are negative logic.

3.2 Design Example - 64 bit CLA adder

The first design example to be presented is that of a conventional 64 bit carry lookahead adder (CLA) [38]. Figure 3.1 shows an overall block diagram of the adder. The input operands, A and B, and the sum output, S, are assumed to be negative logic, while the carry-in and carry-out are assumed to be positive logic. The 64 bit A and B input operands are partitioned into 16 four bit groups. Each group has Group Generate and Propagate Logic which computes a group carry generate (G) and a group carry propagate (P). The Carry Lookahead Logic in the center of the figure combines the G and P signals from each group with the carry-in signal to produce 16 group carries ($c_k, k = 0, 4, \ldots, 60$) and the adder carry-out (c_{64}). Each four bit group has an Output Stage which uses the corresponding group carries to produce a 4 bit section of the final 64 bit sum.

3.2.1 Group Logic

The group generate logic, group propagate logic, and the final output stage for each 4 bit section can be combined into a single modular logic section. A possible gate level implementation for a four bit group is shown in Figure 3.2. Complex or multiple gates contained within dotted boxes represent logic which can be implemented with a single ECL tail current. The individual bit g_k and p_k, (k=0,1,2,3), are produced by the gates labeled Y, and are used to produce the group generate (G) and group propagate (P) signals, as well as being used internally to produce bit to bit carries. G and P for the group are produced by the gates labeled X, according to the following equations :

$$\begin{aligned}
G &= g_0^3 && (3.10)\\
&= g_3 + p_3\, g_2 + p_3\, p_2\, g_1 + p_3\, p_2\, p_1\, g_0 && (3.11)\\
P &= p_0^3 \\
&= p_3\, p_2\, p_1\, p_0 && (3.12)
\end{aligned}$$

The outputs of individual gates are connected via a wire-OR to produce G. The output stage is formed by gates Z and produces the sum at each bit position by a three way EXCLUSIVE OR of a_k and b_k with the carry (c_k) reaching a particular bit. The carry

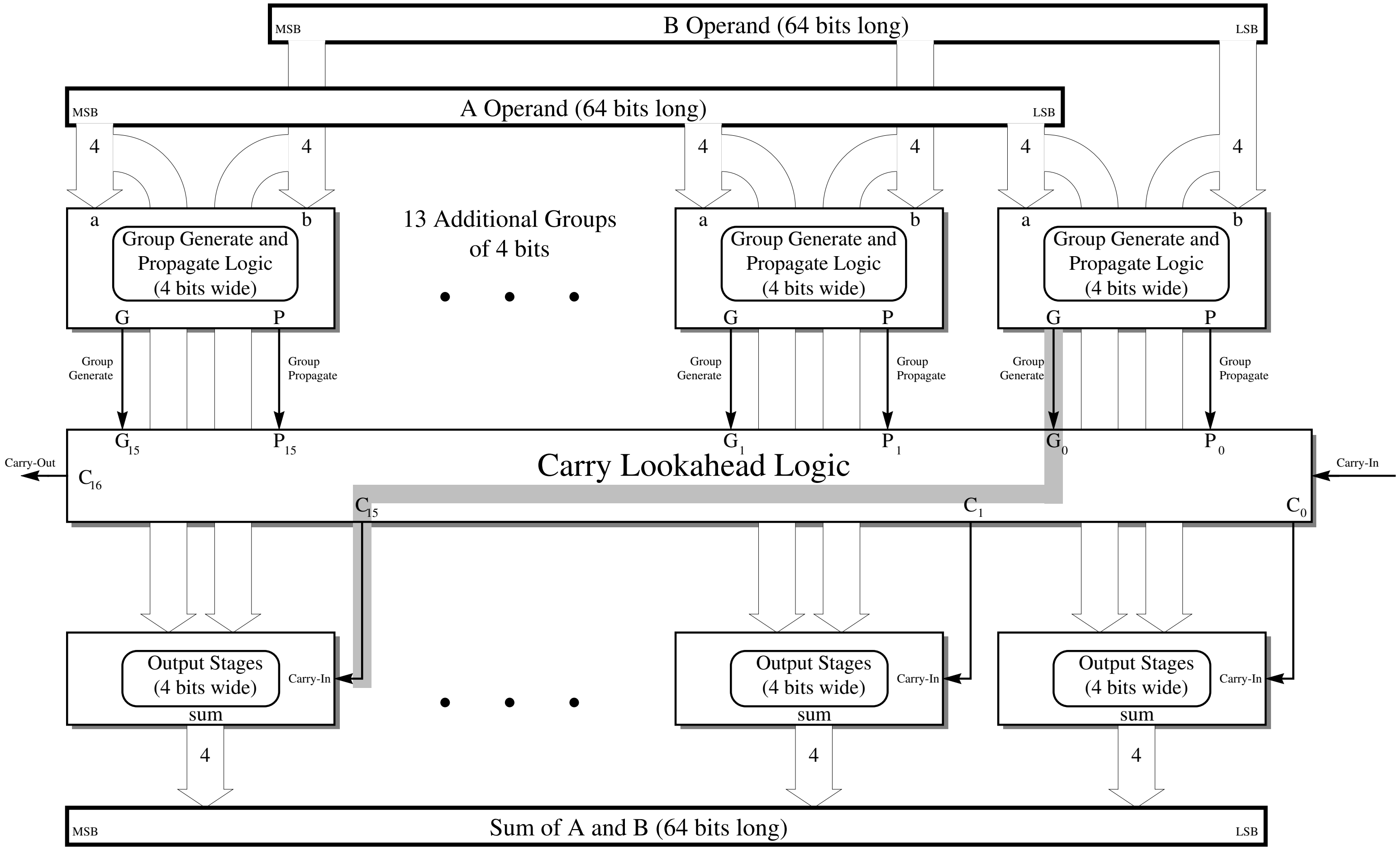


Figure 3.1: Carry lookahead addition overview.

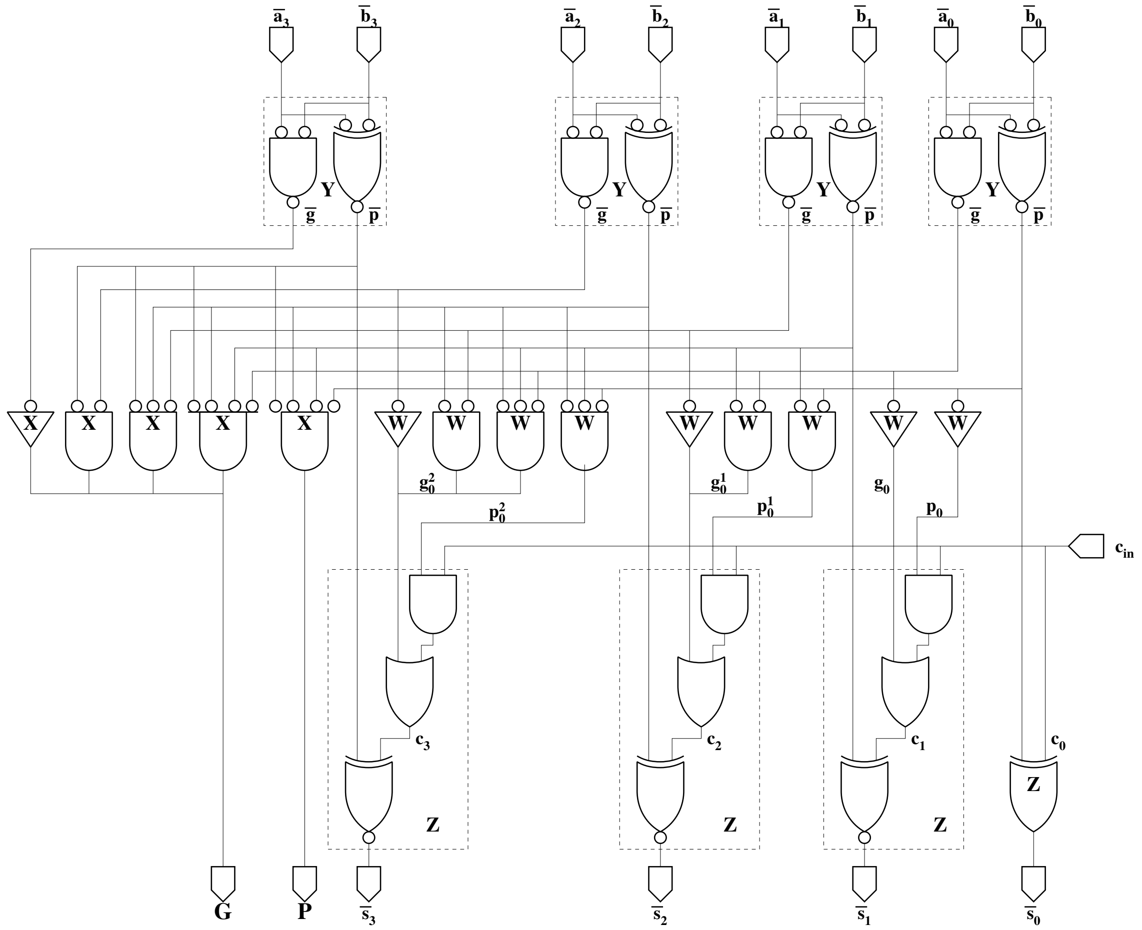


Figure 3.2: 4 bit CLA group.

reaching a particular bit can be related to the group carry-in (c_{in}) by the following :

$$c_k = g_0^k + p_0^k c_{in}$$

The signal c_{in} usually arrives later than the other signals, (since it comes from the global carry lookahead logic which contains long wires), so the logic needs to be optimized to minimize the delay along the c_{in} path. This is done by using Shannon's Expansion Theorem [27] [28] applied to $\overline{s_k}$ as a function of c_{in} :

$$\overline{s_k} = \overline{a_k} \oplus b_k \oplus c_k$$

$$\begin{aligned} &= \overline{a_k} \oplus b_k \oplus \left(g_0^k + p_0^k c_{in}\right) \\ &= c_{in} \left(\overline{a_k} \oplus b_k \oplus \left[g_0^k + p_0^k\right]\right) + \overline{c_{in}} \left(\overline{a_k} \oplus b_k \oplus \left[g_0^k\right]\right) \end{aligned} \tag{3.13}$$

Being primary inputs, $\overline{a_k}$ and $\overline{b_k}$ are available very early, so the value $\overline{a_k} \oplus b_k = \overline{\overline{a_k} \oplus \overline{b_k}} = \overline{p_k}$ is also available fairly early. The values g_0^k and p_0^k can be produced using only locally available signals (that is signals available within the group). Because the wires within a group should be fairly short, these signals should also be available rather quickly (the gates labeled W in Figure 3.2 produce these signals). The detailed circuitry for an output stage gate which realizes equation 3.13, given $a_k \oplus b_k$ (the half sum) with a single tail current is shown in Figure 3.3. This gate is optimized in such a way that the carry to output delay is

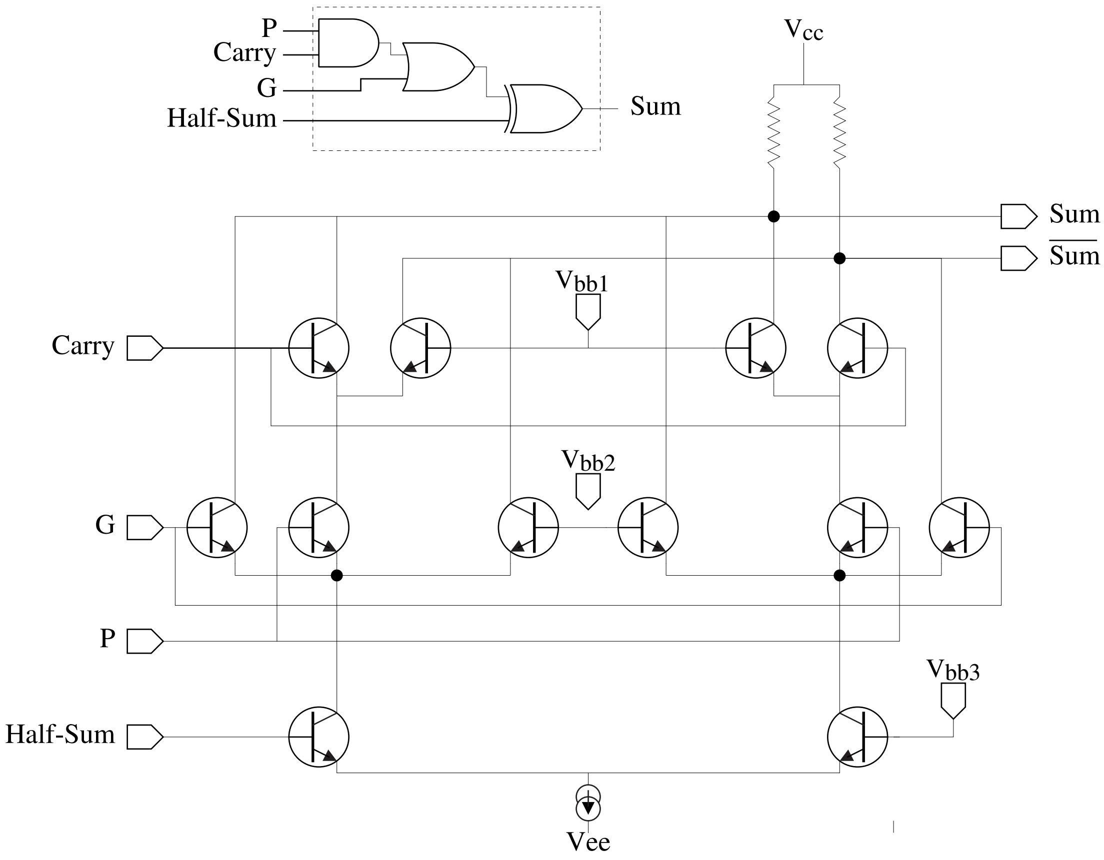


Figure 3.3: Output stage circuit. For proper operation, G and P must not both be high.

much smaller than the delay from the other inputs of the gate.

3.2.2 Carry Lookahead Logic

The carry lookahead logic which produces the individual group carries is illustrated in Figure 3.4. The carries are produced in two stages. Since the group G and P signals are positive logic coming from the groups, the first stage is set up in a product of sums manner (i.e. the first stage is OR-AND-INVERT logic, which can be efficiently implemented in ECL using NOR gates and wire-OR). The first stage of the carry lookahead logic produces supergroup G and P for 1 to 4 groups according to the following :

$$
\begin{aligned}
\overline{G_0^0} &= \overline{G_0} \\
\overline{P_0^0} &= \overline{P_0} \\
\\
\overline{G_0^1} &= \overline{(P_1 + G_1)\,(G_1 + G_0)} \\
\overline{P_0^1} &= \overline{P_1} + \overline{P_0} \\
\\
\overline{G_0^2} &= \overline{(P_2 + G_2)\,(G_2 + P_1 + G_1)\,(G_2 + G_1 + G_0)} \\
\overline{P_0^2} &= \overline{P_2} + \overline{P_1} + \overline{P_0} \\
\\
\overline{G_0^3} &= \overline{(P_3 + G_3)\,(G_3 + P_2 + G_2)\,(G_3 + G_2 + P_1 + G_1)\,(G_3 + G_2 + G_1 + G_0)} \\
\overline{P_0^3} &= \overline{P_3} + \overline{P_2} + \overline{P_1} + \overline{P_0}
\end{aligned}
$$

A gate level implementation of the supergroup G and P using NOR gates and wire-OR is shown in Figure 3.5. Note that some gates have multiple outputs. These can usually be obtained by adding multiple emitter followers at the outputs, or by duplicating the gates in question. The second stage of the carry lookahead logic uses the supergroup G and P produced in the first stage, along with the carry-in, to make the final group carries, which are then distributed to the individual group output stages. This process is similar to the *canonic addition* described in [36]. The equations relating the super group G and P signals to the final carries are :

$$c_0 = C$$

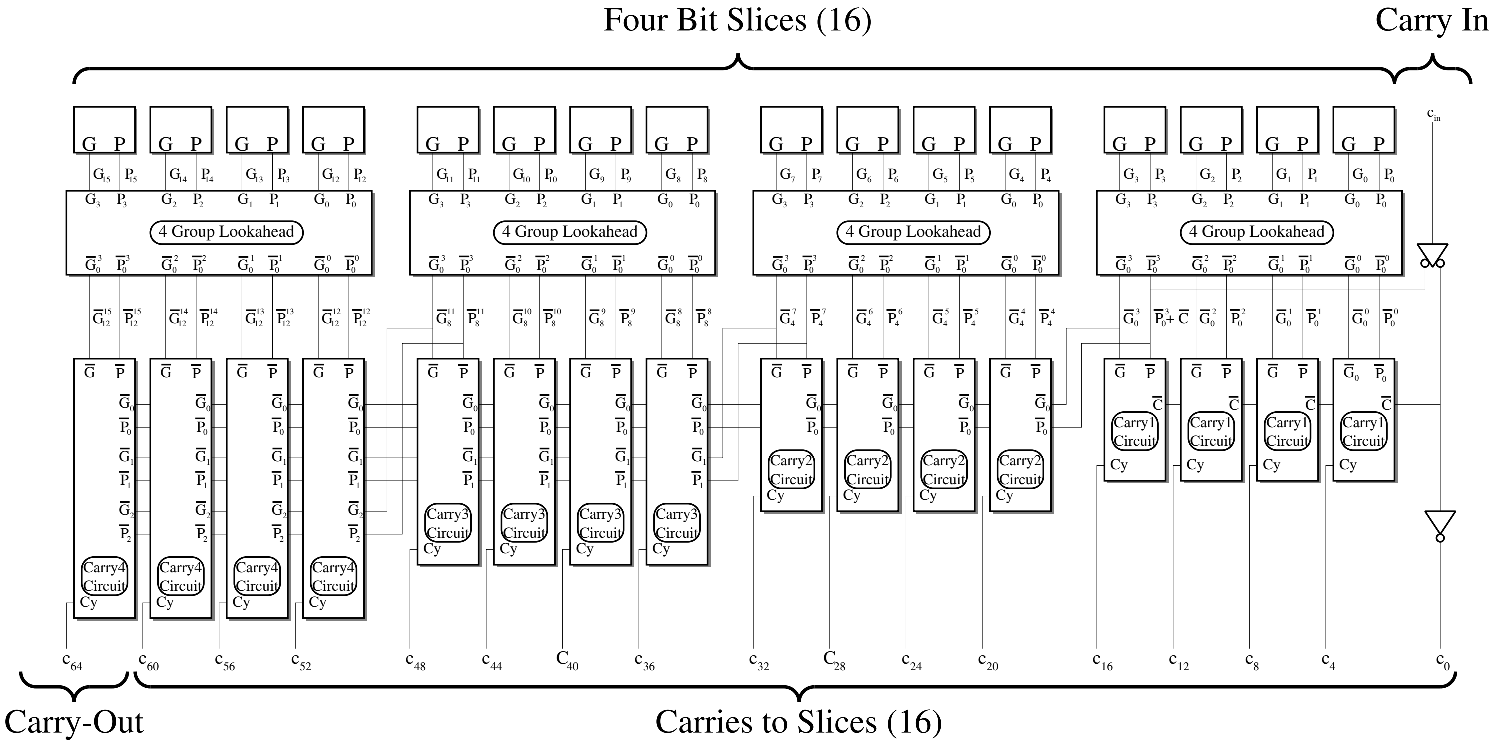


Figure 3.4: Detailed carry connections for 64 bit CLA.

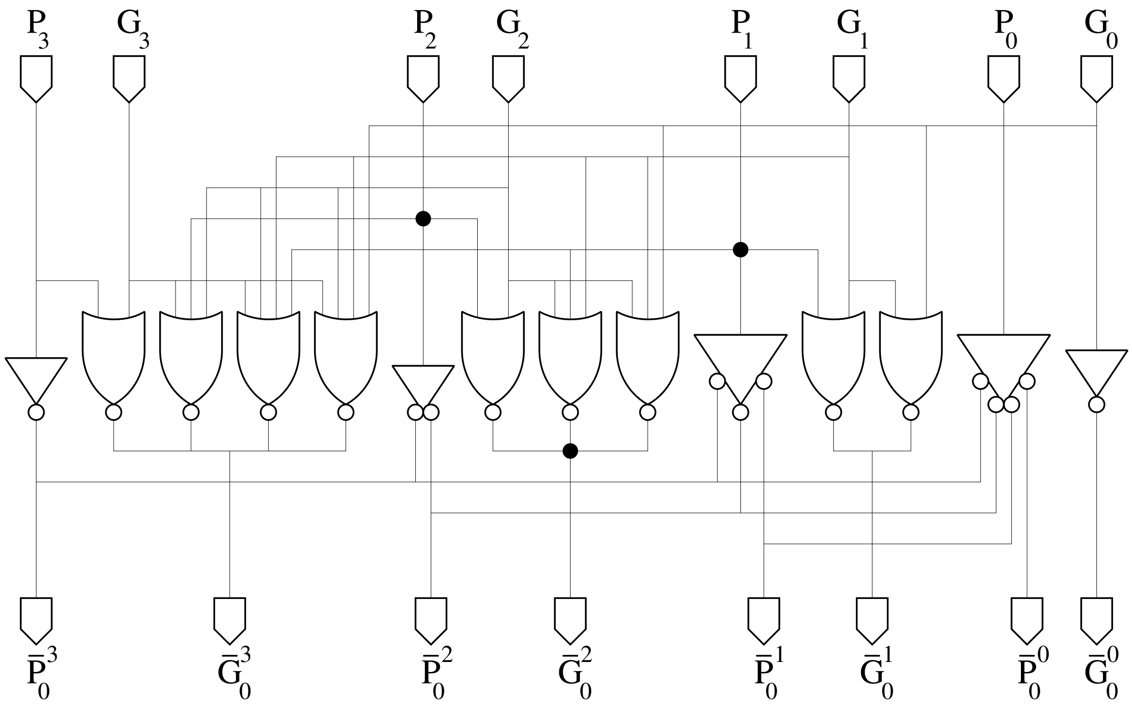


Figure 3.5: Supergroup G and P logic - first stage.

$$
\begin{aligned}
c_4 &= G_0^0 + P_0^0\,C \\
c_8 &= G_0^1 + P_0^1\,C \\
c_{12} &= G_0^2 + P_0^2\,C \\
c_{16} &= G_0^3 + P_0^3\,C \\
c_{20} &= G_4^4 + P_4^4\,G_0^3 + P_4^4\,P_0^3\,C \\
c_{24} &= G_4^5 + P_4^5\,G_0^3 + P_4^5\,P_0^3\,C \\
c_{28} &= G_4^6 + P_4^6\,G_0^3 + P_4^6\,P_0^3\,C \\
c_{32} &= G_4^7 + P_4^7\,G_0^3 + P_4^7\,P_0^3\,C \\
c_{36} &= G_8^8 + P_8^8\,G_4^7 + P_8^8\,P_4^7\,G_0^3 + P_8^8\,P_4^7\,P_0^3\,C \\
c_{40} &= G_8^9 + P_8^9\,G_4^7 + P_8^9\,P_4^7\,G_0^3 + P_8^9\,P_4^7\,P_0^3\,C \\
c_{44} &= G_8^{10} + P_8^{10}\,G_4^7 + P_8^{10}\,P_4^7\,G_0^3 + P_8^{10}\,P_4^7\,P_0^3\,C \\
c_{48} &= G_8^{11} + P_8^{11}\,G_4^7 + P_8^{11}\,P_4^7\,G_0^3 + P_8^{11}\,P_4^7\,P_0^3\,C \\
c_{52} &= G_{12}^{12} + P_{12}^{12}\,G_8^{11} + P_{12}^{12}\,P_8^{11}\,G_4^7 + P_{12}^{12}\,P_8^{11}\,P_4^7\,G_0^3 + P_{12}^{12}\,P_8^{11}\,P_4^7\,P_0^3\,C
\end{aligned}
$$

$$
\begin{aligned}
c_{56} &= G_{12}^{13} + P_{12}^{13} G_{8}^{11} + P_{12}^{13} P_{8}^{11} G_{4}^{7} + P_{12}^{13} P_{8}^{11} P_{4}^{7} G_{0}^{3} + P_{12}^{13} P_{8}^{11} P_{4}^{7} P_{0}^{3} C \\
c_{60} &= G_{12}^{14} + P_{12}^{14} G_{8}^{11} + P_{12}^{14} P_{8}^{11} G_{4}^{7} + P_{12}^{14} P_{8}^{11} P_{4}^{7} G_{0}^{3} + P_{12}^{14} P_{8}^{11} P_{4}^{7} P_{0}^{3} C \\
c_{64} &= G_{12}^{15} + P_{12}^{15} G_{8}^{11} + P_{12}^{15} P_{8}^{11} G_{4}^{7} + P_{12}^{15} P_{8}^{11} P_{4}^{7} G_{0}^{3} + P_{12}^{15} P_{8}^{11} P_{4}^{7} P_{0}^{3} C
\end{aligned}
$$

All of the above functions can be implemented by 4 different INVERT-AND-OR blocks, which are shown in Figure 3.6. Because $\overline{C}$ is connected with a wire-OR to $\overline{P_0^3}$, the maximum number of inputs on any gate is 4, and the maximum number of wire-OR outputs is 5.

3.2.3 Remarks on CLA Example

The 64 bit CLA design presented above combines elements of conventional carry lookahead adders, canonic adders, and conditional sum adders [29]. In addition circuit configurations are chosen to specifically fit circuit tricks that are available with ECL. The result is a reasonably modular, high performance adder. Along the critical path, there are 4 NOR and 1 EXCLUSIVE-OR equivalent stages of gates. The next design example will further increase the performance by reducing the number of logic stages along the critical path, while retaining the same basic modular structure.

3.3 Design Example - 64 Bit Modified Ling Adder

A faster adder can be designed by using a method developed by H. Ling [16]. In the Ling scheme, the group carry generate and propagate (G and P) are replaced by similar functions (called H and I respectively) which can be produced in fewer stages than the group G and P. These signals are distributed around in a manner which is almost identical to that of the group G and P. When a real G or P is needed, it is recreated using H and I plus a single signal which is locally available. The algebra behind this substitution will be presented as needed in the discussion that follows.

An overview of a 64 bit modified Ling adder is shown in Figure 3.7. The structure is very similar to that of the CLA described above, but there are two additional signals which connect adjacent blocks ($\overline{p^{+}_{3}}$ and $\overline{p^{+}_{3}}$(dot)). There are also minor differences in

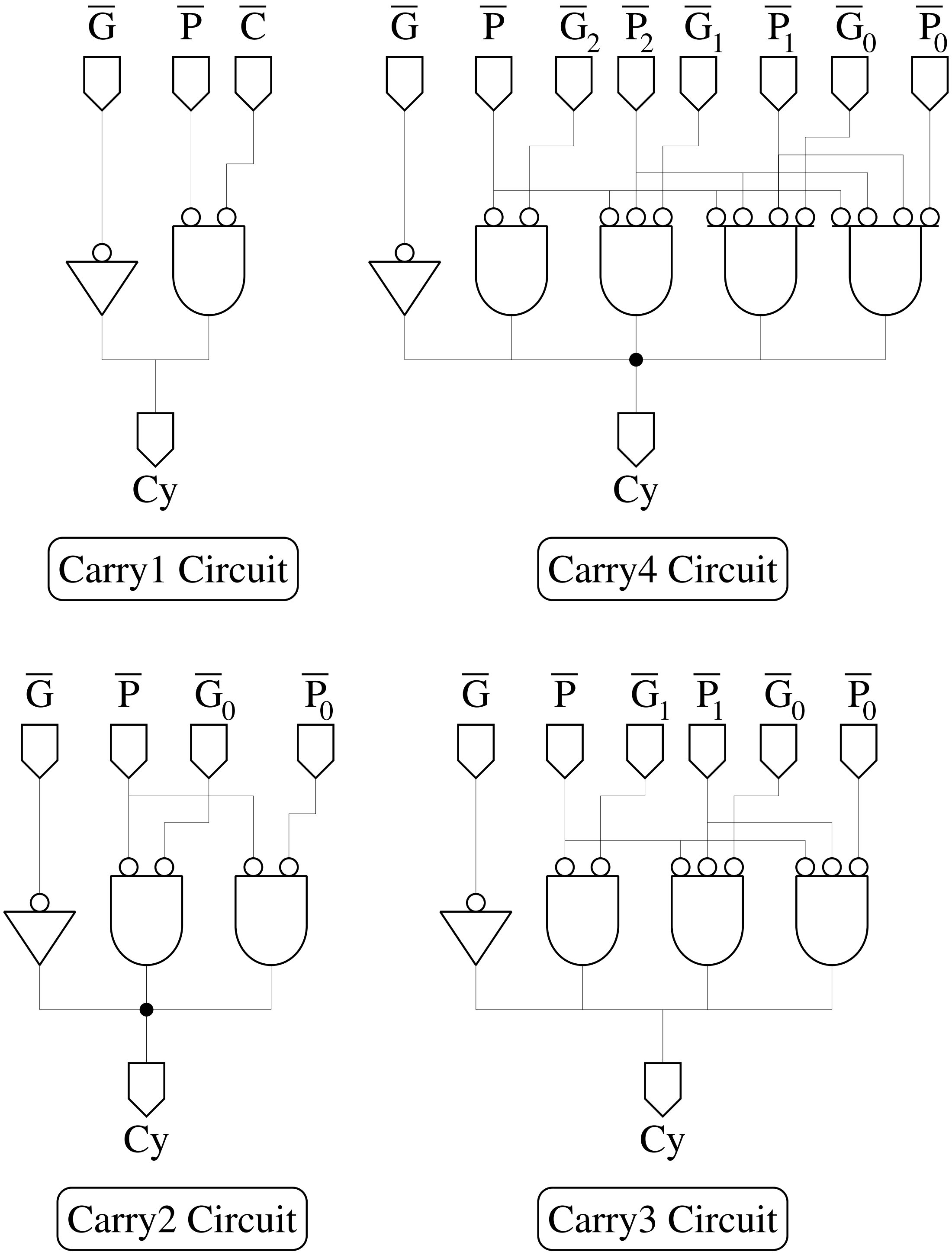


Figure 3.6: Stage 2 carry circuits.

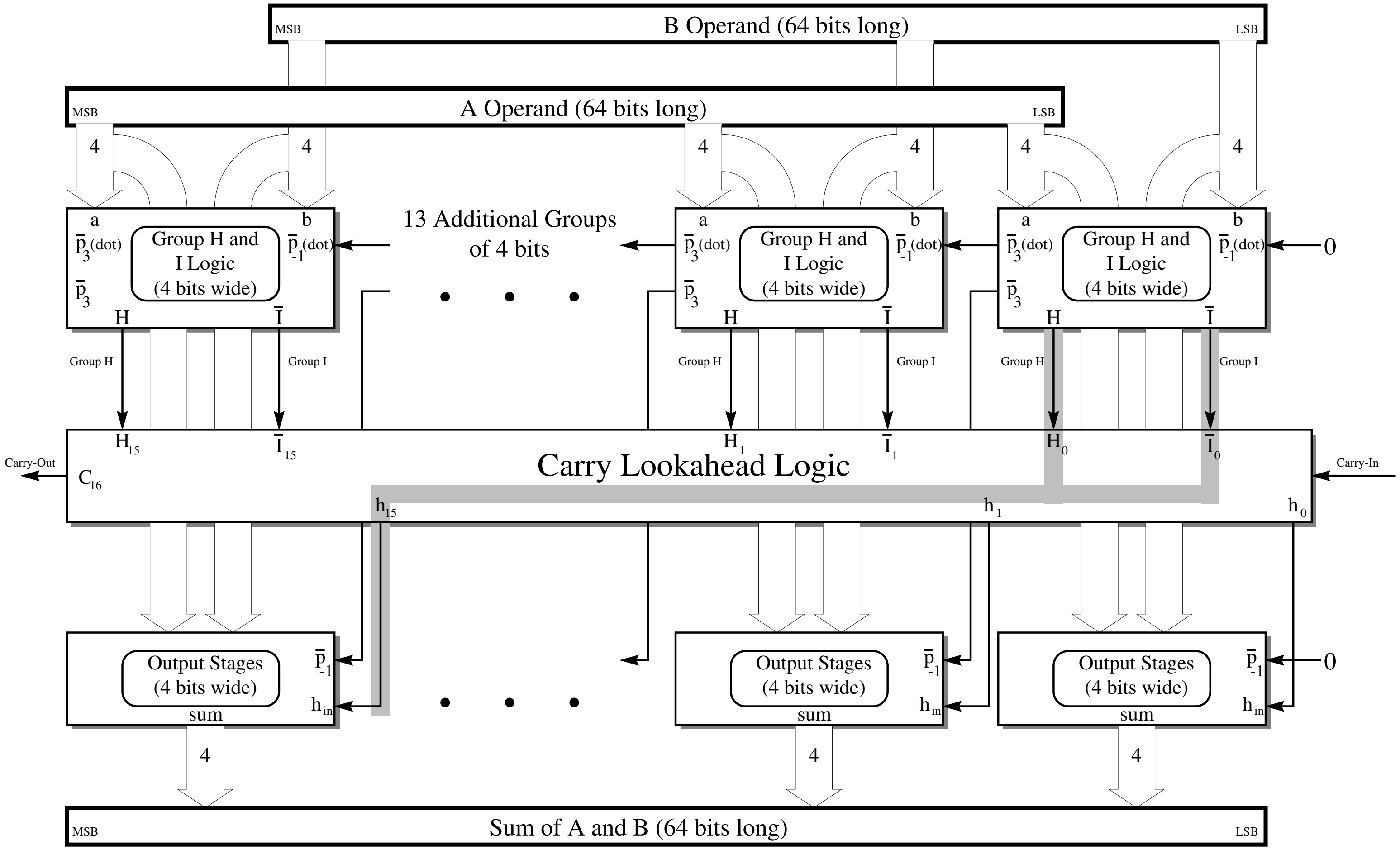


Figure 3.7: Ling adder overview.

the group and group lookahead logic. The major difference between the Ling scheme and the conventional CLA is that the group H signal (which replaces the group G signal from the CLA) is available one stage earlier than the corresponding G signal. Also the group propagate signal (P) is replaced with a signal that performs an equivalent function in the Ling method (I).

3.3.1 Group Logic

To understand the operation of the Ling adder, consider the equation for the group G signal in the conventional 4 bit CLA group (Figure 3.2).

$$\begin{aligned} G &= g_0^3 \\ &= g_3 + p_3\, g_2 + p_3\, p_2\, g_1 + p_3\, p_2\, p_1\, g_0 \end{aligned} \tag{3.14}$$

Now consider g_3. From equation 3.4 :

$$\begin{aligned} g_3 &= a_3\, b_3 \\ &= (a_3\, b_3)\,(a_3 + b_3) \\ &= {p^+}_3\, g_3 \end{aligned} \tag{3.15}$$

It is important to note, that the equation above is true only if p_3 is formed as the inclusive-or of a_3 and b_3. The exclusive-or form of p_3 *will not work!* At this point it is assumed that p_3 is produced from equation 3.5. That is :

$$\begin{aligned} p_3 &= {p^+}_3 \qquad (3.16) \\ &= a_3 + b_3 \end{aligned}$$

Now substituting equation 3.15 into equation 3.14 gives :

$$\begin{aligned} G &= {p^+}_3\, g_3 + {p^+}_3\, g_2 + {p^+}_3\, p_2\, g_1 + {p^+}_3\, p_2\, p_1\, g_0 \\ &= {p^+}_3\, (g_3 + g_2 + p_2\, g_1 + p_2\, p_1\, g_0) \\ &= {p^+}_3\, H \end{aligned}$$

which provides the definition for a new type of group signal, the *Ling group pseudo carry generate*. This leads to the general definition for the function h, when computed across a series of bits :

$$g_k^j = p^+{}_j \, h_k^j \tag{3.17}$$

Or equivalently :

$$h_k^j = g_j + g_k^{j-1} \tag{3.18}$$

Again referring back to Figure 3.2, G is produced by two stages of logic. The first stage computes the bit g_k and p_k, and the second stage computes G from the bit g_k and p_k. The Ling pseudo-generate, H, can be produced in a single stage plus a wire-OR. To see this, expand H directly in terms of the a_k and b_k inputs, instead of the intermediate g_k and p_k :

$$\begin{aligned} H &= a_3\, b_3 + a_2\, b_2 + a_1\, a_2\, b_1 + a_1\, b_1\, b_2 \\ &\quad + a_0\, a_1\, a_2\, b_0 + a_0\, a_1\, b_0\, b_2 + a_0\, a_2\, b_0\, b_1 + a_0\, b_0\, b_1\, b_2 \end{aligned} \tag{3.19}$$

If negative logic inputs are assumed, then the function H can be computed in a single INVERT-AND-OR stage. In principle, G can also be realized in a single INVERT-AND-OR stage, but it will require gates with up to 5 inputs, and 15 outputs must be connected together in a large wire-OR. Figure 3.8 shows a sample Ling 4 bit group.

3.3.2 Lookahead Logic

Consider the defining equation for h across 16 bits (from equation 3.18) :

$$\begin{aligned} h_0^{15} &= g_{15} + g_0^{14} \\ &= g_{15} + g_{12}^{14} + p_{12}^{14}\, g_8^{11} + p_{12}^{14}\, p_8^{11}\, g_4^7 + p_{12}^{14}\, p_8^{11}\, p_4^7\, g_0^3 \\ &= g_{15} + g_{12}^{14} + p_{12}^{14}\left(g_{11} + p_{11}\, g_8^{10}\right) + p_{12}^{14}\, p_8^{11}\left(g_7 + p_7\, g_4^6\right) \\ &\quad + p_{12}^{14}\, p_8^{11}\, p_4^7\left(g_3 + p_3\, g_0^2\right) \end{aligned}$$

Assume that each of p_{11}, p_7, and p_3 are produced as $p^+{}_{11}$, $p^+{}_7$, and $p^+{}_3$. Then :

$$h_0^{15} = g_{15} + g_{12}^{14} + p_{12}^{14}\left(p^+{}_{11}\, g_{11} + p^+{}_{11}\, g_8^{10}\right) + p_{12}^{14}\, p_8^{11}\left(p^+{}_7\, g_7 + p^+{}_7\, g_4^6\right)$$

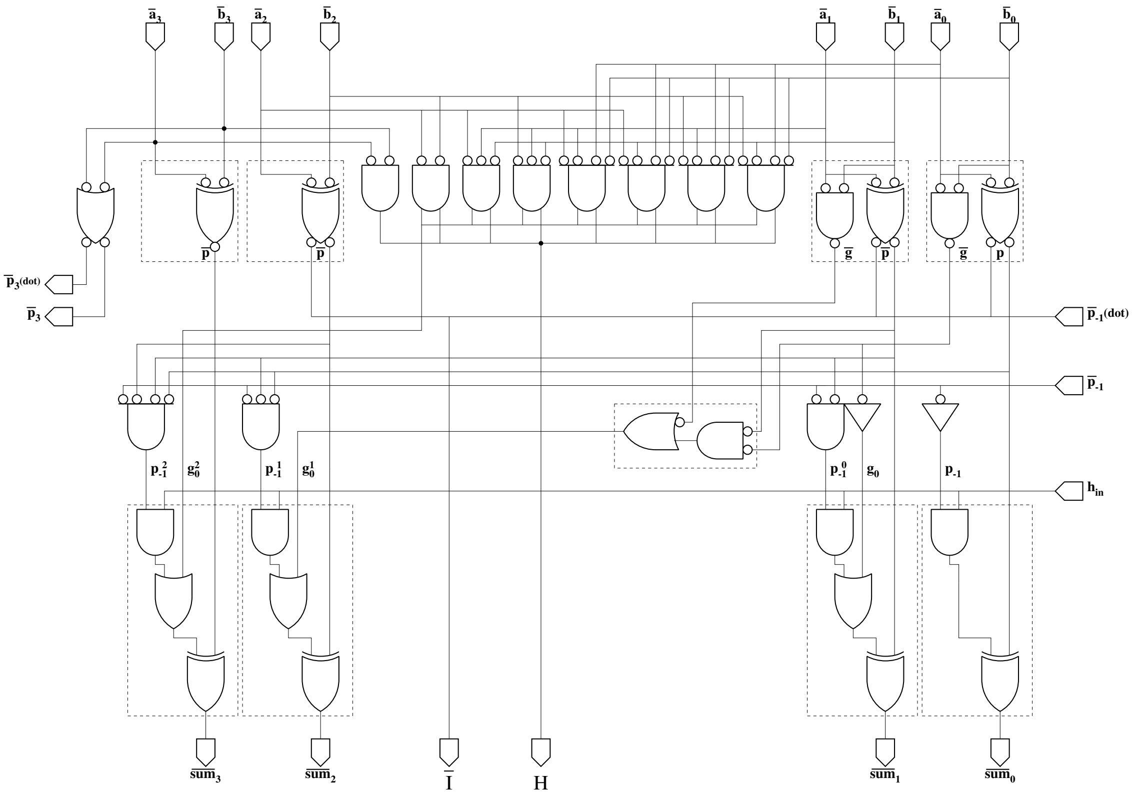


Figure 3.8: 4 bit Ling adder section.

$$
\begin{aligned}
& + p_{12}^{14}\, p_8^{11}\, p_4^{7} \left(p^{+}{}_{3}\, g_3 + p^{+}{}_{3}\, g_0^{2}\right) \\
= \; & g_{15} + g_{12}^{14} + p_{11}^{14} \left(g_{11} + g_8^{10}\right) + p_{12}^{14}\, p_7^{11} \left(g_7 + g_4^{6}\right) + p_{12}^{14}\, p_8^{11}\, p_3^{7} \left(g_3 + g_0^{2}\right) \\
= \; & g_{15} + g_{12}^{14} + p_{11}^{14} \left(g_{11} + g_8^{10}\right) + p_{11}^{14}\, p_7^{10} \left(g_7 + g_4^{6}\right) + p_{11}^{14}\, p_7^{10}\, p_3^{6} \left(g_3 + g_0^{2}\right) \\
= \; & h_{12}^{15} + p_{11}^{14}\, h_8^{11} + p_{11}^{14}\, p_7^{10}\, h_3^{7} + p_{11}^{14}\, p_7^{10}\, p_3^{6}\, h_0^{3} \\
= \; & h_{12}^{15} + i_{12}^{15}\, h_8^{11} + i_{12}^{15}\, i_8^{11}\, h_3^{7} + i_{12}^{15}\, i_8^{11}\, i_4^{7}\, h_0^{3}
\end{aligned}
$$

where i is a new function defined as :

$$i_k^{i} = p_{i-1}\, p_{i-2} \ldots p_k\, p_{k-1} \tag{3.20}$$

Note that the indexes on the p terms are slightly different than that of the i term. Using this definition of i, the formation of h across multiple groups from the group H and I signals is exactly the same as the formation of g across multiple groups from the group G and P signals. Thus, exactly the same group and supergroup lookahead logic can be used for the Ling adders, as was used in the CLA. Detail for the Ling lookahead logic is shown in Figure 3.9. The only real difference is that G and P are replaced by I and H, which for a four bit group are :

$$
\begin{aligned}
H &= h_0^{3} \\
&= g_3 + g_2 + p_2\, g_1 + p_2\, p_1\, g_0 \\
I &= i_0^{3} \\
&= p_2\, p_1\, p_0\, p^{+}{}_{-1}
\end{aligned}
$$

Note that the formation of I requires the p^{+} from the most significant bit position of the adjacent group.

One minor nuisance with this implementation of the Ling adder, is that the complement of H is a difficult function to implement. As a result, only a positive logic version of H is available for use by the first level of the lookahead logic. The fastest realization of the group I signal is only available in a negative logic form. The first layer of lookahead circuits (Figure 3.10) must be modified to accept a positive logic H and a negative logic I. This requires a strange type of NOR gate which has a single inverting input, and from

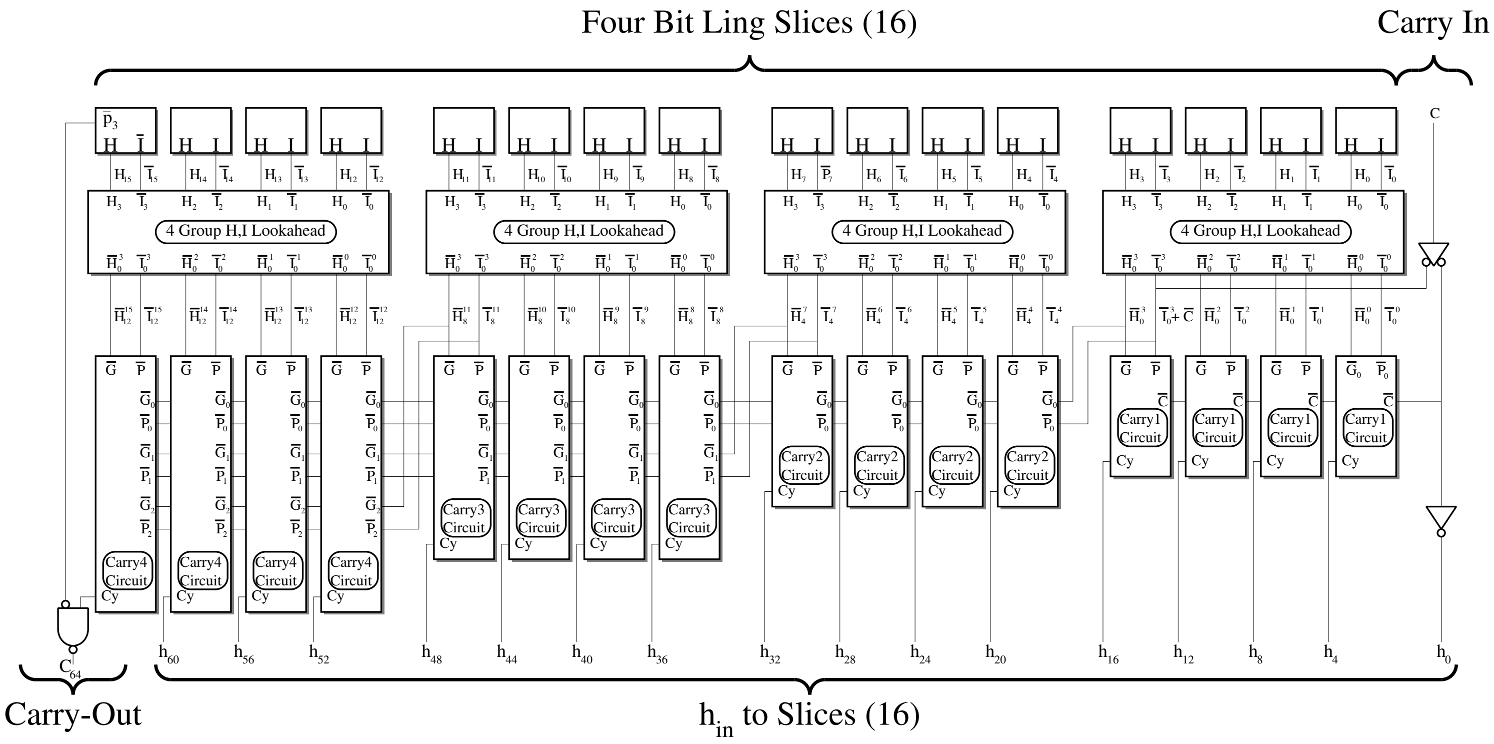


Figure 3.9: Group H and I connections for Ling adder.

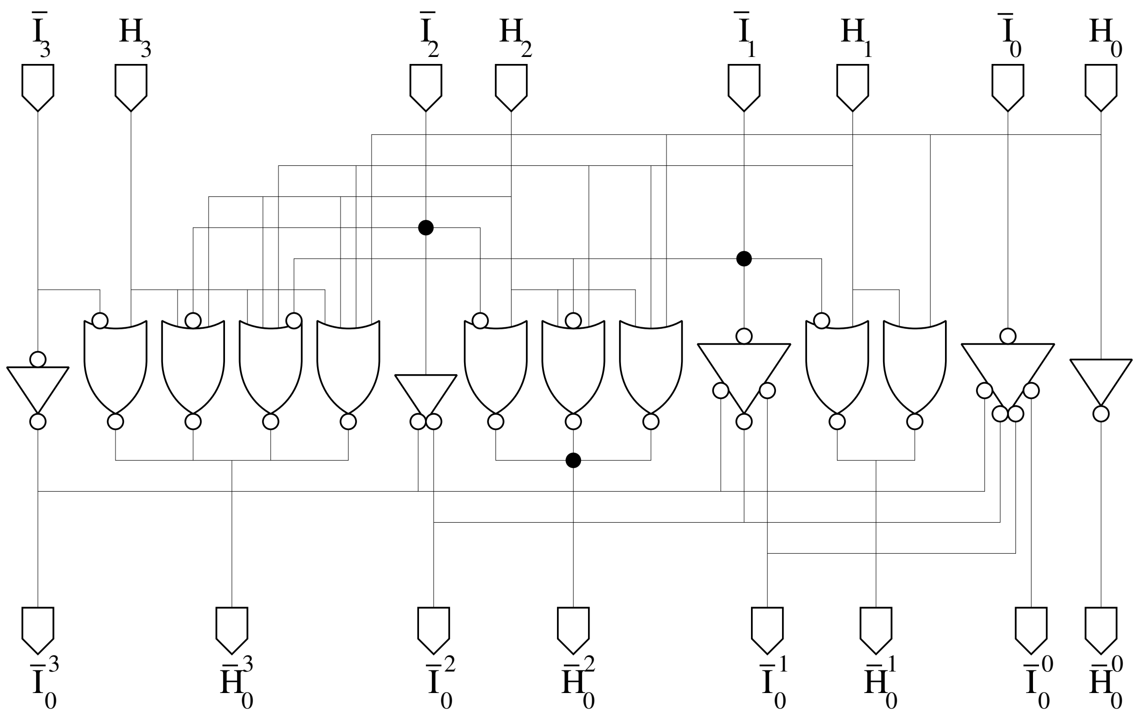


Figure 3.10: H and I circuits.

1 to 3 non-inverting inputs. The circuit for such a strange looking NOR gate is shown in Figure 3.11.

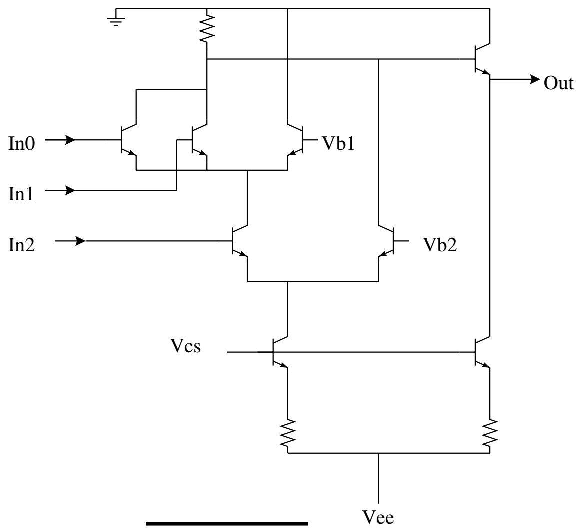
3.3.3 Producing the Final Sum

The lookahead logic returns the signal h_{in}, which is not a true carry, to each of the groups. For example, the signal supplied to the high order group (h_{60} from Figure 3.9) has produced the following signal :

$$h_{60} = h_0^{59} + i_0^{59}\, c_{in}$$

Computation of the final sum requires the carry (c_{60}), which can be recovered from h_{60} by using equations 3.17 and 3.20:

$$\begin{aligned} c_{60} &= g_0^{59} + p_0^{59}\, c_{in} \\ &= {p^+}_{59}\, h_0^{59} + {p^+}_{59}\, {p^+}_{-1}^{58}\, c_{in} \end{aligned}$$



$$\text{Out} = \overline{\overline{\text{In2}} + \text{In1} + \text{In0}}$$

Figure 3.11: NOR gate with 1 inverting input and 2 non-inverting inputs.

$$
\begin{aligned}
&= p^{+}{}_{59} \left[h_0^{59} + i_0^{59}\, c_{in} \right] \\
&= p^{+}{}_{59}\, h_{60}
\end{aligned}
$$

This result can be used in place of c_{in} in equation 3.13 to modify the logic in the output stage to produce the proper sum [3] [34].

3.3.4 Remarks on Ling Example

This Ling adder example builds upon the CLA example presented previously. The Ling scheme is potentially faster than the CLA design because the critical path consists of 3 NOR stages and a single EXCLUSIVE-OR stage vs 4 NOR stages and an EXCLUSIVE-OR for the CLA. Since the wire lengths and gate counts of the two are very close, this results in a faster adder.

3.4 Multiple Generation for Multipliers

Chapter 2 described various partial product recoding algorithms, and in particular the Booth series of multiplication algorithms. The Booth 3 multiplication algorithm can provide a significant reduction in the hardware requirements over conventionally implemented algorithms, but requires the production of 3 times the multiplicand (3M). A general purpose adder can be used to perform this computation, by adding the multiplicand to 2 times the multiplicand. An adder that is designed specifically for computing this times 3 multiple will result in a significant reduction in the hardware. An example is given in the first half of this section.

The partially redundant Booth 3 algorithm described in the previous chapter provides the hardware reduction of the general Booth 3 algorithm, along with removal of a carry propagate add from the critical path. The performance depends on the the fast computation of short length multiples (say $\leq$ 14 bits or so). The second half of this section shows how these short length multiples can be efficiently and quickly computed.

3.4.1 Multiply by 3

The general idea is to replace the Ling 4 bit group (Figure 3.8), with a 7 bit group which is specifically optimized for computing 3 times the input operand. The carry lookahead network remains the same. Because a group now consists of 7 bits, instead of 4 bits, the lookahead network is smaller, and could (depending on the length required) be fewer stages.

For this discussion, the assumption is that the B operand has been replaced by a shifted copy of the A operand :

$$\begin{aligned} B &= \sum_{k=0}^{n-1} a_k \cdot 2^{k+1} \\ &= \sum_{k=1}^{n} a_{k-1} \cdot 2^k \end{aligned}$$

This gives the following result for g_k and p_k :

$$g_k = a_k\, a_{k-1} \tag{3.21}$$

$$p_k = a_k + a_{k-1} \tag{3.22}$$

Substituting this into the equation for the group G (equation 3.10)gives :

$$g_0^3 = a_3\, a_2 + a_2\, a_1 + a_3\, a_1\, a_0 + a_2\, a_0\, a_{-1}$$

This is much simpler than even the Ling expansion (equation 3.19). Sticking with the limit of gates with no more than 4 inputs, it is possible to compute h_0^6 in a single stage:

$$h_0^6 = a_6\, a_5 + a_5\, a_4 + a_4\, a_3 + a_5\, a_3\, a_2 + a_4\, a_2\, a_1 + a_5\, a_3\, a_1\, a_0 + a_4\, a_2\, a_0\, a_{-1}$$

A sample 7 bit times 3 group is shown in Figure 3.12. This section can be interchanged with the four bit Ling group (Figure 3.8), with the carry lookahead logic remaining unchanged.

Internal carries required for the final sum generation (as per equation 3.13) are produced directly from the primary inputs according to the following :

$$\begin{aligned}
g_0^0 &= a_0\, a_{-1} \\
g_0^1 &= a_1\, a_0 + a_0\, a_{-1} \\
g_0^2 &= a_2\, a_1 + a_1\, a_0 + a_2\, a_0\, a_{-1} \\
g_0^3 &= a_3\, a_2 + a_2\, a_1 + a_3\, a_1\, a_0 + a_2\, a_0\, a_{-1} \\
g_0^4 &= a_4\, a_3 + a_3\, a_2 + a_4\, a_2\, a_1 + a_3\, a_1\, a_0 + a_4\, a_2\, a_0\, a_{-1} \\
g_0^5 &= a_5\, a_4 + a_4\, a_3 + a_5\, a_3\, a_2 + a_4\, a_2\, a_1 + a_5\, a_3\, a_1\, a_0 + a_4\, a_2\, a_0\, a_{-1}
\end{aligned}$$

Note the significant sharing possible between adjacent g terms, which is taken advantage of in the implementation.

3.4.2 Short Multiples for Multipliers

A minor change to Figure 3.12 allows production of the biased short length multiple required by the redundant Booth 3 multiplication algorithm from Chapter 2. However, this modification still leaves a latency of 3 stages for this multiple. As this multiple is likely to be on the critical path through the multiplier, one stage can be eliminated by modifying the output stages to merge the gates labeled 1 in the figure into the output stages (gates labeled 2). A sample circuit which performs this merge is shown in Figure 3.13. The length of the short multiple can be approximately doubled by connecting two short multiple generators together as outlined in Figures 3.14, 3.15, and 3.16. Figure 3.14 shows a 13 bit section of a redundant times 3 multiple, in the format shown in Figure 2.18 of Chapter 2. Note that the scheme shown here is only good for negative logic inputs and outputs. Positive logic inputs and outputs are slightly different. The figures show a 13 bit short multiple, which is the limit for this scheme if 4 input gates are used.

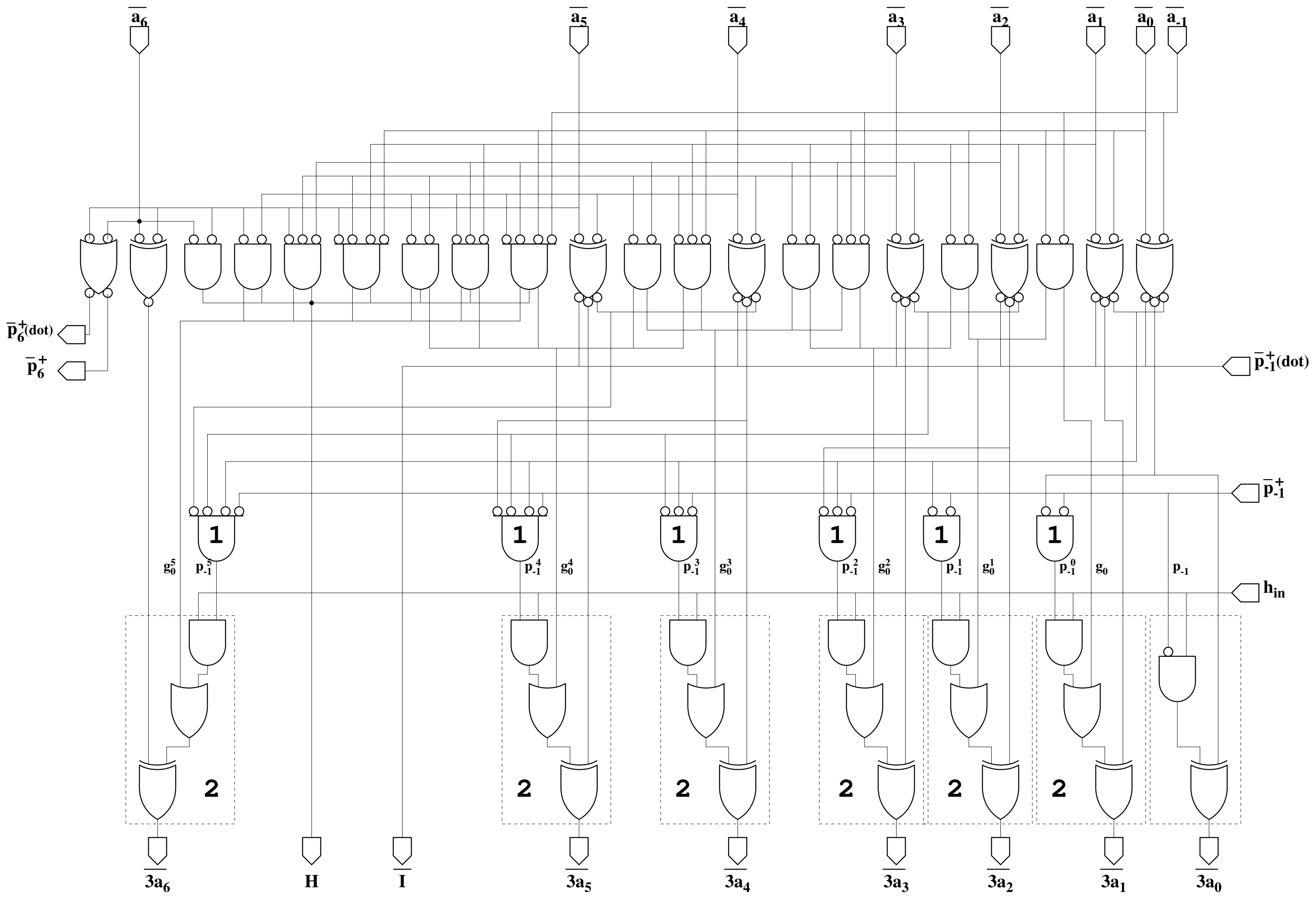


Figure 3.12: Times 3 multiple generator, 7 bit group.

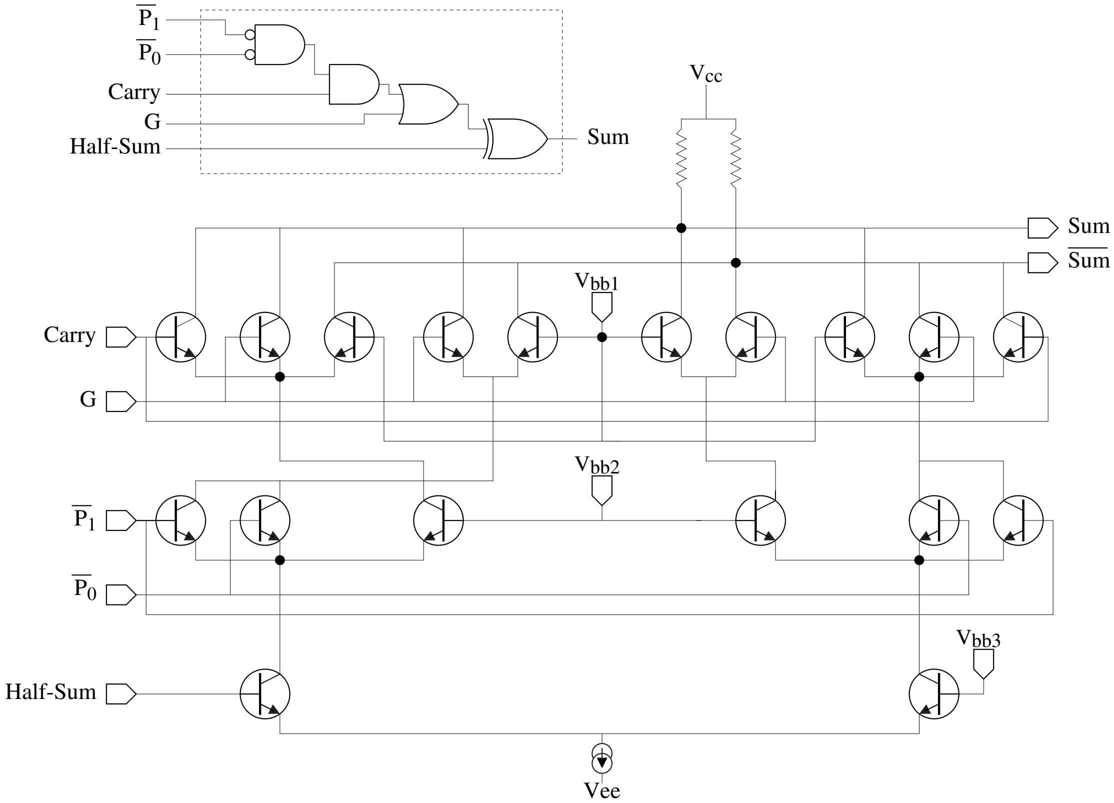


Figure 3.13: Output stage.

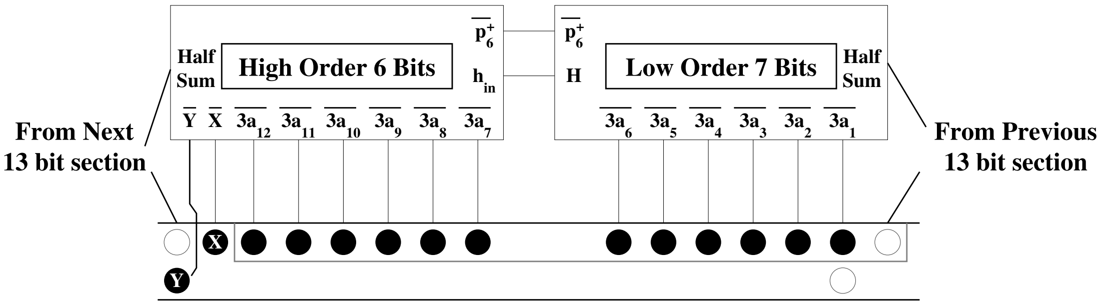


Figure 3.14: 13 bit section of redundant times 3 multiple.

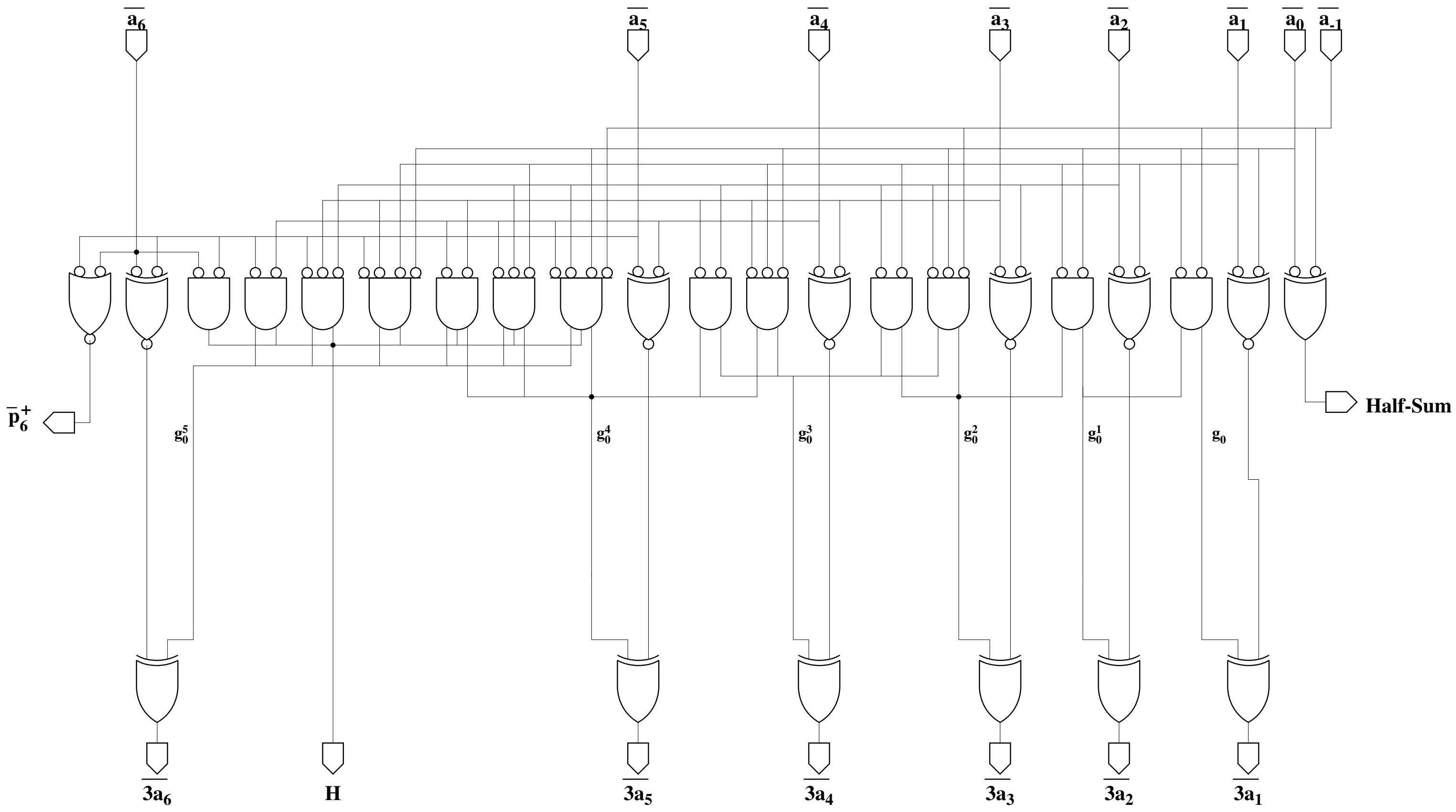


Figure 3.15: Short multiple generator - low order 7 bits.

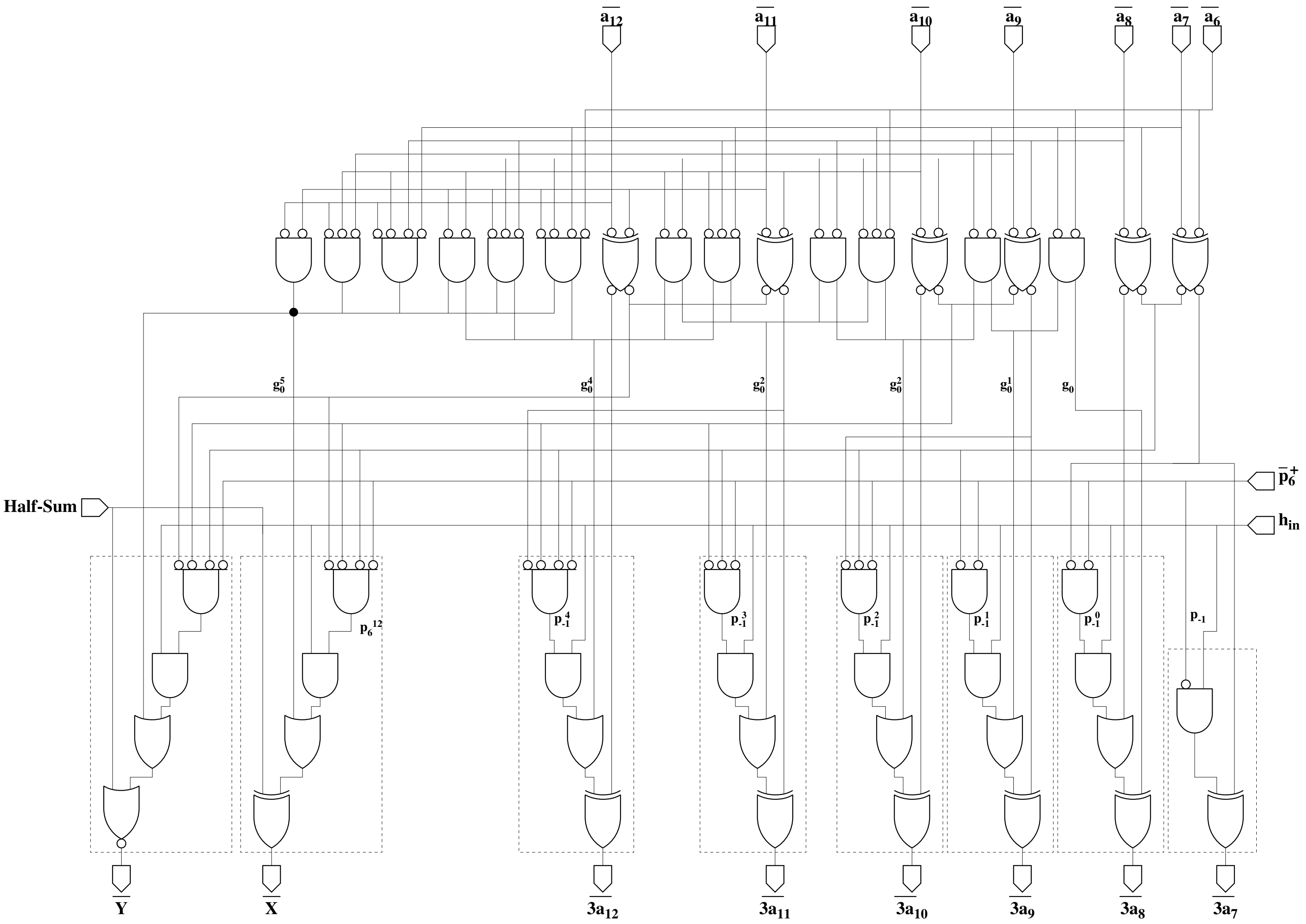


Figure 3.16: Short multiple generator - high order 6 bits.

3.4.3 Remarks on Multiple Generation

Efficient methods for producing 3 times an operand, both full length and short lengths, have been presented above. Other useful multiples to generate would be 5 times, and 7 times an operand, but there appears to be no better scheme than just using a conventional carry propagate adder.

3.5 Summary

As will be shown in Chapter 5, carry propagate adders play a crucial role in the overall performance of high speed multipliers. This chapter has described a number of efficient and high performance adder designs which will be used in the multiplier evaluations in the following chapters. Although the designs have been specifically tuned for ECL based adders, the ideas can be applied to other technologies.

Specifically, this chapter has presented an adder design that uses the Ling lookahead method. This adder has one less stage of logic along the critical path than an adder using the traditional carry lookahead method. Since the complexity and wire lengths are comparable, this leads to a faster adder.

Significant hardware reductions (about a 20% reduction in gate count) can result by designing a specialized adder to compute 3M. Because the basic group size can be made longer the performance may also improve, since fewer stages are required for the carry propagation network.

By carefully optimizing the circuits, an efficient and fast (2 stages of logic) short multiple generator can also be designed. The speed and efficiency of this block is crucial to the performance of the redundant Booth 3 multiplication algorithm described in Chapter 2.

Chapter 4

Implementing Multipliers

Chapter 2 described various methods of generating partial products, which then must be added together to form a final product. Unfortunately, the fastest method of summing the partial products, a Wallace tree or some other related scheme, requires very complex wiring. The lengths of these wires can affect the performance, and the wires themselves take up valuable layout area. Manually wiring a multiplier tree is a laborious process, which makes it difficult to accurately evaluate different multiplier organizations. To make it possible to efficiently design many different kinds of multipliers, an automated multiplier generator that designs the layout of partial product generators and summation networks for multipliers is described in this chapter. Since the partial product generator and summation network constitute the bulk of the differences between various multiplication algorithms, many implementations can be evaluated, providing a systematic approach to multiplier design. The layouts produced by this tool take into consideration wire lengths and delays as a multiplier is being produced, resulting in an optimized multiplier layout.

4.1 Overview

The structure of the multiplier generator is shown in Figure 4.1. Inputs to the tool consists of various high level parameters, such as the length and number of partial products, and the algorithm to be used in developing the summation network. Separately input to the tool is technology specific information, such as metal pitches, geometric information about the

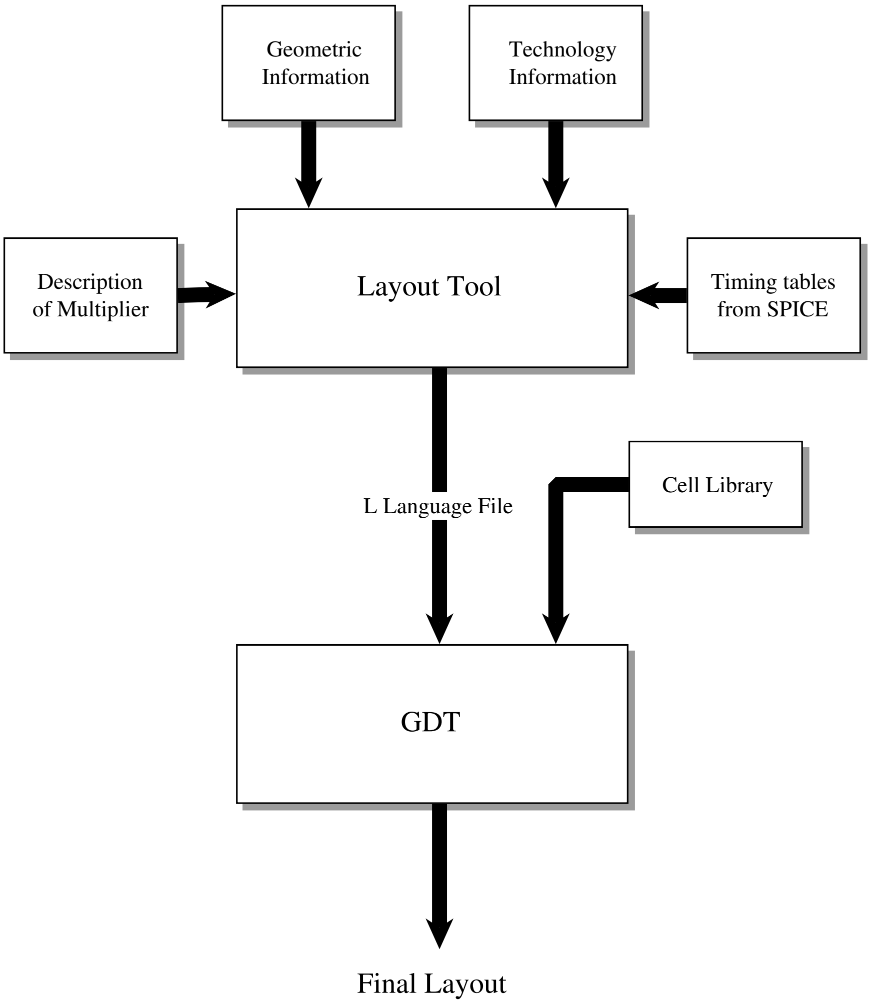


Figure 4.1: Operation of the layout tool.

primitive cells, such as the size of a CSA, I/O terminal locations, etc., and timing tables, which have been derived from HSPICE [18]. The output of the tool is an L language (a layout language) file, which contains cell placement information and a net list which specifies the cell connections. The L file is then used as input to a commercial IC design tool (GDT from Mentor Graphics). This commercial tool actually places the cells, and performs any necessary routing using a channel router. Because most things are table driven, the tool can quickly be modified to adapt to different technologies or layout tools.

4.2 Delay Model

An accurate delay model is an essential part of the multiplier generator if it is to account for the effect of wire lengths on the propagation delay while the layout is being generated. Simple models which ignore fanout, wire delays, and inputs that differ in propagation delays (like that of Winograd [39] [40]), can lead to designs which are slower and/or larger than the technology would allow.

The multiplier generator uses a delay model (Figure 4.2) based upon logic elements that are fan-in limited, but each input has a different arrival time at the main logic element (Delay1, Delay2, etc.) The main logic element has an intrinsic delay (Main Delay), and

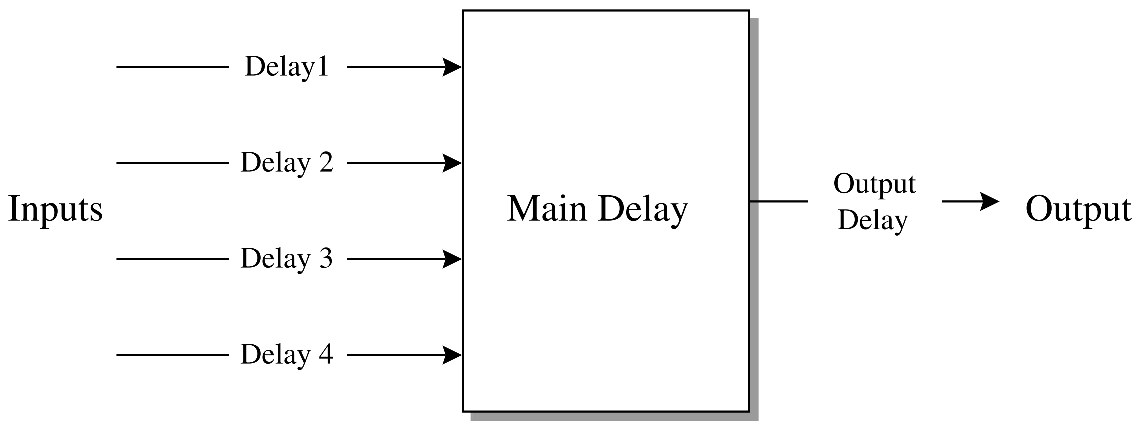


Figure 4.2: Delay model.

the output also has a fixed delay (Output Delay) [1]. Each output also has a delay which

[1] In actual use the Main Delay and the Output Delay are not really needed and in fact are set to 0.

is proportional to the length of wire being driven. A factor for the fan-out should also be included, but is not necessary for multipliers, since all of the CSAs have a fan-out of 1. The individual delays are determined by running SPICE or HSPICE, as is the proportionality constant for the wire delay.

4.3 Placement methodology

A general block diagram for a multiplication implementation is shown in Figure 4.3. A high speed parallel multiplier consists of a partial product generator, a summation network responsible for summing the partial products down to two final operands in a carry propagate free manner, and a final carry propagate adder which produces the final product.

4.3.1 Partial Product Generator

To understand how the partial products are formed, an 8x8 bit example using the simple multiplication algorithm described in Chapter 2 will be used. The partial product dot diagram for such a multiplication is shown in Figure 4.4. Each dot represents an AND gate which produces a single bit. The dots in the figure are numbered according to the particular bit of the multiplicand (M) that is attached to the input of the multiplexer. These multiplexers are then grouped into rows which share a common select line to form a single partial product. Each row of the dot diagram represents an 8 bit wide selection multiplexer, which selects from the possible inputs 0 and M. The select line on the 8 bit multiplexer is controlled by a particular bit of the multiplier (Figure 4.5). A diagonal swatch of multiplexers (Figure 4.6) consists of multiplexers that require access to the same bit of the multiplicand. Finally a vertical column of multiplexers all have outputs of the same arithmetic weight (Figure 4.7).

The layout tool uses the following methodology as it places the individual multiplexers that form each partial product (refer to Figure 4.8). The first row of multiplexers is placed from right to left corresponding to the least significant bit to the most significant bit. The select for each partial product is then run horizontally over all the multiplexers in the row. A vertical routing channel is allocated between each column of multiplexers. The multiplexers

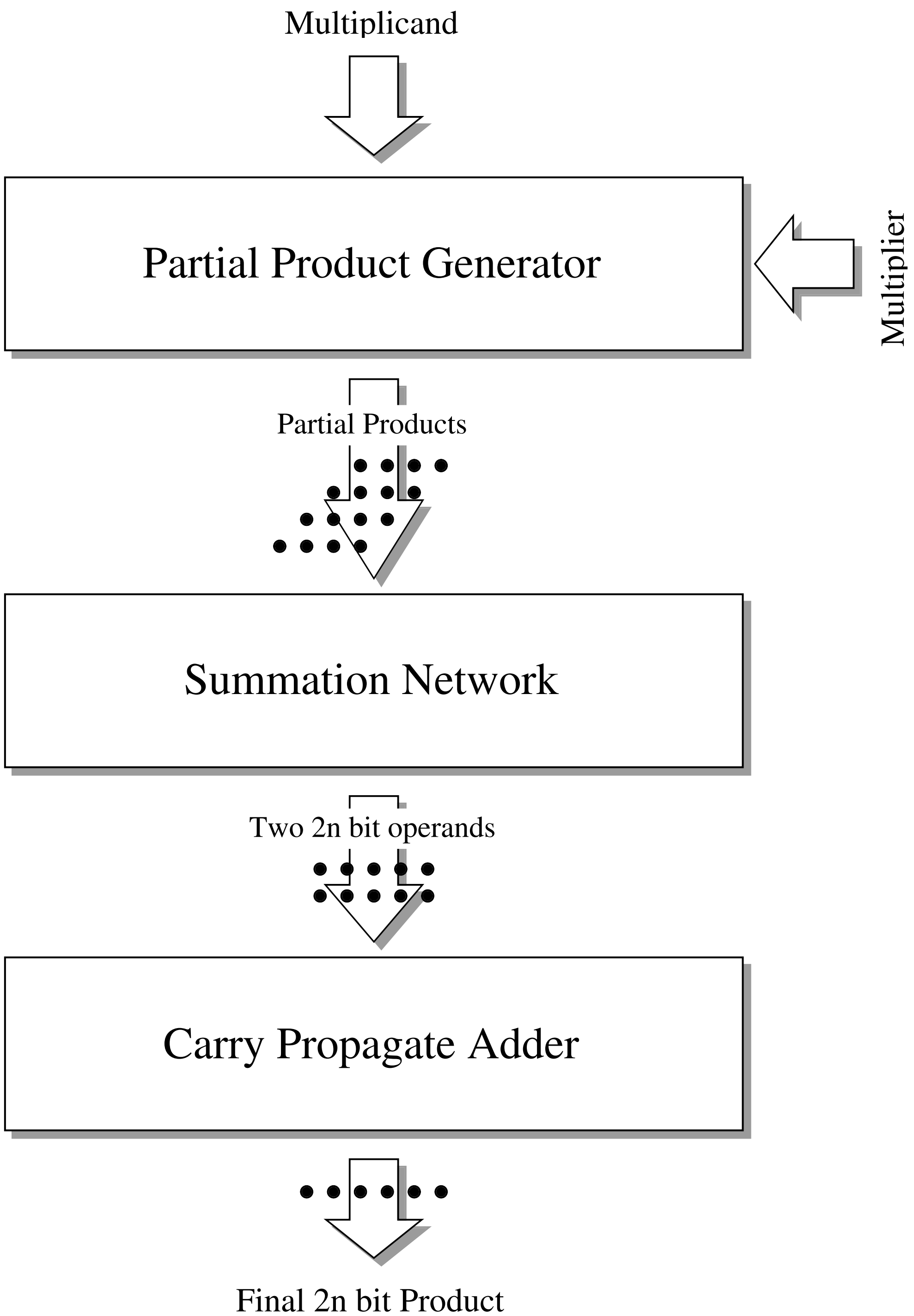


Figure 4.3: Multiplication block diagram.

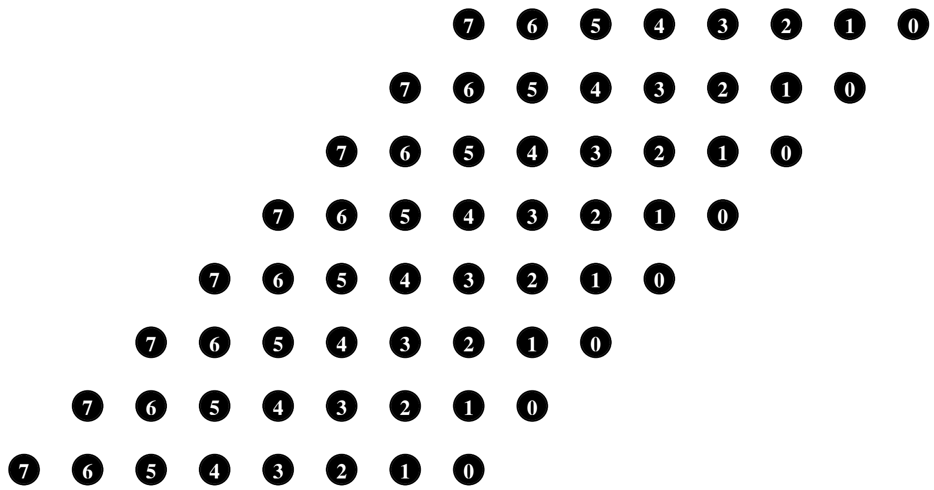


Figure 4.4: Partial products for an 8x8 multiplier.

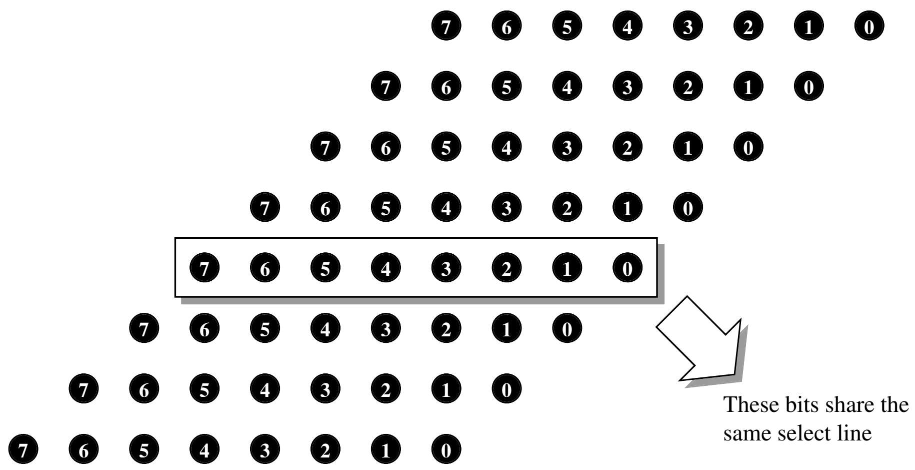


Figure 4.5: A single partial product.

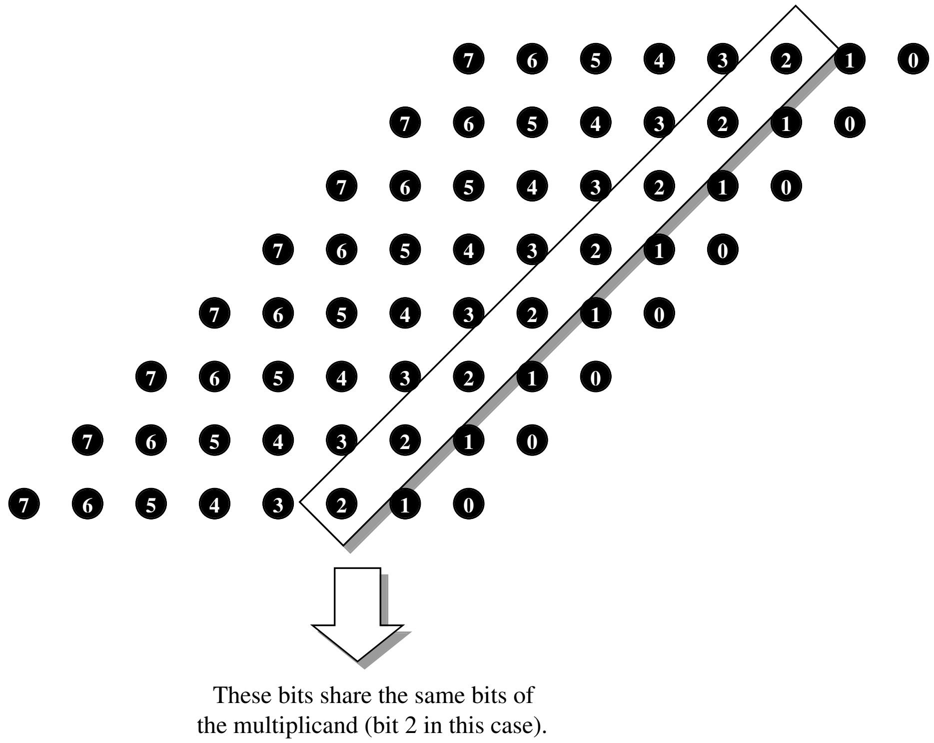


Figure 4.6: Dots that connect to bit 2 of the multiplicand.

for the second row of horizontal dots are then placed immediately underneath the first row of multiplexers, but shifted one position to the left to account for the additional arithmetic weight of the second partial product with respect to the first. Bits of the multiplicand that must connect to diagonal sections are routed in the routing channel and over the columns of cells using feedthroughs provided in the layout of the individual multiplexers. The outputs of the multiplexers are then routed to the summation network at the bottom. Note that all bits of the same arithmetic weight are routed in the same routing channel. This makes the wiring of the CSAs relatively simple.

Multiplexer Alignment

Early versions of this software tool allowed adjacent bits of a single partial product generator to be unaligned in the Y direction. For some of the multiplication algorithms, a large number of shared select wires control the multiplexers that create these bits. If these multiplexers

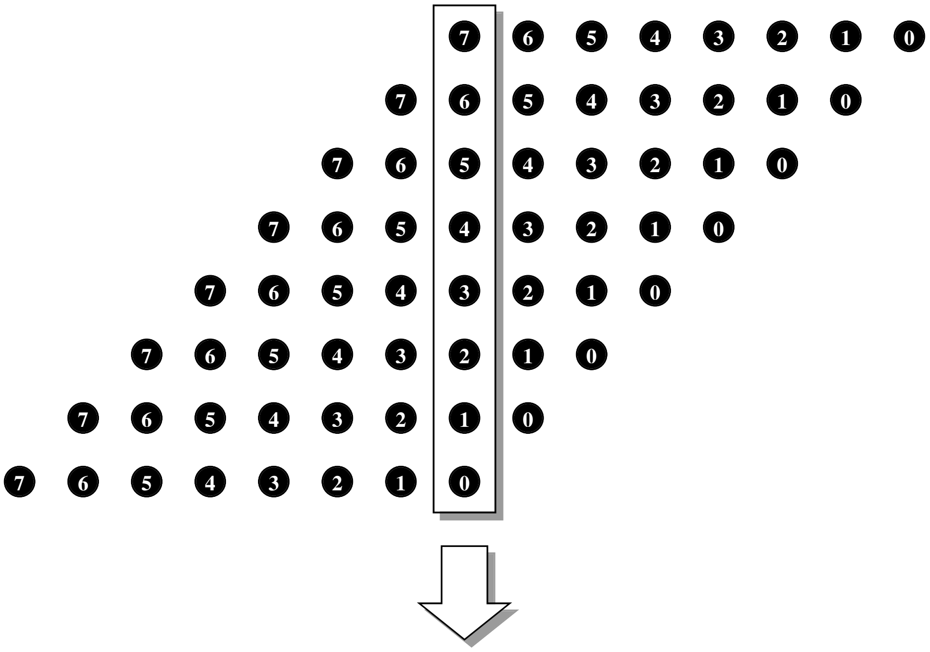


Figure 4.7: Multiplexers with the same arithmetic weight.

are aligned in the Y direction (as shown in the top of Figure 4.9, these shared wires, run horizontally in a single metal layer and occupy no vertical wiring channels. If these multiplexers are instead allowed to be misaligned (the bottom of Figure 4.9), the wires make vertical jogs in the routing channel, and an additional metal layer will be needed for the vertical sections. This could cause the channel to expand in width. For this reason, the current implementation forces all bits in a single partial product to line up in the Y direction. An improved version of the program might allow some limited amount of misalignment to remove "packing spaces". These are areas that are too small to fit anything into, created by the forced alignment of the multiplexers. The final placement of the multiplexers for the sample 8x8 multiplier is shown in Figure 4.10

An alternate approach for organizing the partial product multiplexers, that was not used, involves aligning the partial products in such a way that selects run horizontally (same as before), and bits of the multiplicand run vertically (Figure 4.11). Cell feedthroughs are still required, as a particular bit of the the multiplicand may still have to reach multiplexers that are in two adjacent columns, if the Booth 2 or higher algorithms are being realized. In

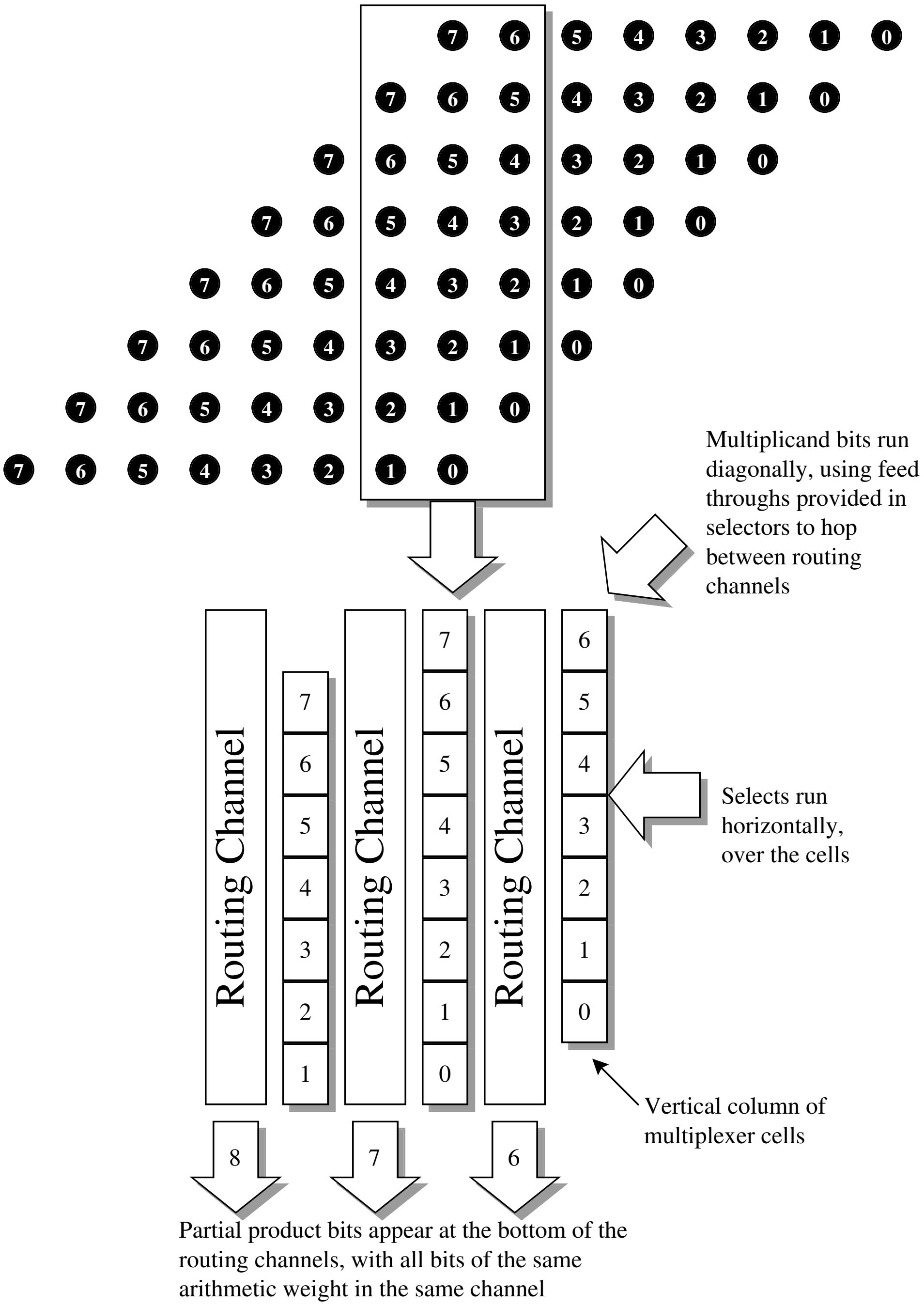


Figure 4.8: Physical placement of partial product multiplexers.

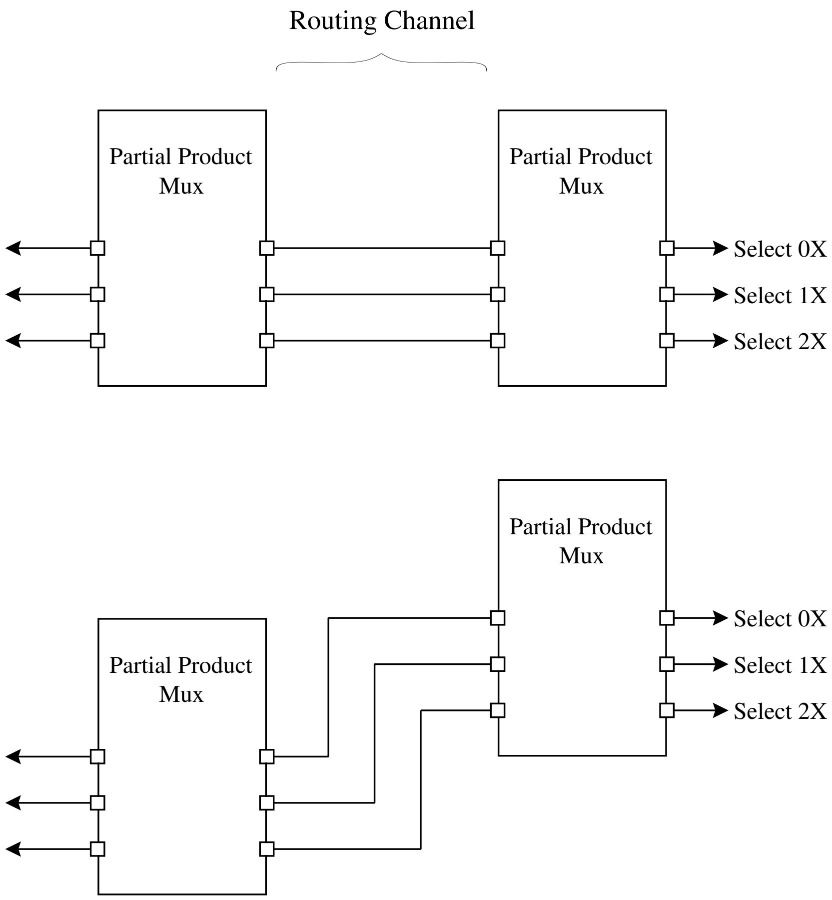


Figure 4.9: Alignment and misalignment of multiplexers.

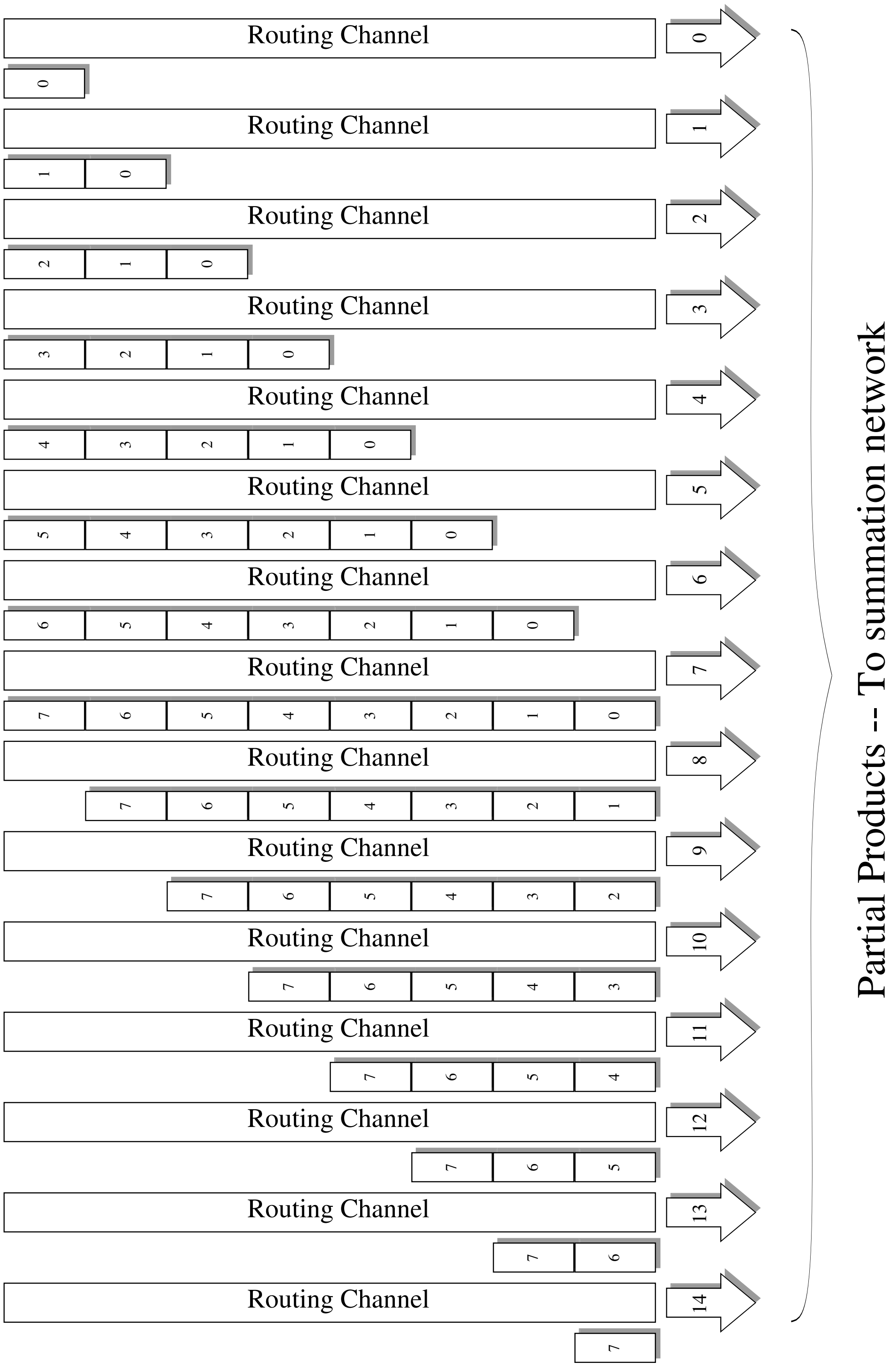


Figure 4.10: Multiplexer placement for 8x8 multiplier.

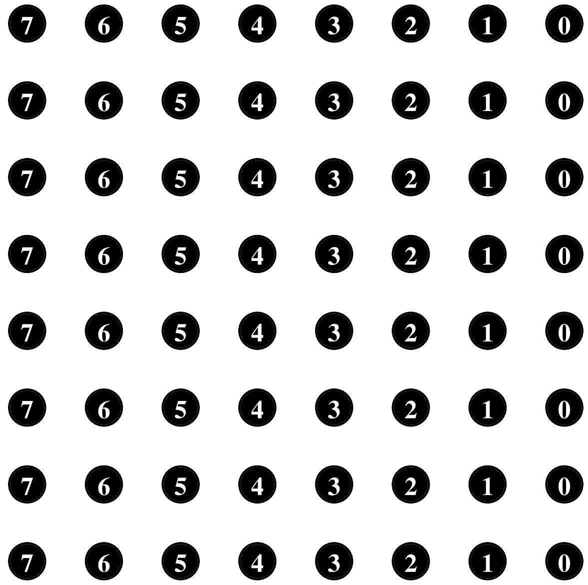


Figure 4.11: Aligned partial products.

addition, the partial product bits in any particular routing channel are of varying arithmetic weight, requiring unscrambling before being applied to the summation network. This methodology is used for linear arrays, as the unscrambling can occur in sequence with the summation of the next partial product. The unscrambling requires about as much extra wiring as routing the bit of the multiplicand diagonally through the partial product selectors, which was why this method was not used by the multiplier generator. Aligning the partial products should have comparable area and performance. Note that this method requires approximately N (N is the length of the multiplicand) routing channels, whereas the previous method required about 2N routing channels. The tree folding optimization (described below) reduces the number of routing channels actually needed in the previous method to about N. The decision was made to concentrate on the first method because there are many more partial product output wires (N^2) than there are multiple wires (N), and it will require less power to make N wires a little longer verses N^2 a little longer. Also having wires of the same arithmetic weight emerge from the same routing channel makes the placement and wiring of the CSAs in the summation network easier.

4.3.2 Placing the CSAs

The goal of the CSA placement phase of the multiplier generator is to place and wire up the CSAs, given a particular partial product multiplexer arrangement. Using the minimum amount of area and the smallest delay, the partial products are to be reduced to two numbers which can then be added to form the final product.

The multiplexer placement scheme used by the multiplier generator creates a topology illustrated in Figure 4.10. The multiplexers have been placed such that all multiplexer outputs of a given arithmetic weight border the same vertical routing channel. The task now is to place CSAs in the cell columns and wire them together in the routing channel. Since all inputs of a correctly wired CSA must have the same arithmetic weight, and all multiplexer outputs of a given arithmetic weight border the same vertical routing channel, the cell column that a CSA will be placed in is completely determined by the arithmetic weight of it's inputs. The placement of the CSAs occurs sequentially, and as each CSA is added it is placed below all other previously placed cells. Other phases after the initial placement can move CSAs around in an attempt to reduce area or delay.

The assumed geometry for a CSA is shown in Figure 4.12. The power supplies run vertically over the cell in some top level metal. The inputs are all on the right side of the cell. The sum output is also on the right hand side, but the carry output is on the left side. The placement and wiring of a CSA in a vertical column can be thought of as taking 3 wires of a given arithmetic weight out of the routing channel on the immediate right, and replacing them with a single wire of the same weight and creating a new wire of weight+1 which is placed in the routing channel to the immediate left.

At any point during the placement of the CSAs, there are a number of multiplexer outputs or previously placed CSA outputs that have not been connected to any input. The next CSA to placed must be connected to 3 of these unwired outputs. The 3 wires are chosen using the following heuristic :

- A virtual wire is attached to each unwired CSA or multiplexer output. This wire extends to the bottom of the placement area. This virtual wire is added because even if an output is never wired to a CSA, it must eventually connect to the carry propagate adder placed at the bottom. By placing a virtual wire it makes outputs that are already

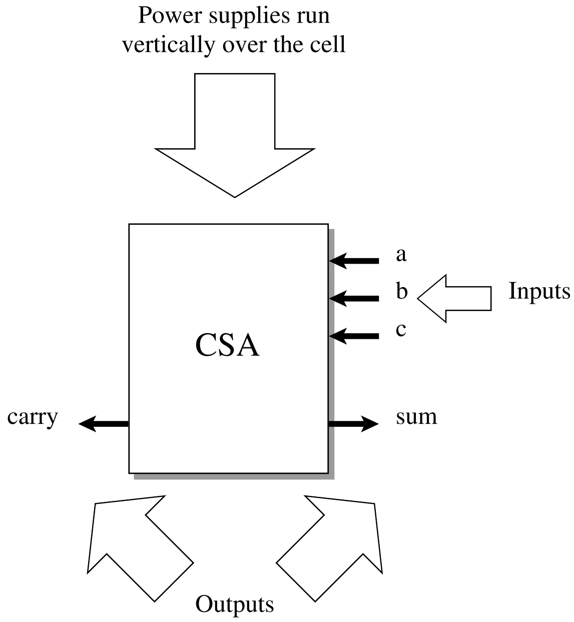


Figure 4.12: Geometry for a CSA.

near the bottom more likely to be connected to a CSA input, and outputs that are near the top (and require a long wire to reach the bottom) less likely to be connected to a CSA input. As a result, faster outputs (near the bottom) will go through more levels of CSAs and slow outputs (due to long wires to reach the bottom) will go through fewer levels of CSAs, improving overall performance.

- The propagation delay from the multiplicand or multiplier select inputs to each of the unwired outputs is computed, using the delay model described earlier. Individual bits of multiples of the multiplicand or the multiplier select signals are assumed to be valid at a time determine by a lookup table. This lookup table is determined by iterative runs of the multiplier generator, which can then allow for wire delays and possible differences in delays for individual bits of a single partial product.

- The output having the *fastest* propagation delay is chosen as the primary candidate to be attached to a CSA. A search is then made for two other unwired outputs of the same arithmetic weight. If two other unwired outputs of the same arithmetic weight cannot be found, then this output is skipped and the next fastest output is chosen, etc., until a group of at least 3 wires of the same arithmetic weight are found. If no group can be found, then this stage of the placement and wiring is finished, and the algorithm terminates.

- A new CSA is placed in the column determined by the arithmetic weight of the group. The primary candidate is wired to the input of the new CSA which has the longest input delay. Of the remaining unwired outputs with the same arithmetic weight as the primary candidate, the two *slowest* possible outputs are chosen which do not cause an increase in the output time of the CSA. These outputs are then wired to the other two inputs of the CSA.

In effect, this is a greedy algorithm, in that it is constantly choosing to add a CSA delay along the fastest path available. There are other procedures that will be described below that help the algorithm avoid local minimums as it places and wires the CSAs

This algorithm can run into problems, illustrated by the following example. Refer to the top of Figure 4.13. The left section shows a collection of dots which represent unwired outputs. The arithmetic weight of the outputs increases from right to left, with dots that are vertically aligned being of the same arithmetic weight. The above algorithm will find the 3 outputs in the little box and wire them to a CSA. This will give an output configuration as shown in the center section. The algorithm will repeat, giving the right section. This sequence of CSAs will be wired in series – essentially they will be wired as a ripple carry adder. This is too slow for a high performance implementation. The solution is to use half adders (HA) to break the ripple carry. As shown in the bottom of Figure 4.13, the first step uses a CSA, but also a group of half adders to reduce the unwired outputs to the final desired form in one step.

Placement of half adders

When and where to place half adders is based upon a heuristic, which comes from the following observations. These observations are true in the case where the propagation delay from any input of a CSA to any output are equal and identical to the the propagation delay from any input of a HA to any output. Also, all delays must be independent of any

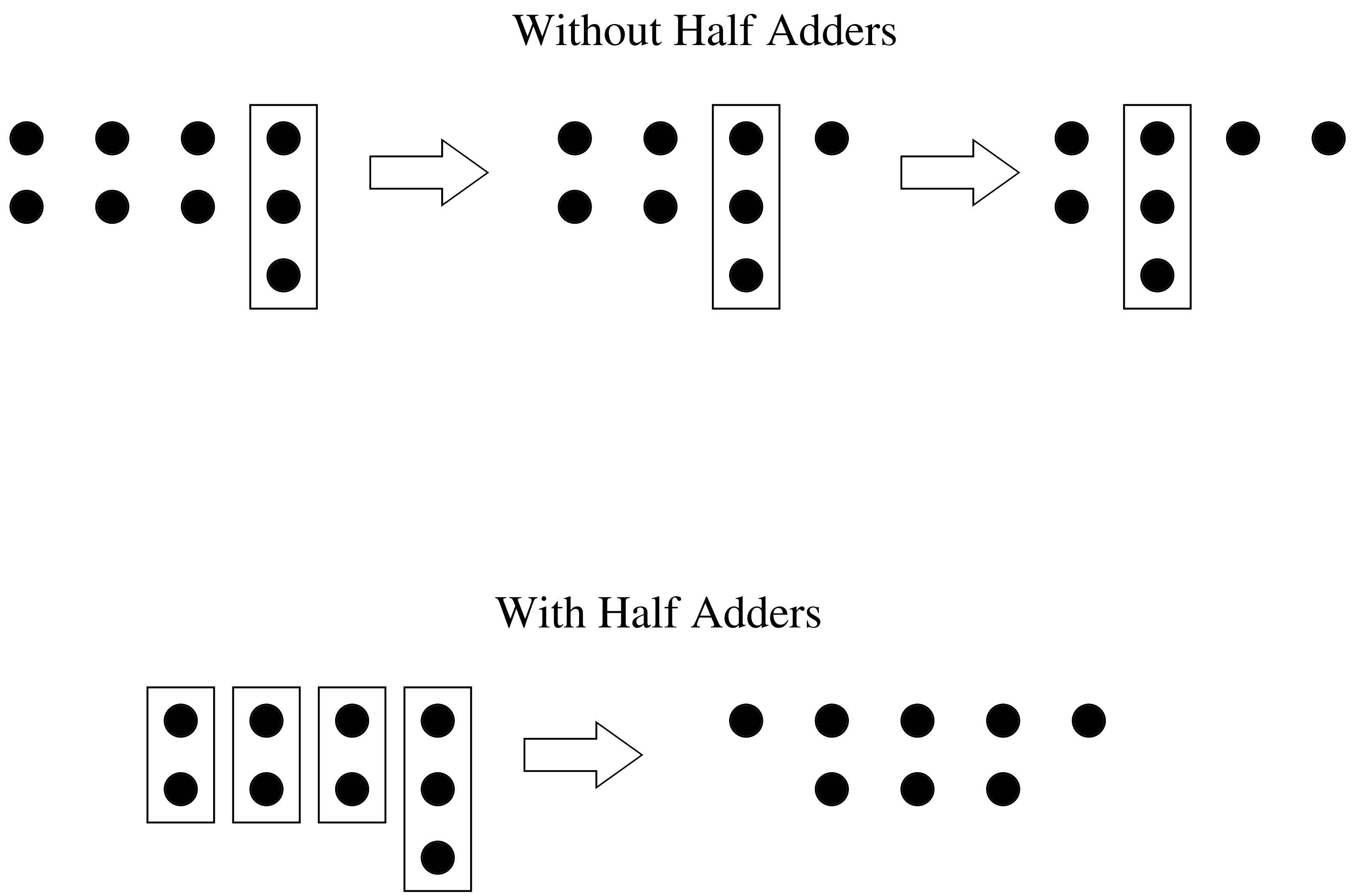


Figure 4.13: Why half adders are needed.

fan-out or wire length.

Observation 1 *If a group of CSAs and HAs are wired to produce the minimum possible propagation delay when adding a group of partial products, then there will be at most one HA for any group of wires with the same arithmetic weight.*

Proof : Assume that a minimum propagation delay wiring arrangement that has 2 or more HA's connected to wires of the same arithmetic weight. Pick any two of the HA's (left side of Figure 4.14). The HA's have a propagation delay from any input to any output of δ. The

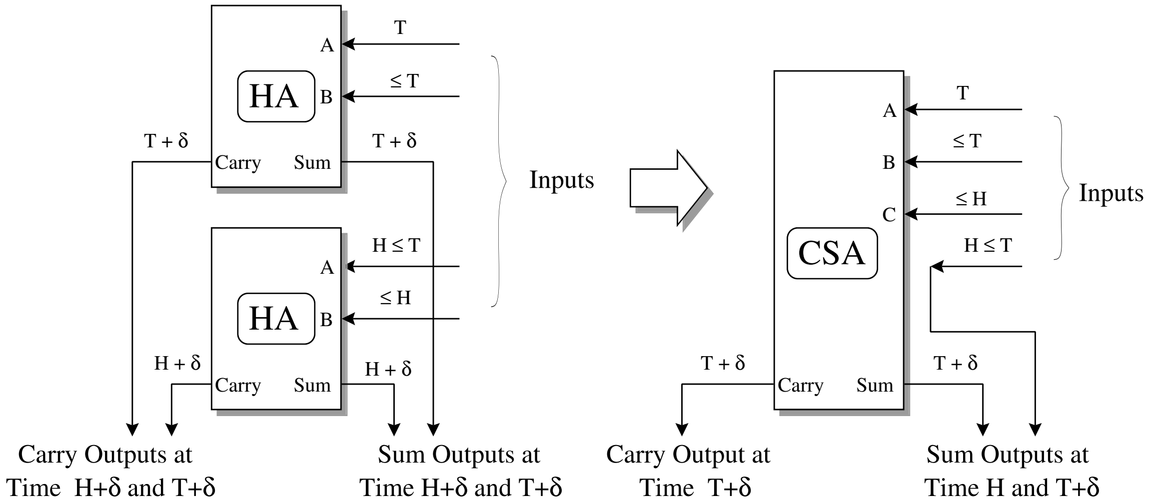


Figure 4.14: Transforming two HA's into a single CSA.

top HA in the figure has arrival times of T on the A input, and an arrival time of less than or equal to T on the B input. Thus, the propagation delay of the top HA is determined by the A input. Similarly, for the bottom HA the propagation delay is again determined by the A input arrival time of H, with the assumption that H is less than or equal to T. Such a configuration can be replaced by a single CSA (right side of Figure 4.14), where the inputs are rewired as shown. The outputs of the CSA configuration are available at the same time or before the outputs of the HA configuration, thus the propagation delay of the entire system cannot be increased. This substitution process can be performed as many times as needed to reduce the number of HA's connected to wires of the same arithmetic weight to 1 or 0. To emphasize, Observation 1 is true only when the delay effects of wires are ignored, and the propagation delay from inputs to outputs on CSAs and HAs is the same for all input to output combinations. As a result it does not apply to real circuitry, but it is used as a heuristic to assist in the placement of half adders.

Observation 2 *If group of CSAs and a HA are wired to produce the minimum possible propagation delay when adding a group of partial products, then the inputs of the HA can be connected directly to the output of the partial product generator.*

Proof : Assume that Observation 1 is applied to reduce the number of HA's attached to wires of a specific arithmetic weight to 1. If the HA is not connected directly to a partial product generator output, then there must be some CSA that is connected directly to a partial product generator output. This configuration is illustrated by the left side of Figure 4.15. The arrival times on the A inputs of both the CSA and the HA determine the output times

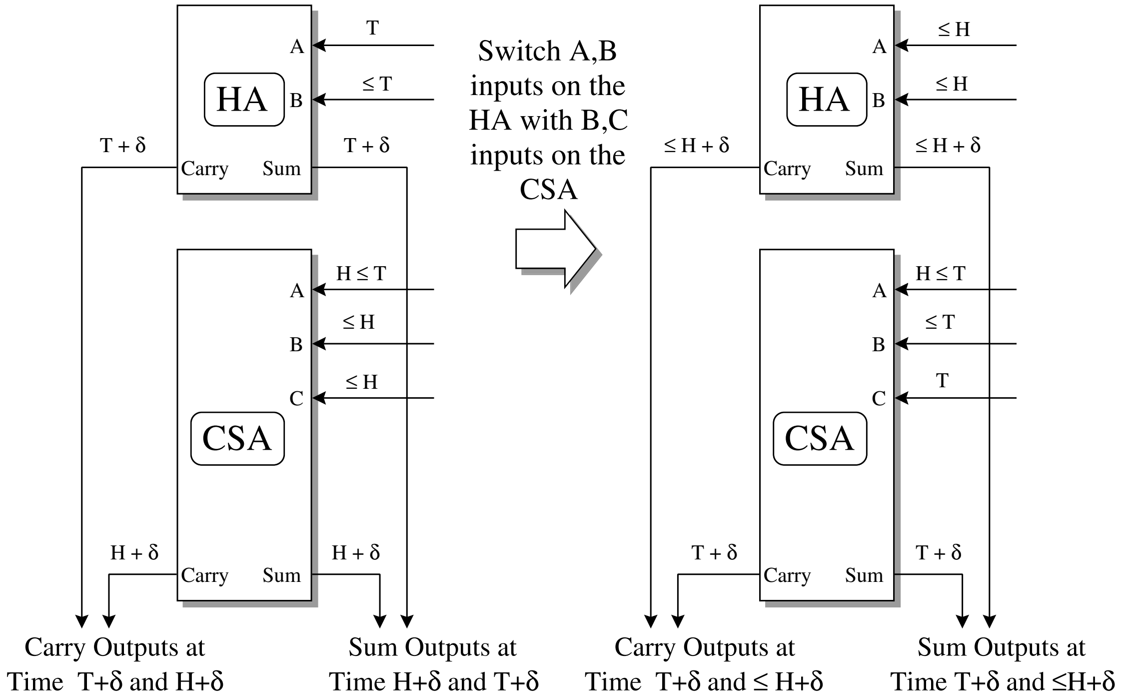


Figure 4.15: Interchanging a half adder and a carry save adder.

of the two counters[2]. The CSA A input arrives earlier than the A input on the HA. The two counters can be rewired (right side of Figure 1) such that the A input on the HA arrives

[2]A counter refers to either a CSA or a HA

earlier, without increasing propagation delay of the entire system. This process can be repeated until the HA is attached to the earliest arriving signals, which would be the output of the partial product generator.

Even though Observations 1 and 2 are not valid in the presence of wire delays and asymmetric input propagation delays, they can be used as the basis for a heuristic to place and wire any needed HAs. Half adders are wired as the very first counter in every column, and the multiplier is then wired as described above. The critical path time of the multiplier is then determined. Then starting with the most significant arithmetic weight, the half adder is temporarily removed and the network is rewired. If the critical path time increases, then it is concluded that a half adder is needed at this weight, and the removed half adder is replaced. If the critical path time does not increase, then the half adder is removed permanently. The process is then repeated for each arithmetic weight, giving a list of weights for which half adders are required.

4.3.3 Tree Folding

The layout tool, as described so far, organizes the partial products in rows of multiplexers. The shifting that occurs between partial products to allow for the different arithmetic weights, causes the layout to take a trapezoidal shape (refer back to Figure 4.10). Adding the CSAs exaggerate this shape even more, making it almost football shaped, since there are more CSAs in columns that have the most vertical partial product bits. This shape does not lend itself to rectangular fabrication. Although circuitry can sometimes be hidden in these areas, it is more efficient to use a layout methodology that produces a more rectangular shape. The method of aligning the partial products was mentioned earlier, but the wiring of the CSAs is more difficult, since outputs of many differing arithmetic weights appear in a single routing channel. Tree folding is another method of making the layout more rectangular. Figure 4.16 shows the right half of Figure 4.10, plus there is a black line through the third routing channel from the right. All multiplexers that lie to the right of this line are folded back under as shown in Figure 4.17. Although this would seem to create unusable holes of empty space, the technique of embedding CSAs (described below) among the partial product multiplexers is able to move CSAs into most of these holes, so

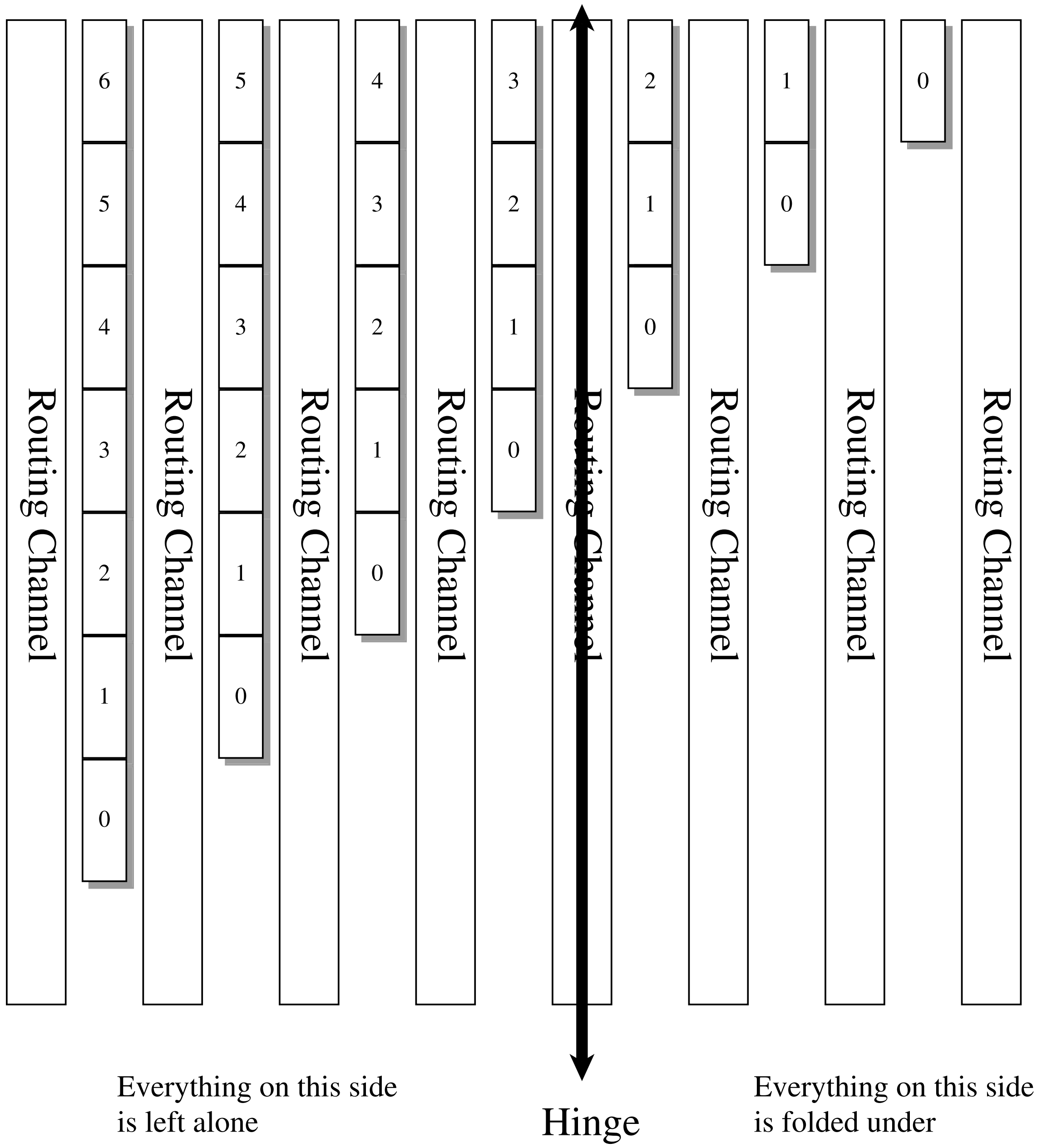


Figure 4.16: Right hand partial product multiplexers.

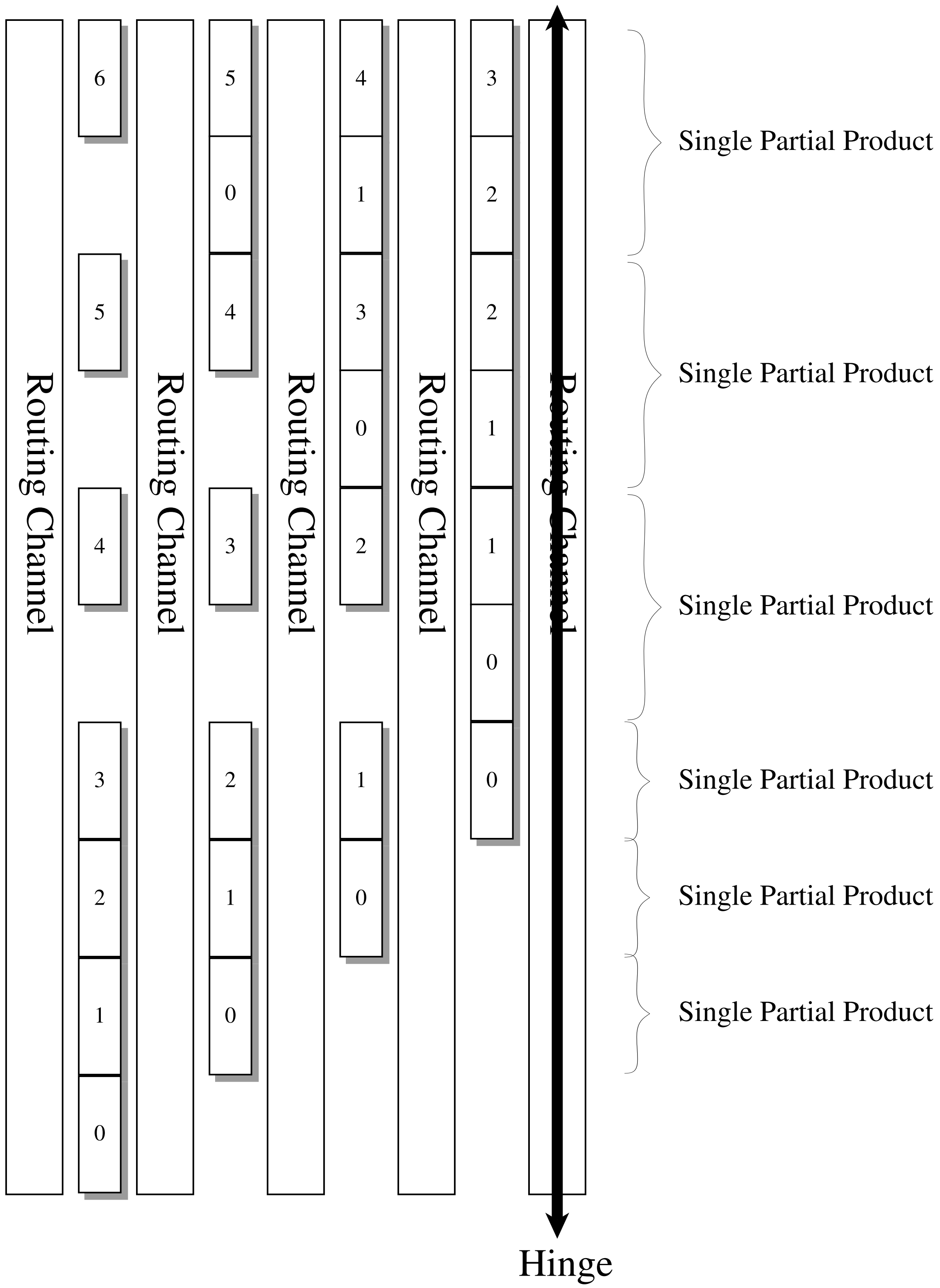


Figure 4.17: Multiplexers folded under.

very little space is wasted. The same scheme can be used on the left half of the layout. In general, this technique can eliminate almost half of the required routing channels. The program chooses the hinge point by iteration. The right most routing channel is used as the initial hinge point. The layout is done, and if the area is smaller than any previous layouts, the hinge point gets moved one column to the left. This continues until the smallest area is obtained. The method is then repeated for the left side.

The final result from the summation network emerges folded back upon itself, but some experiments were done with adder layouts and it seems as though the size and performance of the final carry propagate add is not effected significantly by this folding.

4.3.4 Optimizations

There are a number of optimizations which are done as the layout is being developed, to improve the area or reduce the delay.

Embedded CSAs

To further reduce the number of vertical wiring tracks needed in the routing channels, a CSA can be moved closer to the outputs that are connected to it's inputs. These outputs can come from either a partial product multiplexer or another CSA. For example, the configuration shown in the left half of Figure 4.18 takes 3 vertical routing tracks. Moving the CSA to a location between the outputs requires only 2 routing tracks (right side of Figure 4.18). To provide space for such movement, the initial placement of the partial product selection multiplexers has vertical gaps. There are also gaps created by the tree folding as described previously. As the CSAs are added, checks are made to determine whether a CSA can be moved into such an area, subject to the constraint that the propagation delay of the path through this CSA cannot increase. This overly constrains the problem, because not every CSA is along the system critical path. After the CSAs are all placed, and the critical path is determined, additional passes are done which attempt to move the CSAs into such locations to minimize the number of vertical routing channels.

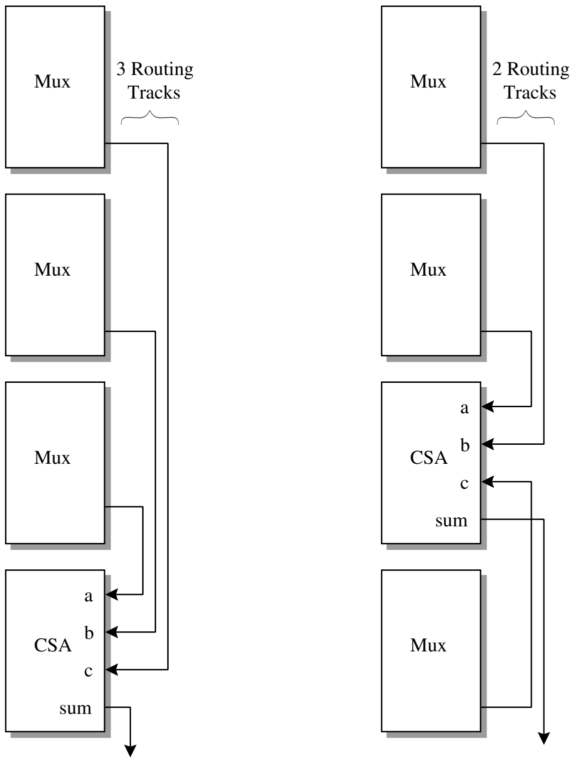


Figure 4.18: Embedding CSA within the multiplexers.

Wire Crossing

Wire crossing elimination is used to improve performance and wiring channel utilization. The left side of Figure 4.19 illustrates a possible wire crossing. These wire crosses are

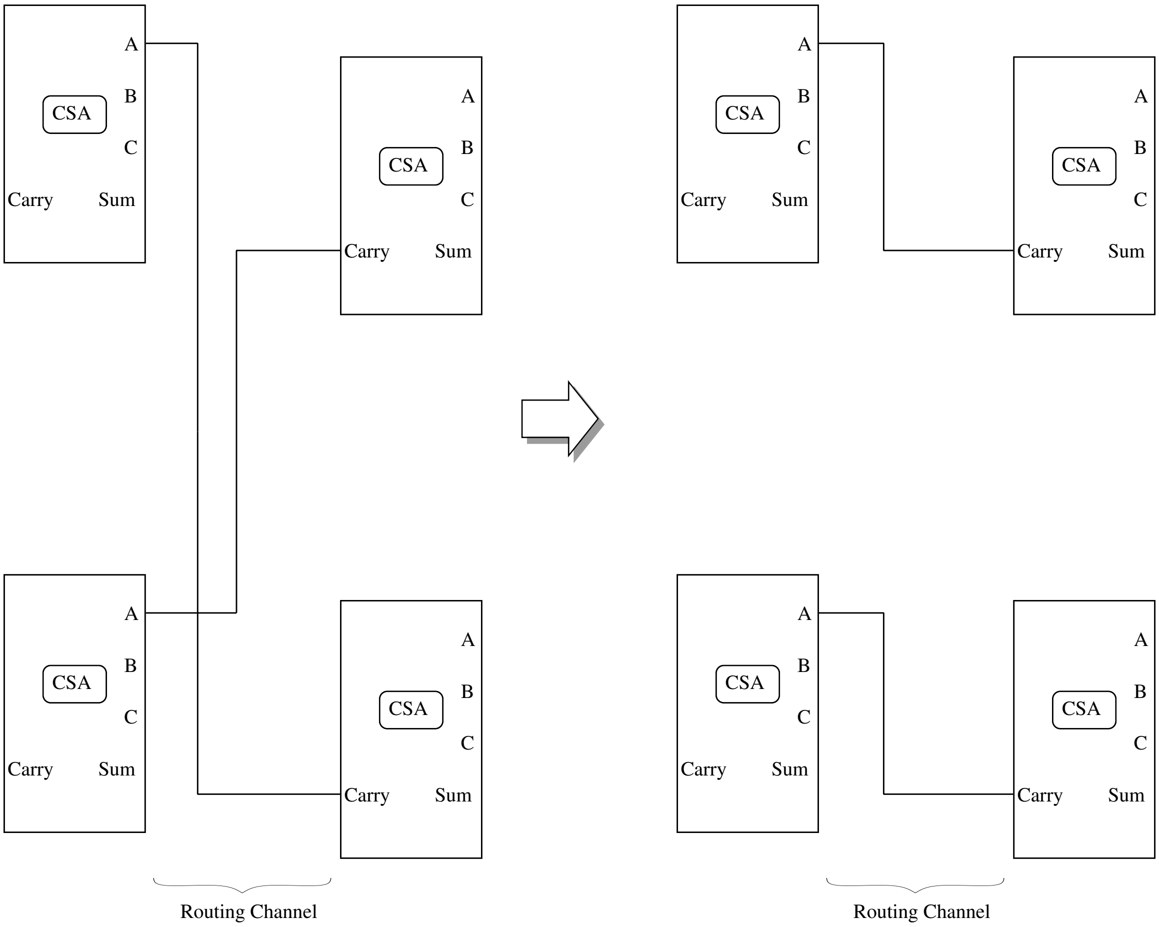


Figure 4.19: Elimination of wire crossing.

created when a CSA is moved upward in a cell column as described earlier. The inputs can be interchanged (right side of Figure 4.19), and the width of the routing channel reduced if the following three conditions are met :

- The wires must have the same arithmetic weight.
- The delay along the critical path must not increase.

- A cycle must not be created by the interchange. That is there cannot be feedback, either direct or indirect, from the output of a counter to one of it's own inputs.

Each wire crossing eliminated saves 2 routing tracks, allowing possible compression of the routing channel. The delay may also be reduced since the wires driven by the outputs are shorter.

Differential Wiring

A major performance gain can be obtained by selectively using differential ECL in place of standard single ended ECL. This optimization is illustrated by the circuit shown in Figure 4.20. The reference input in the standard gate is replaced by a second wire which

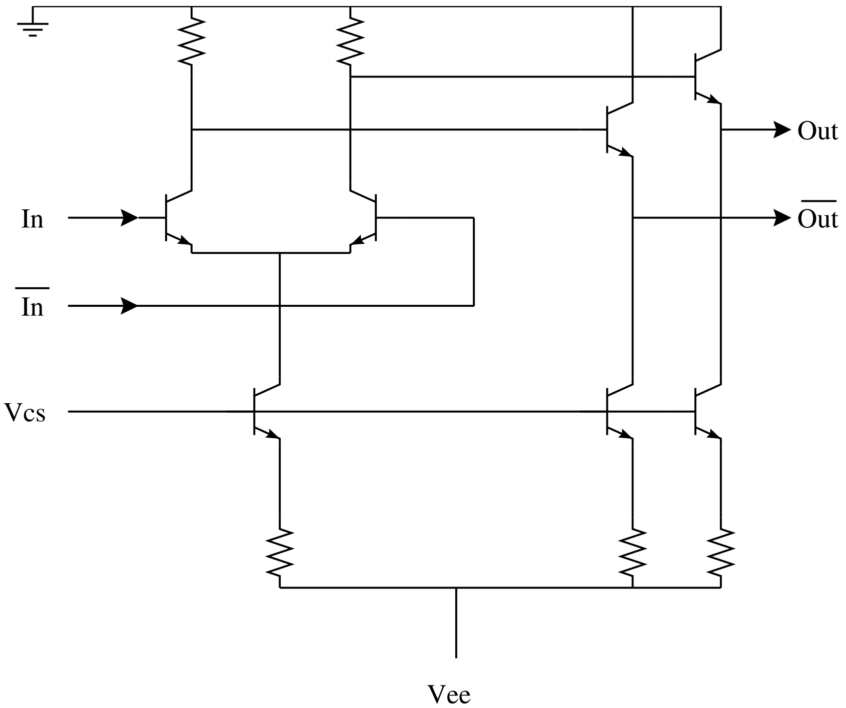


Figure 4.20: Differential inverter.

is always the complement of the input. The addition of the second wire allows the voltage swing on both wires to be half that of the single ended case, yet maintaining the same (or better) noise margin. The gate delay of the driving gate is halved, as is the wire delay. On the down side, the area and power consumption of the driving gate is increased, due to the second output buffer. A larger routing channel may also be needed to accommodate the extra signal wire required. This optimization is very useful in reducing the delay along critical paths. Differential wires are introduced according to the following rules:

- A candidate wire must lie along a critical path through the multiplier, and it must not already be a differential wire.
- The addition of the second wire must not increase the routing channel width.
- If no wire can be found that satisfies both of the above conditions, then find a wire that satisfies only the first condition and expand the routing channel.

This process is continued until no wires can be found that satisfy the first condition. The process may also be discontinued prematurely if this is desired.

Emitter Follower Elimination

Emitter followers are output buffers that are used to provide gates with better wire driving capability and also to provide any level shifting that is required to avoid saturating the input transistors of any gates being driven. For differential output gates, two emitter followers are needed. All single ended gates require some level shifting to facilitate the production of the reference voltages. Differential gates do not require such a reference voltage, so this level shifting may not be required. For short wires, the buffering action of the emitter follower is also not needed, so these emitter followers can be eliminating, reducing area and power consumption.

Power Ramping

The delay through a short wire (length $\leq$ 2mm) is inversely proportional to the current available to charge or discharge the wire (see Equation 1.2). This provides a direct trade-off that can be made between the power consumed by an output buffer and the delay through the wire driven by the buffer. In a full tree multiplier, there are large numbers of wires that do not lie along the critical path, thus there is the potential for large power savings by tuning the current in the emitter follower output buffer. In principle, a follower driving a completely non-critical wire could be ramped to a negligible current. For noise margin reasons, however, there is a limit to the minimum current powering a follower, so the practical minimum is about $\frac{1}{4}$ the maximum follower current.

The currents powering non-critical logic gates can also be reduced, increasing the propagation delay of the gate. The noise margin requirements are different for gates, so they can be ramped to lower currents than can the emitter followers. The minimum current is again limited, but this time by the fact that lower currents need larger resistor values in the current source powering the gate. This larger resistors can consume large amounts of layout area. Although the resistors can be hidden under routing channels, the practical limit seems to be about 10KΩ. This again provides a ratio of about $\frac{1}{4}$ between the smallest and largest currents allowed.

4.4 Verification and Simulation

The correctness of the layout is constantly monitored during the layout process, but it is still useful to have some form of cross checking to guard against the presence of software bugs. A verification pass is performed on the final net list. This verification consists of the following checks :

- All CSAs (carry save adders) have all inputs connected to something (No floating inputs).
- All CSAs in the summation network have all outputs connected to something (No bits are lost).
- All partial product multiplexer outputs are connected to a CSA input (No bits are lost).
- All inputs to a given CSA have the same arithmetic weight (Make sure the correct things are added).
- No input to a given CSA can be driven directly or indirectly by any output from the same CSA (no feedback).

- All wires have exactly one CSA input attached (Each partial product is added no more than once).
- All wires have exactly one output attached, which could come from either a partial product multiplexer or a CSA (No outputs are tied together).

Addition verification can also be performed by a transistor level simulation of the layout (see Section 5.5).

4.5 Summary

An automatic software tool which assembles summation networks for multipliers has been described. This tool produces placement and routing information for multipliers based upon a variety of algorithms, using a CSA as the basic reduction block. Most of the algorithms used in the tool for placement and routing have been developed by the process of trying many different methods and refining and improving those methods that seem to work. A number of speed, power and area optimizations have been presented.

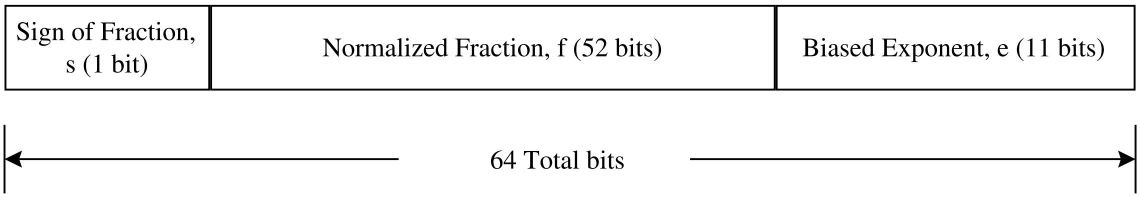
Chapter 5 will use this software tool to evaluate implementations using various partial product generation algorithm. Implementations produced with the tools will then be compared to other implementations described in the literature.

Chapter 5

Exploring the Design Space

This chapter presents the designs of a number of different multipliers, using the partial product generation methods described in Chapter 2. The speed, layout area, and power for multipliers implemented with each of these methods can only be accurately determined with a complete design, including the final layout. The layout generator described in Chapter 4 provides a mechanism with which a careful analysis can be performed, as it can produce a complete layout of the partial product generation and summation network. In combination with a design for an efficient carry propagate adder and appropriate multiple generators (both described in Chapter 3), a complete multiplier can be assembled in a mostly automated manner. Important measures can then be extracted from these complete designs.

The target multiplier for this study is based upon the requirements of IEEE-754 double precision floating point multiplication [12]. The format for an IEEE double precision number is shown in Figure 5.1. The IEEE representation stores floating numbers in a normalized, sign magnitude format. The fraction is 52 bits long, normalized (leading bit of 1), with the "1" implied and not stored. This effectively gives a 53 bit fraction. To meet the accuracy requirements of the standard the full 106 bit product must be computed, even though only the high order 53 bits will be stored. Although the low order 51 bits are used only in computing the "sticky bit" (if the low order 51 bits of the product are all 0, then the "sticky bit " is high - see Appendix B), all of the carries from the low order bits must be propagated into the high order bits. The critical path and most of the layout area ($>$ 95%) involved in a floating point multiplication is due to the multiplication of the fractions, so



$$\text{Number Represented} = (-1)^{s}\,(1.f)(2^{e-1023})$$

Figure 5.1: IEEE-754 double precision format.

this is the only area that will be addressed in the sections that follow.

Since the emphasis of this thesis is on speed, the delay goal for the complete fraction multiply is 5 nsecs or less.

5.1 Technology

All multiplier designs are based upon an experimental BiCMOS process[15]. A brief summary of the process technology is shown in Figure 5.1. Although this process is BiCMOS, the test designs use only bipolar ECL logic with 0.5V single ended/0.25V differential logic swings.

The basic circuits for a CSA and a Booth 2 multiplexer are shown in Figures 5.2 and 5.3. In order to provide some form of delay reference, the propagation delay curves for the CSA are shown in Figure 5.4. This figure shows the propagation delay vs load capacitance for a CSA with a 200μA tail current. There are three 0.5V swing single ended curves, corresponding to an output driven through an emitter follower and 0,1 or 2 level shifting diodes. Each emitter follower is powered by a 200μA pulldown current. Four curves are shown for 0.25V differential swings. The differential @0 output has no emitter followers, the others have a pair of emitter followers, each powered with 200μA, and 0,1, or 2 diodes per follower. In this technology 1 mm of wire corresponds to about 300fF. Figure 5.5 zooms in on the area where the load capacitance is less than 100fF. The dashed vertical line corresponds to the approximate capacitance that would be seen if another CSA was being driven through a wire that is twice the CSA height.

- Process :
  - 0.6μ (drawn) BiCMOS
  - 4 layer metal, thick MET4 for power
- Bipolar Transistors :
  - 16 GHz F_T @ $200\mu A$
  - $2K\Omega$/square polysilicon resistor
- CMOS (3.3V) :
  - 0.45μ nfet/pfet L_{eff}
  - nfet/pfet $V_T = \pm 0.6V$
  - 10.5 nm gate oxide thickness

Table 5.1: BiCMOS Process Parameters

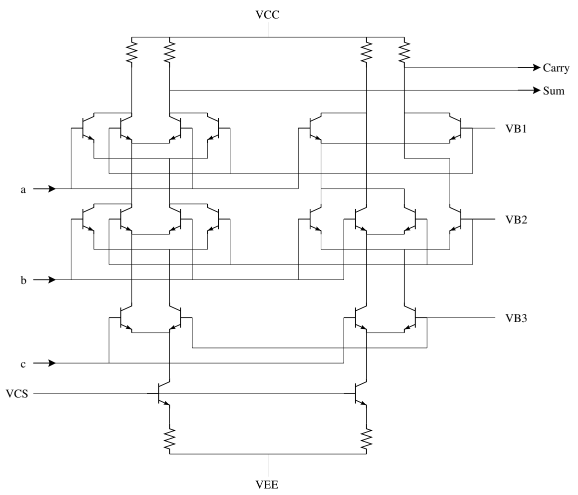


Figure 5.2: CML/ECL Carry save adder.

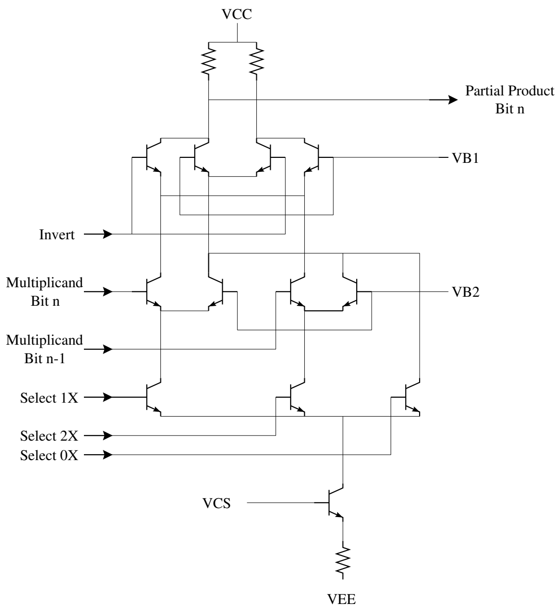


Figure 5.3: CML/ECL Booth 2 multiplexer.

5.2 High Performance Multiplier Structure

The basic structure of a multiplier is the same regardless of the particular partial product generation algorithm that is used. The multiplier structure used in this study is shown in Figure 5.6, and consists of a number of subsections which will be considered separately in the discussion to follow. The delay components of a multiplier based upon this structure are shown in Figure 5.7. In this figure time moves from left to right, with operations that can be performed in parallel arranged vertically. The delay through all blocks, except for the final carry propagate add, are dependent to some degree by the particular partial product generation algorithm that is being implemented. The software layout tool described in Chapter 4 produces the summation network, but the other parts also contribute significant delay and layout area. Evaluation of a particular multiplier implementation must include the effects of these other blocks.

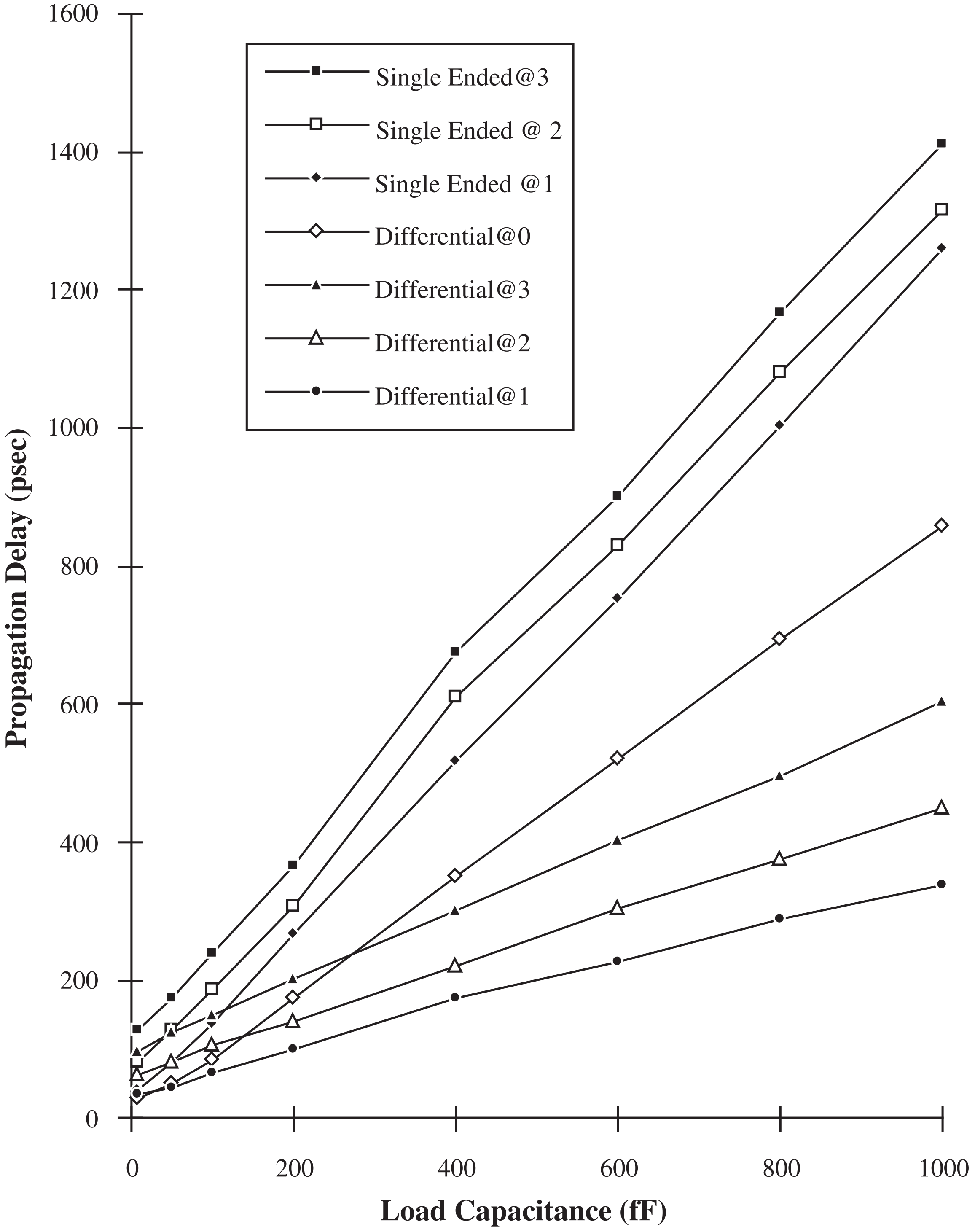


Figure 5.4: Delay curves for CSA adder.

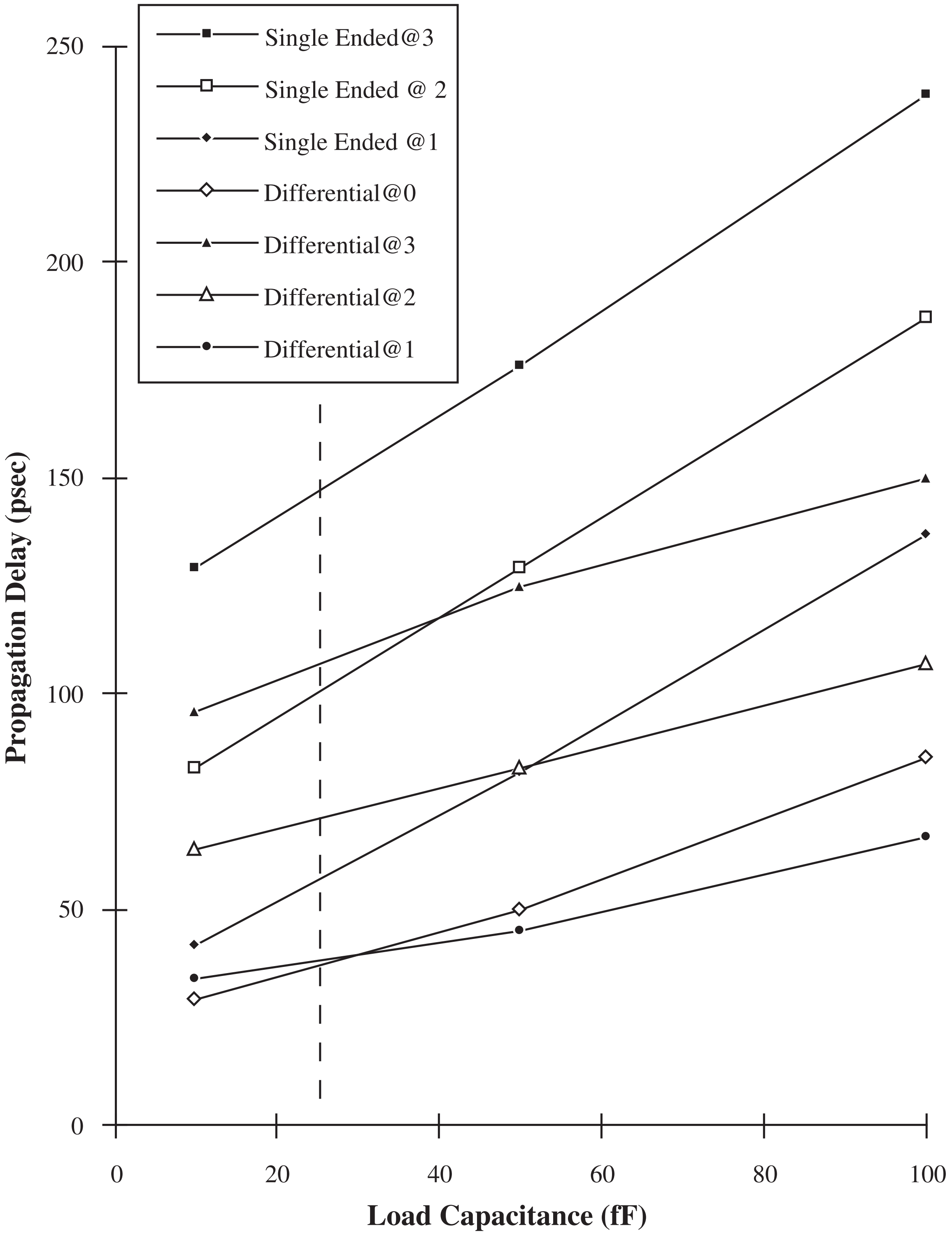


Figure 5.5: Delay for loads under 100fF.

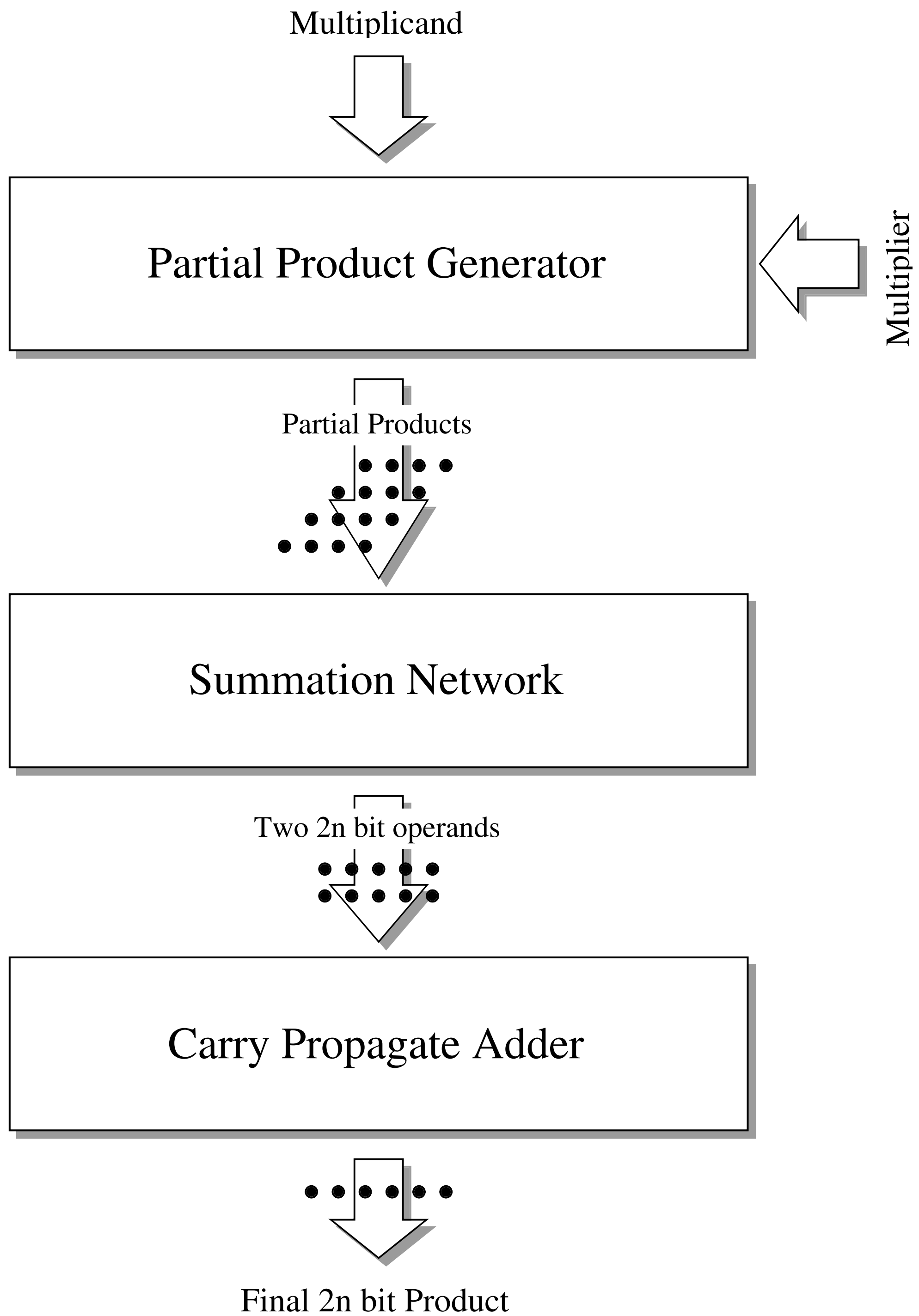


Figure 5.6: High performance multiplier.

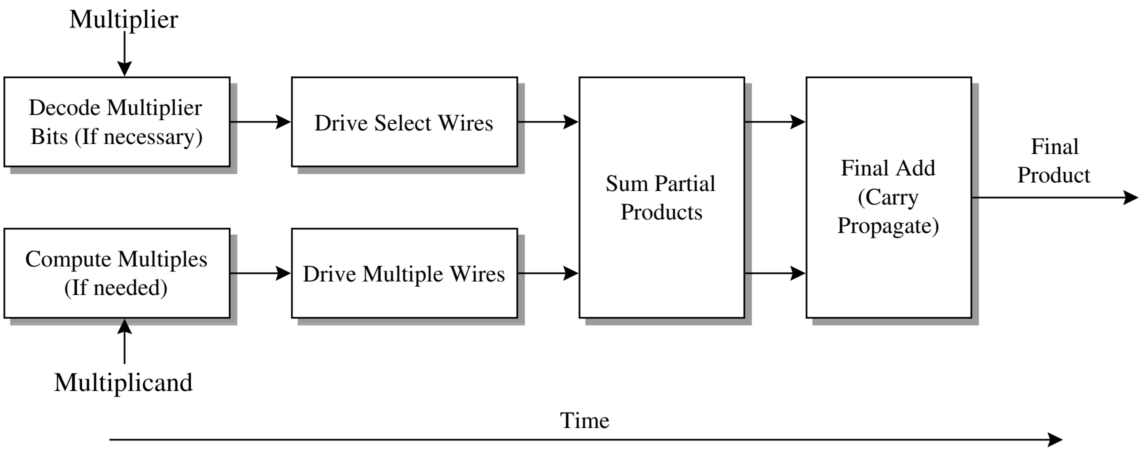


Figure 5.7: Multiplier timing.

Partial Product Selection and Select Wires

Each partial product is produced by a horizontal row of multiplexers which have common select controls (the layout tool may fold the row back upon itself). Using the dot diagrams of Chapter 2, a single horizontal row of dots corresponds to a row of multiplexers (or AND gates). The select controls are shared by all multiplexers used in selecting a single partial product, and in the layout scheme adopted here, run horizontally over the multiplexers (refer back to Chapter 4 for more a more detailed description). The select controls are composed of recoding logic, which use various bits of the multiplier to produce the required decoded multiplexer signals, such as select_Mx1, select_Mx2, select_Mx3, etc., which are in turn used to choose a particular multiple of the multiplicand in forming a given partial product. The decoded multiplexer signals are then fed to buffers which drive the long wires connecting the multiplexers. The low level circuit design of the output driver for each select takes advantage of the fact that the selects are mutually exclusive (only one is high at any given time) to reduce the power consumption. During a multiply operation and after the select lines have stabilized, exactly one of the select lines will be high. Therefore when the select lines need to switch, only one wire will be making a high to low transition, so a single pulldown current source can be shared by all 5 wires, instead of 5 separate pulldown current sources.

Figure 5.8 shows a simplified driver circuit using 2 select output drivers. To expand this to 5 (or more) select drivers, 3 (or more) additional driver circuits would have to be added, but they would all share the same pulldown current source shown in the figure. To

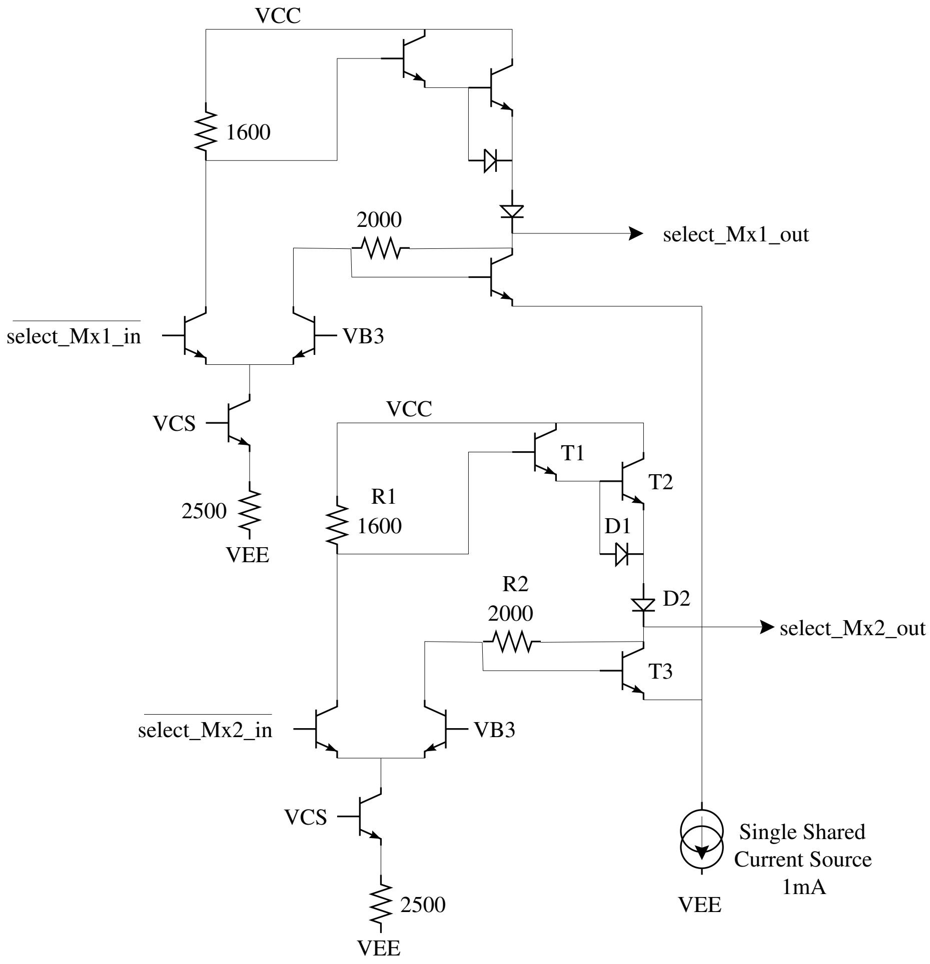


Figure 5.8: Dual select driver.

understand how this circuit works, consider the bottom driver in the figure. There are 4 major components. The driver gate, which connects to the input, an output pullup darlington formed by T1,T2, and D1[1], an output pulldown transistor T3 and the shared current source.

When the input transitions from low to high, all the gate current flows through R1, creating a voltage drop across R1. The output darlington voltage will be low. There is no current through R2 and no voltage drop between the base and collector of T3. This makes

[1]D1 reduces the gain of the output stage to reduce ringing

transistor T3 look like a diode that connects the shared current source to the output, pulling the output down very quickly, with the full force of the shared current source (remember exactly 1 output is high at any one time).

When the input transitions from high to low, all of the gate current is steered through R2, creating a voltage drop across R2, turning off transistor T3. At the same time there is no current through R1, therefore no voltage drop across R1, causing the darlington to pull up very fast. The current through R2 also provides a trickle current through the darlington to establish the output high voltage.

To reduce the wire delay, the voltage swing on the wires is reduced to 300mV, from the 500mV nominal swing for the other circuits, without sacrificing noise margin. Since exactly 1 wire is high at any given time, it can act like a reference voltage to the other 4 wires that are low (or are in transition to a low). As a result, much of the DC noise (such as voltage drops on the power supply rails) on the 5 select wires becomes common mode noise, in much the same way that DC noise becomes common mode noise for a differential driver. This allows a somewhat reduced voltage swing without sacrificing noise margins.

In the comparisons that follow, the recoding time plus the wire delay time is assumed to be fast enough that it is never in the critical path. Since the layout tool reports back the actual lengths of the select wires, the power driving the wire is adjusted to assure that this delay time.

Multiplicand and Multiples Wires

In parallel with the partial product selection, any required multiples of the multiplicand (M) must be computed and distributed to the partial product multiplexers. The delay can be separated into two components :

- **Hard Multiple Generation :** This applies only to higher ($\geq$ 3) Booth algorithms. Based upon the full carry propagate adder described below, the delay of a full 53 bit multiple is estimated to be 700 psec, with a power consumption of 350mW for the 3M multiple and 500mW for 5M and 7M. The area of these adders is about 0.5 mm^2 for 3M, and 0.7 mm^2 for 5M and 7M. The reduction in the size and area for the 3M multiple is obtained by using the method described in Chapter 3.

- **Multiple Distribution :** This is the wire delay due to the distribution of the bits of the multiplicand and any required multiples. These multiples run diagonally across the partial product multiplexers, so these wires are longer than the selection wires. Again the wire lengths are available as output from the layout program, and the power driving the wires can be adjusted (within reason) to give any desired wire delay.

The multiple generation and distribution is constrained to be less than 600 psec, by adjusting the power used in driving the long wires. This time is determined by the largest single transistor available in the technology (2mA), the typical wire length for multiples in driving to the partial product generator, and the delay of a buffering gate for driving the multiples. When a hard multiple is distributed, this constraint cannot be met (the hard multiple takes 700 psec to produce because it requires a long length carry propagate addition), so the driving current is limited to 2 ma (largest single transistor available) per wire and the propagation delay is increased.

The Summation Network

This block contributes the bulk of the layout area and power. The software layout program described in Chapter 4 generates complete layout of this section, providing accurate (within 10% of SPICE) delay, power, and area estimates. In addition the lengths of the select and multiples wires are also computed.

Carry Propagate Addition

Since all multipliers being considered in this section are 53x53 bits, producing a 106 bit product, a 106 bit carry propagate adder is needed. This adder can be considered as a fixed overhead, since it is the same for all algorithms. Such an adder has been designed and layed out, using the modified Ling scheme presented in Chapter 3. This adder accepts two 106 bit input operands and produces the high order 66 bits of the product, plus a single bit which indicates whether the low order 40 bits of the product are exactly zero. The important measurables for this adder are shown in Table 5.2. These adder parameters were obtained assuming a nominal -5V supply at 100° C, driving no load. The timing information is based on SPICE simulations of the critical path using capacitances extracted from the layout.

Area (mm^2)	Delay (nsec)	Power (Watts)
1.125	860	1.13

Table 5.2: 106 Bit Carry Propagate Adder Parameters

Because the adder design was done in a standard cell manner, the wire capacitance was increased by 50% in the simulation runs to account for possible Miller capacitance between simultaneously switching, adjacent wires.

5.2.1 Criteria in Evaluating Multipliers

There are three important quantities that can be used to evaluate the implementation of various multiplication algorithms.

Delay

All delays are for the entire multiply operation, not just the summation network time.

Power

The power values shown in the evaluation tables include all of the power necessary to operate the multiplier.

Layout Area

The area includes all components of the multiplier. The area can also impact the performance, in that larger area generally means longer wires and more wire delay.

5.2.2 Test Configurations

The evaluation of the various multiplier algorithms are based on five variations, which can be produced by adjusting various parameters of the layout tool. All configurations are based on a fully parallel implementation of the summation network.

- **Fastest :** This variation attempts to maximize the speed of the multiplier, ignoring area and power, except in the way they impact the performance (for example through wire lengths). Full use of differential wiring is used where possible to reduce the critical path time.

- **Minimum Area :** In this variation, all critical paths are fully powered, single ended swings. Differential wiring is not used, with the exception that differential, level 0 signals are used if no additional area is needed for the extra wire. This configuration is close to a traditional ECL implementation, giving the minimum area and minimum power for a full tree ECL design.

- **Minimum Width :** The goal is to improve the speed of the minimum area variation by allowing differential wiring wherever the impact on the layout area is negligible. Differential wiring is used where possible to reduce the critical path time, as long as the width of the routing channels (and hence the entire layout) does not increase. The use of differential wiring sometimes requires an extra output buffer, which increases the height of the layout slightly, so the actual area will be a little more than the minimum area variation. This variation is interesting in that it shows the performance increment, with only a small increase in layout area, that is possible with the selective use of differential wiring.

- **90% Summation Network Speed :** Since the cost of the maximum possible speed may be quite high (in terms of area and power), an interesting configuration is one in which the speed of the summation network is not pushed to it's absolute maximum, but instead is only 90% of the maximum speed. That is, the delay of the summation network in this configuration is $\frac{\text{Fastest}}{0.9}$.

- **75% Summation Network Speed :** Similar to the 90% speed configuration, except that the speed of the summation network is pushed only to 75% of the maximum speed available.

All of the above configurations vary only the speed, power and area of the summation network. Since there are other components in the complete multiplier (such as adders, recoders, wire drivers, etc.), the actual effect on the entire system will be reduced.

5.3 Which Algorithm?

Physically large layouts will have problems with wire delays, since the larger the multiplier, the longer the wires are that interconnect the various components of the multiplier. In addition, more circuitry generally means more power consumption. An appropriate choice of algorithm will produce as small a layout as possible, consistent with the performance goals. Various algorithms and implementation methods are available to the designer, and a careful evaluation of each is necessary to obtain a "good" design. Implementations of the conventional partial product generation algorithms described in Chapter 2 will be compared and contrasted, and some comments will be made about them. Then the implementations of the redundant Booth 3 algorithm (also presented in Chapter 2) and an improvement to the conventional Booth 3 algorithm will be compared to the conventional algorithms.

5.3.1 Conventional Algorithms

The conventional algorithms to be compared are based upon 53x53 unsigned multiplication. The results include all components of each multiplier and are are summarized in Table 5.3 and shown graphically in Figures 5.9, 5.10, and 5.11.

Comments on Conventional Algorithm Implementations

Referring to Table 5.3 it is obvious that simple multiplication is markedly inferior to the Booth based algorithms in all important measures. Others have reached different conclusions, such as Santoro [24], Jouppi et el[13], and Adlietta et el [1], so some explanation is in order.

- Power - The Santoro and Jouppi implementations are based on CMOS. The power characteristics are quite different between ECL and CMOS designs, the former being dominated by static power, the latter almost entirely dynamic power. Consequentially, power consumption measurements based upon one technology probably can not be applied directly to the other. It seems possible, however, that a CMOS multiplexer might consume less power than a CMOS CSA, if only because the former has one output and the latter has two, so Booth encoding may still save power.

Variation	Algorithm	Delay (nsec)	Area (mm^2)	Power (Watts)
Fastest	Simple	3.1	33.0	26.9
	Booth 2	2.6	15.4	14.4
	Booth 3	3.0	11.0	9.7
	Booth 4	3.1	13.7	10.3
Minimum Width	Simple	4.2	18.2	15.0
	Booth 2	3.5	9.3	7.2
	Booth 3	3.7	8.0	6.1
	Booth 4	3.7	10.7	7.3
Minimum Area	Simple	4.7	17.1	11.1
	Booth 2	3.7	9.2	6.4
	Booth 3	4.0	7.8	5.3
	Booth 4	4.0	10.6	6.9
90% Tree Speed	Simple	3.3	29.4	24.5
	Booth 2	2.8	14.3	12.9
	Booth 3	3.2	10.4	9.0
	Booth 4	3.2	12.6	9.8
75% Tree Speed	Simple	3.7	24.5	20.4
	Booth 2	3.0	12.0	10.9
	Booth 3	3.4	9.3	7.8
	Booth 4	3.5	11.4	8.6

Table 5.3: Delay/Area/Power of Conventional Multipliers

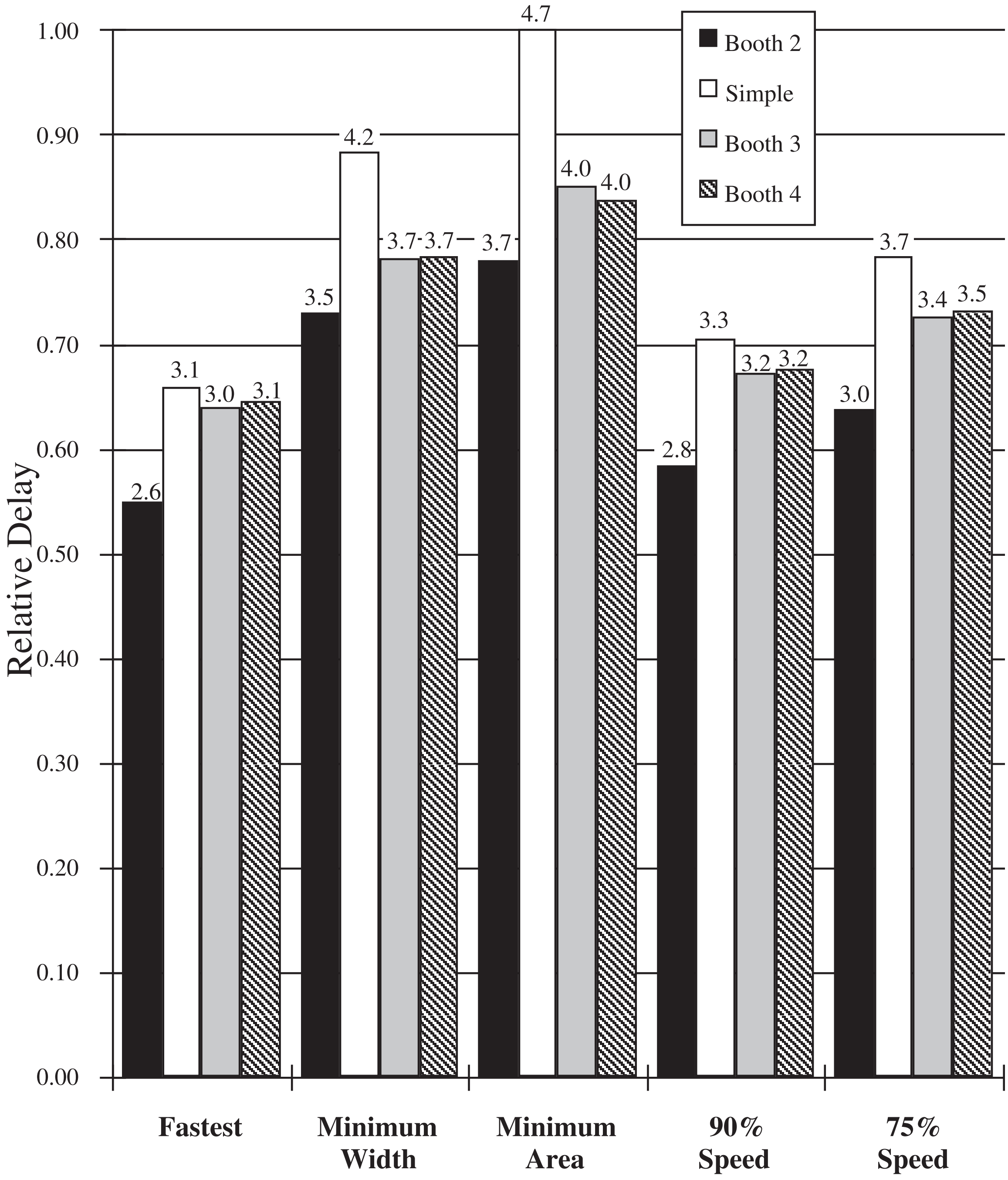


Figure 5.9: Delay of conventional algorithm implementations. Delays are in nsecs.

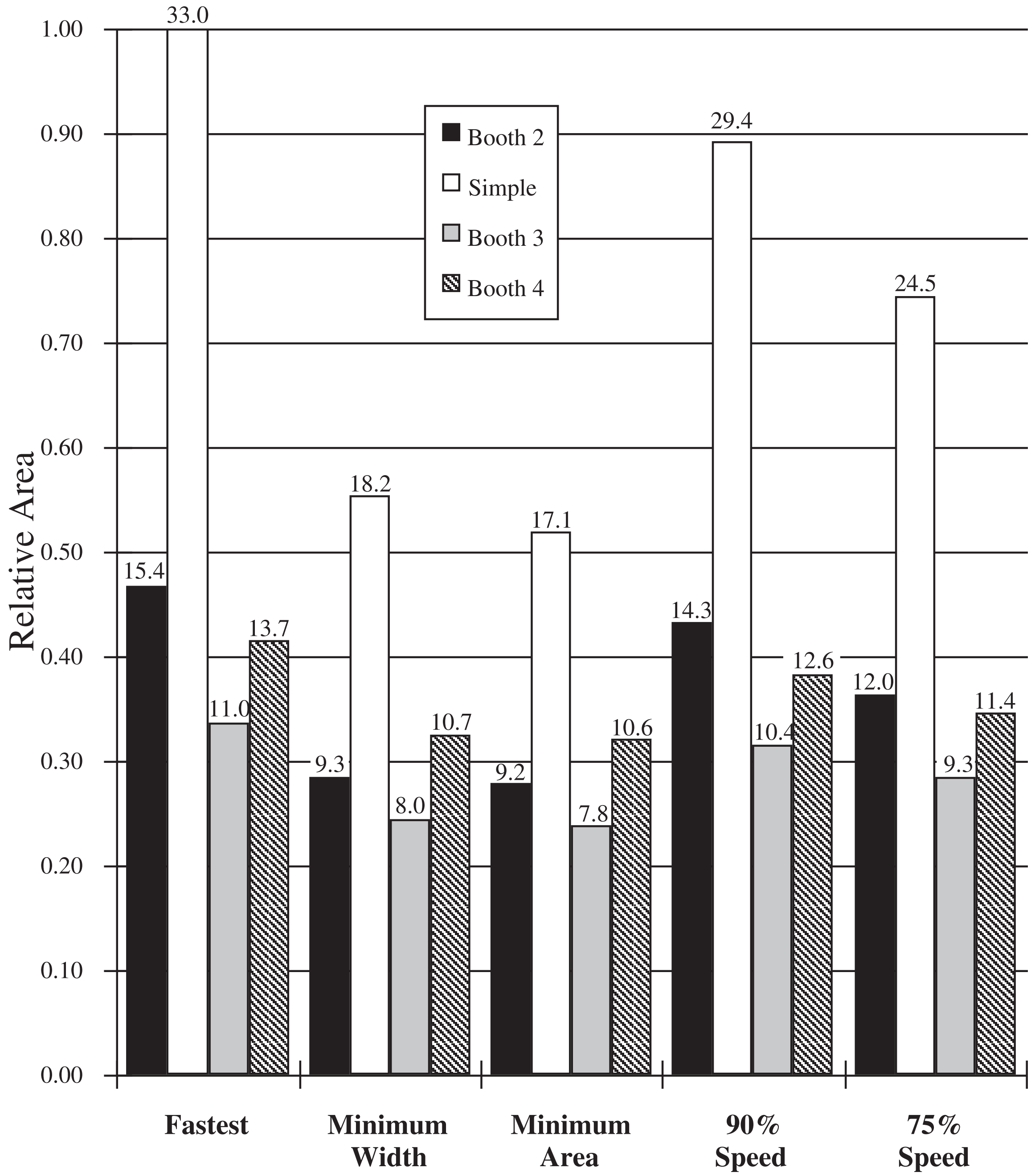


Figure 5.10: area of conventional algorithm implementations. Areas are in mm^2.

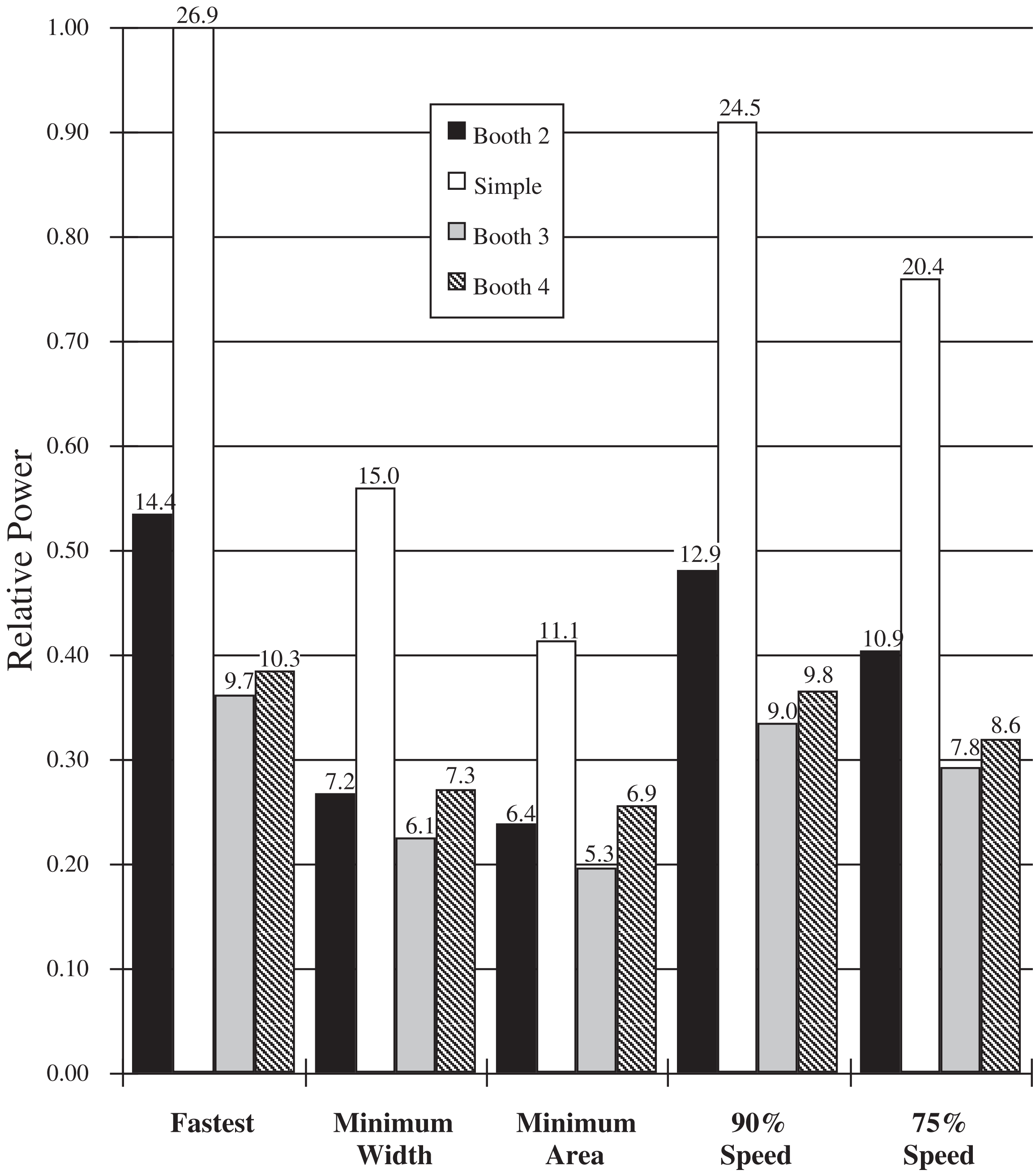


Figure 5.11: Power of conventional algorithm implementations. Power is in Watts.

The multiplier described by Adlietta is based on an ECL implementation, but is a gate array based design. ECL custom design allows the construction of a Booth multiplexer using 2 current sources (one for the multiplexer, one for the emitter follower output stage), whereas a CSA requires 4 current sources. The Booth 2 algorithm essentially replaces half of the CSAs required by simple multiplication with an equal number of multiplexers, at a considerable savings in power. If the gate array library doesn't have a 2 current source Booth multiplexer available, then it will have to be constructed out of multiplexer and an EXCLUSIVE OR. This would require 4 current sources, increasing the power consumption significantly and probably removing any difference in power consumption between the two methods.

- Delay - Simple multiplication is not significantly slower than Booth based implementations. Even though there are twice as many partial products to be added, the delay through the summation network is basically logarithmic in the number of partial products, minimizing the difference. Any difference can be explained by the replacement of the the top two layers of CSA delay with a single multiplexer delay, the delay of a CSA and a multiplexer being comparable. Also the extra area of simple multiplication contributes to longer wires, and thus longer delays.

- Area - The Booth 2 multiplexers used in this study are 24.6μ in height, compared to 31.6μ for a CSA (the widths are the same). A one for one replacement of CSA's with multiplexers, as happens when comparing simple multiplication to Booth 2 multiplication, should result in a modest reduction in total layout area. However, simple multiplication still requires AND gates for the selection of the partial products. The actual logic gate can frequently be designed into a CSA, with only a slight increase in area of the CSA. The wires distributing the multiplicand and the multiplier throughout the tree still require area, so the partial product selection still requires non-zero layout area. The remaining area difference can be explained by level shifters that are required for the multiplicand at $\frac{2}{3}$ of the inputs of all of the top level CSAs.

  Santoro observes that the size of the Booth multiplexers is limited by the horizontal wire pitch. Figure 5.12 shows a possible CMOS multiplexer. This particular version has 4 horizontal wires crossing through each row of multiplexers that create a single

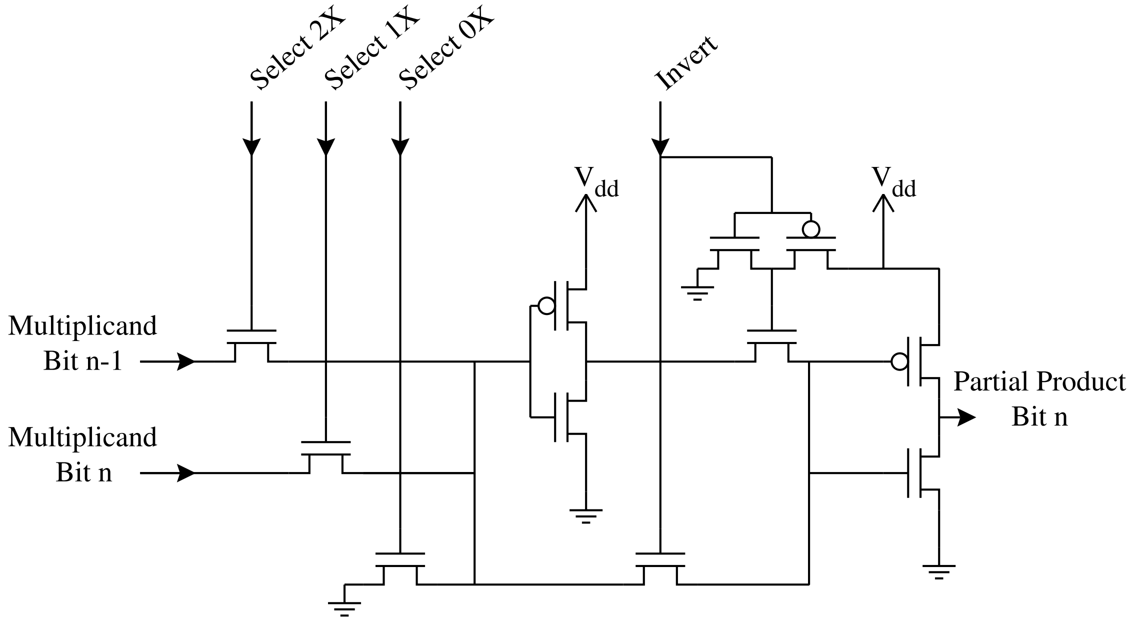


Figure 5.12: CMOS Booth 2 multiplexer.

partial product (other designs could have from 3 to 5 horizontal wires). Assuming 5 horizontal wires per partial product, an NxN bit Booth multiplier would have $\frac{5N}{2}$ total horizontal wires whereas simple multiplication would have N horizontal wires. If a CSA is exactly the same size as a Booth multiplexer, then simple multiplication would still be larger due to the N horizontal wires needed to control the AND gates which generate the partial products.

If the multiplexers are not wire limited, it is extremely unlikely that a multiplexer will be larger than a CSA, since the circuit is much simpler. Figures 5.2, 5.3, 5.12, and 5.13 show designs for ECL and CMOS multiplexers and CSAs and clearly the multiplexers are simpler than the CSAs.

The recoders that drive the select lines which control the multiplexers or AND gates could explain how it might be possible for simple multiplication to be comparable (or even smaller) to Booth encoding in CMOS. A relatively small bipolar transistor drives a large amount of current, so increasing the number of horizontal wires does not increase the size of the Booth multiplexer select drivers significantly. In contrast,

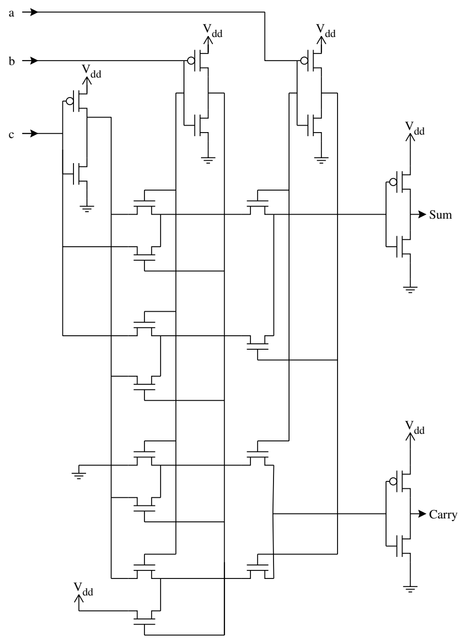


Figure 5.13: CMOS carry save adder.

CMOS will require additional large area transistors to drive the additional long select wires. The increase in the number of long select wires, from N to $\frac{5N}{2}$, may increase the area of the select generators enough to overcome the modest savings provided by Booth encoding, if the 5 wire version of the multiplexer is used.

Returning to Table 5.3, the Booth 4 algorithm has no advantage over the Booth 3 algorithm. The reason for this is that the savings in CSAs that result from the reduction in partial products is more than made up for by the extra adders required to generate the 2 additional hard multiples. The partial product select multiplexers are also almost twice the area (80μ vs 40.5μ in height). Booth 4 may become more competitive if the length of the multiplication is increased, since the area required for the hard multiple generation grows linearly with the length, while the area in the summation network grows as the square of the length. For lengths $\leq$ 64, Booth 4 does not seem to be competitive.

In summary, only Booth 2 and Booth 3 seem to be viable algorithms. Booth 2 is somewhat faster, but Booth 3 is smaller in area and consumes less power.

Delay (nsec)	Area (mm^2)	Adder Power (Watts)	Driver Power	Total Power
0.2	0.53	0.50	0.76	1.26

Table 5.4: Delay/Area/Power of 55 Bit Multiple Generator, built from 14 bit Subsections

5.3.2 Partially Redundant Booth

Chapter 2 presented a new class of multiplication algorithms, combining the Booth 3 algorithm with a partially redundant representation for the hard multiples. In principle, use of this algorithm should be able to approach the small layout area and power of the straight Booth 3 algorithm, with the speed of the Booth 2 algorithm. Implementations using this algorithm require the determination of an extra parameter, the carry interval in the partially redundant representation of the multiples. Before comparing this algorithm with the more conventional algorithms described above, a reasonable value for this parameter is needed. A small carry interval reduces the length of the small adders necessary for multiple generation, however too small an interval causes the number of CSAs (and so the power and area) to go up. A large interval reduces the number of CSAs required, but the delay of the carry propagate adder generating the small multiples increases the total delay through the multiplier.

Redundant Multiple Generation

The model for the short multiples adders will be based upon the actual implementation of a 14 bit 3X adder. Simple modifications will be made to allow the adder length to vary. The vital statistics for this 14 bit multiple generator, when it is used to construct a 55 bit multiple generator, are shown in Table 5.4. The delay does not include the time to drive the long wires at the output, as this delay is accounted for separately.

Using the method described in Chapter 3, it is possible to build short multiple generators of up to 14 bits using only two stages of logic. The delay of the longer generators is slightly more than that of the shorter adders, but the delay difference can be minimized by using more power along a single wire that propagates an intermediate carry from the low order 8 bits to the high order 6 bits. Although the delay is really a continuous function of the length of the adder, the difference between adders of similar lengths is minimal, of the order of 50 psecs between a length 5 adder and a length 14 adder. Although this is a significant variation in the adder times (25%), it is a very small fraction of the total multiply time (2% or less). The power consumption per bit is also roughly constant, with the difference between a length 5 adder and a length 14 adder being about 10mW. Since most of the power and delay involved in the multiple generation is in driving the multiples to all the partial product multiplexers, a more refined model will not be presented. Because the delay and power differences between the shorter multiple generators and the longer ones are very small, they will be ignored.

Varying the carry interval

Tables 5.5 and 5.6 shows the implementation parameters for the redundant Booth 3 algorithm as the carry interval is varied from 5 bits to 14 bits. The results are also shown graphically in Figures 5.14, 5.15, and 5.16.

Comments on the redundant Booth algorithm

Referring to Tables 5.5, 5.6 and Figures 5.14, 5.15 and 5.16 some general patterns can be discerned. Generally, the delay is not dependant on the carry interval. This is due to the logarithmic nature of the delay of the summation network. There are occasional aberrations (such as the data for a carry interval of 8), but these are due to fact that the layout program happens to stumble upon a particularly good solution under some circumstances. The area shows a more definite decrease as the carry interval is increased, again a pretty much expected result, since fewer CSAs and multiplexers are required. A somewhat surprising result is that the power, like the delay, is mostly independent of the carry interval. The reason for this is that most of the additional CSAs required as the carry interval is decreased lie off of the critical path, so these CSAs can be powered down significantly without increasing the delay. In addition, the summation network has been made so fast that the total delay is beginning to be dominated by the final carry propagate adder, and the driving of the long wires that distribute the multiplicand and select control wires through the summation network, not the delay of the summation network itself. It seems as though any carry

Variation	Carry Interval	Delay (nsec)	Area (mm^2)	Power (Watts)
Fastest	5	2.7	16.4	12.1
	6	2.8	15.8	11.5
	7	2.7	14.9	11.4
	8	2.6	14.0	11.7
	9	2.6	13.7	10.7
	10	2.7	13.6	10.8
	11	2.7	12.8	10.5
	12	2.7	13.1	10.7
	13	2.6	12.6	10.7
	14	2.6	13.0	10.5
Minimum Width	5	3.4	11.0	7.1
	6	3.5	10.7	6.9
	7	3.5	10.5	6.7
	8	3.5	9.8	6.4
	9	3.3	9.8	6.9
	10	3.5	9.5	6.4
	11	3.7	9.1	6.0
	12	3.6	9.4	6.2
	13	3.4	9.1	6.3
	14	3.6	8.9	5.8
Minimum Area	5	4.0	10.9	6.1
	6	3.8	10.5	6.1
	7	3.9	10.2	5.9
	8	3.6	9.7	6.3
	9	3.9	9.6	5.7
	10	3.7	9.4	6.0
	11	3.8	9.1	5.9
	12	3.7	9.3	5.9
	13	3.6	8.9	5.9
	14	3.7	8.9	5.6

Table 5.5: Delay/Area/Power of Redundant Booth 3 Multipliers

Variation	Carry Interval	Delay (nsec)	Area (mm^2)	Power (Watts)
90% Tree Speed	5	2.8	14.9	11.0
	6	2.9	14.2	10.6
	7	2.8	13.7	10.6
	8	2.7	13.2	10.6
	9	2.8	12.6	9.9
	10	2.8	12.6	9.7
	11	2.9	12.0	9.5
	12	2.8	12.0	9.4
	13	2.7	11.7	9.5
	14	2.8	12.2	9.7
75% Tree Speed	5	3.1	12.0	8.6
	6	3.2	11.8	8.3
	7	3.1	12.0	8.8
	8	3.0	12.0	9.3
	9	3.1	10.6	8.2
	10	3.1	10.8	8.3
	11	3.1	10.8	8.2
	12	3.1	10.6	8.1
	13	3.0	10.6	8.4
	14	3.1	10.3	8.0

Table 5.6: Delay/Area/Power of Redundant Booth 3 Multipliers (continued)

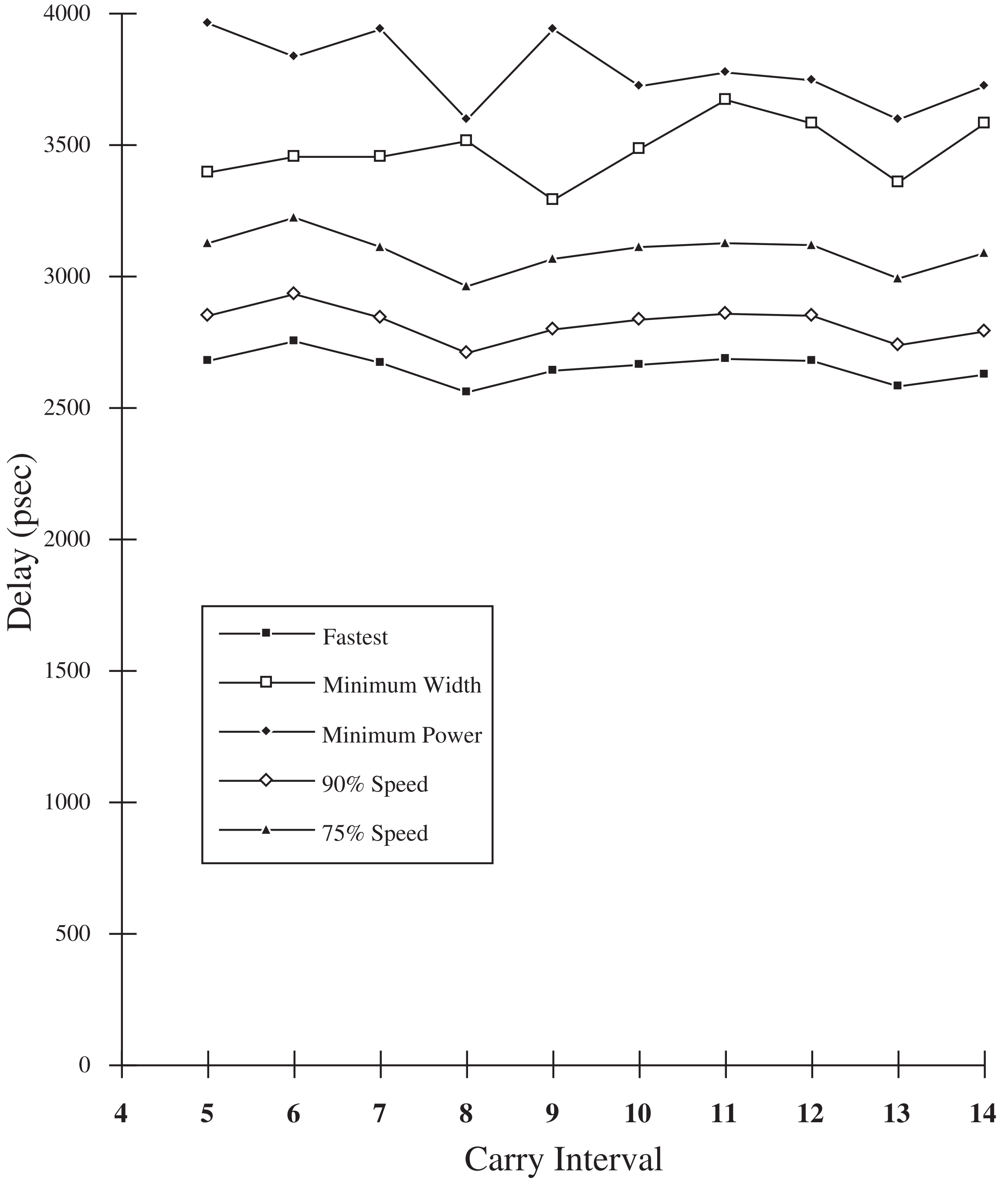


Figure 5.14: Delay of redundant Booth 3 implementations.

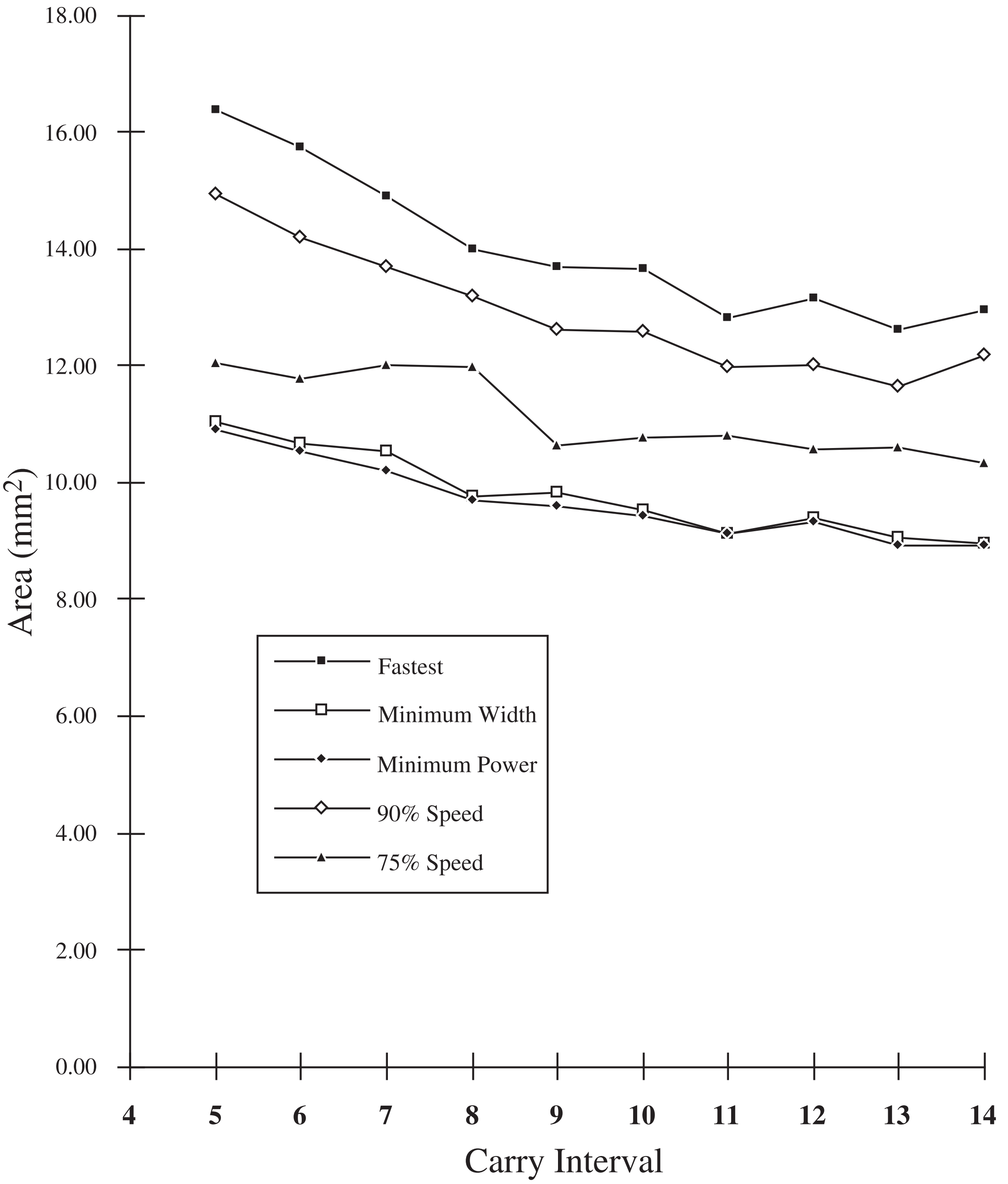


Figure 5.15: Area of redundant Booth 3 implementations.

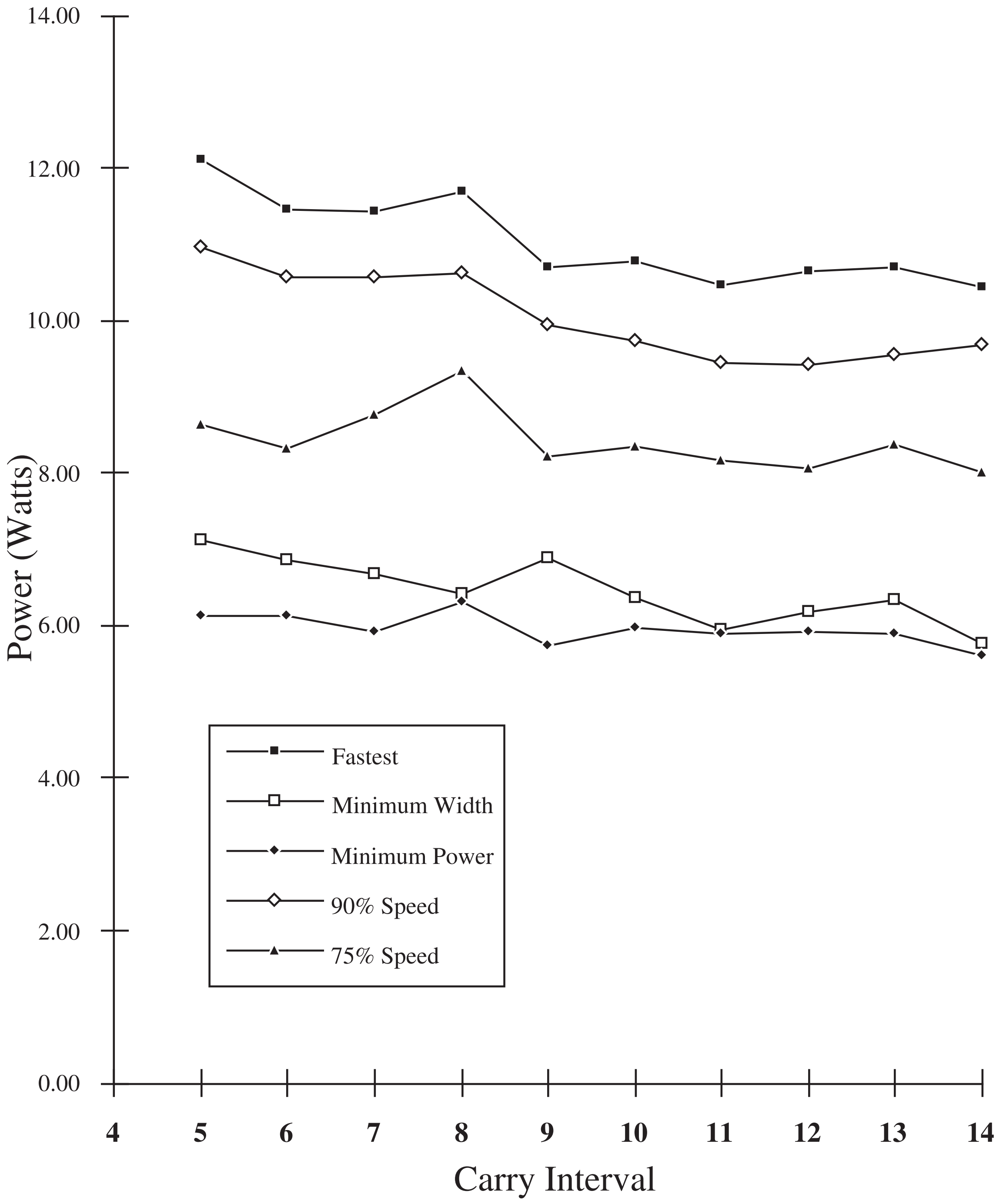


Figure 5.16: Power of redundant Booth 3 implementations.

Bits	Arrival Time
0-14	200 psec
15-36	400 psec
37-54	700 psec

Table 5.7: Improved Booth 3 - Partial Product Bit Delays

interval between 10 and 14 are about equally acceptable.

There is no indication that carry interval values that are not relatively prime to 3 are any worse than any other carry interval, as hinted at towards the end of Chapter 2. This is because this effect is buried by other effects, such as wire delays, asymmetric input delays on the CSAs, the logarithmic nature of the delay of the summation network, and the optimization efforts of the layout program.

5.3.3 Improved Booth 3

To illustrate the versatility of the layout program, a different kind of optimization, based upon the Booth 3 algorithm is also presented. Normally, the bits of the "hard multiple" are assumed to be available at about the same time, as would be the case (approximately) with a carry lookahead based hard multiple generator. That is, bit 0 of the hard multiple is assumed to be available at about the same time as the highest order bit. A multiple generator based upon a ripple carry adder would not have a uniform arrival time, but instead the bits of lower significance would be available earlier than bits of high significance. From the point of view of the summation network in a multiplier, parts of a single partial product would be available at different times. A full ripple carry adder is far too slow to use in a large multiplier, instead the model used here is based on a carry lookahead adder, where low order bits which require few levels of lookahead are available early, and higher order bits are later, due to additional levels of lookahead and longer wires. The assumed delay for various bits of an adder which multiplies by 3 is shown in Table 5.7. Taking advantage of the differing arrival times of the hard multiple would be difficult using a conventional tree approach, such as a Wallace tree or a 4-2 tree, but the layout program can take advantage of early arriving bits to reduce the power and area of the summation network.

5.4 Comparing the Algorithms

Figures 5.17, 5.18, and 5.19 compare the implementation delay, power and area of the two conventional algorithms (Booth 2 and Booth 3) with the redundant Booth 3 and improved Booth 3 algorithm. The carry interval for the redundant Booth 3 algorithm was chosen to be 14, mainly because that was the size used in the test implementation to be described below. Any interval between 10 and 14 could have been chosen with similar results. Like the earlier comparisons, the basic multiplier is a 53x53 bit integer multiply. The redundant Booth 3 algorithm is essentially the same speed as the Booth 2 algorithm, yet makes modest savings in both area and power consumption. The improved Booth 3 algorithm has better power and area numbers than the conventional Booth 3 algorithm, but is roughly comparable in performance. Because of the early arrival of certain bits of the partial products, less use is made of differential wiring to maintain the performance, which reduces the area and power requirements.

5.5 Fabrication

In order prove the design flow, a test multiplier was fabricated in the experimental BiCMOS process described previously. The implementation described here is that of a 53x53 integer multiplier producing a 106 bit product. Due to pad and area limitations, only the high order 66 bits of the product are computed, with the low order 40 bits encoded into a single "sticky" bit, using the method described in Appendix B.1. The algorithm used was the redundant Booth 3 method described in Chapter 2, with 14 bit small adders. CMOS transistors are used only as capacitors on internal voltage references.

After the entire multiplier was assembled, and the final design rule checks performed, a net list of the entire multiplier was extracted from the layout and run through a custom designed simulator built upon the commercial simulator LSIM (from Mentor Graphics). The simulator works at the transistor and resistor level, and is approximately 3 orders of magnitude faster than HSPICE at circuit simulation. It is not quite as accurate, and also provides no timing information. Approximately 3000 carefully selected vectors were run through the simulated multiplier. The simulation run takes about 10 hours of compute time,

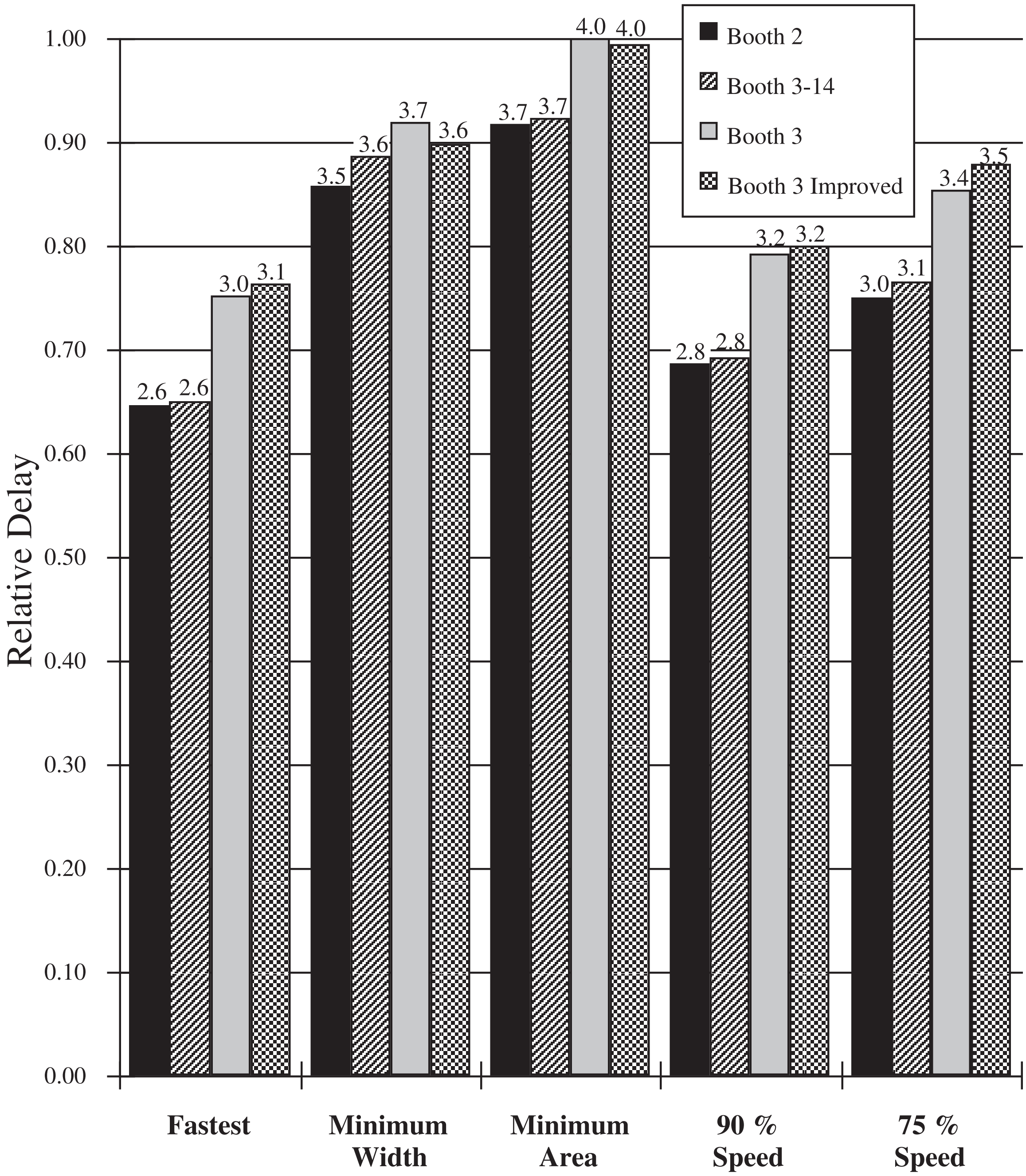


Figure 5.17: Delay comparison of multiplication algorithms. Delays are in nsecs.

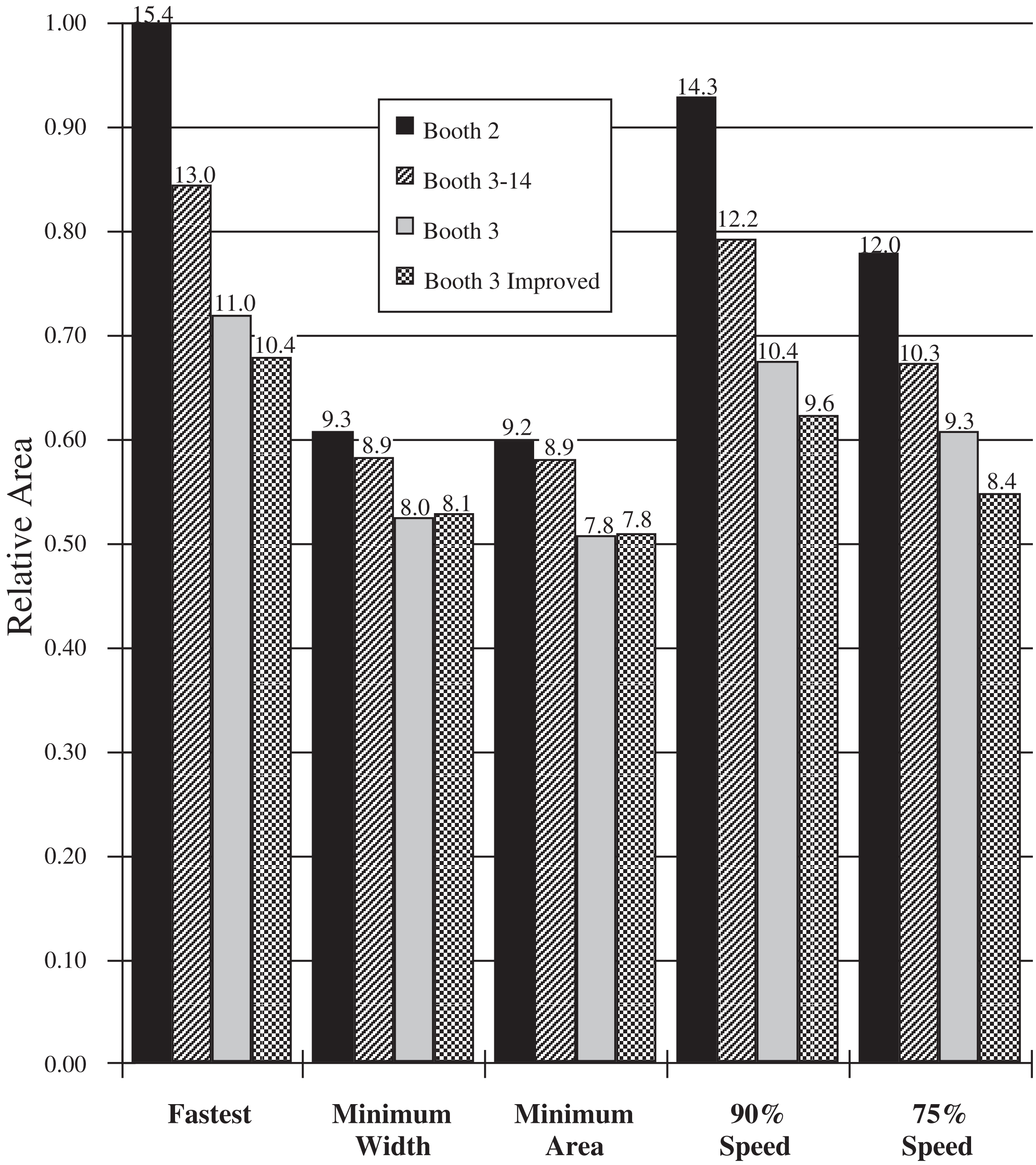


Figure 5.18: Area comparison of multiplication algorithms. Areas are in mm^2.

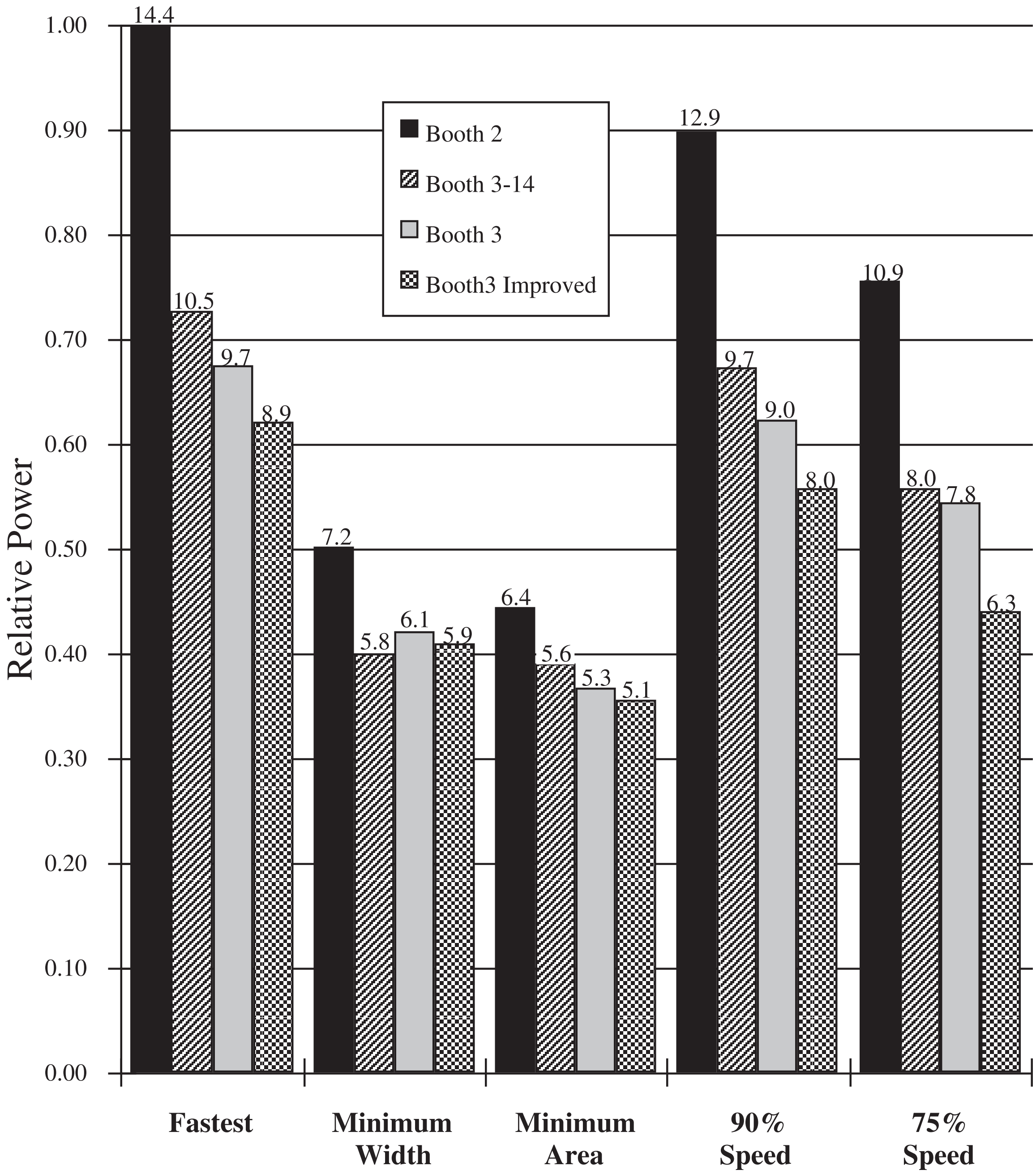


Figure 5.19: Power comparison of multiplication algorithms. Power is in Watts.

since the final multiplier has about 60,000 bipolar transistors.

The design goal for the multiplier is a propagation delay of 3.5 nsec (typical, 4.4 nsec worst case) at a power consumption of 4.5 Watts with a single 5V supply. The delay goal is dictated by limitations in the power dissipation of the package. The final layout size of the multiplier is 4 mm by 2.5 mm (excluding test and I/O circuitry). The floor plan of the chip is shown in Figure 5.20, and a photograph of the chip is shown in Figure 5.21.

5.5.1 Fabrication Results

After the design was taped out (December, 1992), a bug was found in the power ramping section of the summation network layout tool. This bug caused all output drivers in the summation network to be ramped down in power, even those drivers that drove wires along critical paths. Because the transistor level circuit simulator failed to provide any kind of timing information, this error was allowed to propagate to the final design. After reexamining the critical paths, it was determined that the multiplier would be slower and consume less power than expected (5 nsec delay and about 3.5 Watts).

Unfortunately the project driving the development of the fabrication line was cancelled before the yield problems of the fab were solved. Three wafers were obtained, in May 1993, and some parts on one wafer showed some functionality, but no completely working multipliers were obtained.

5.6 Comparison with Other Implementations

Comparisons with other implementations described in the literature is informative, because it provides reference points in evaluating a particular design. However such comparisons are less than straightforward, due to the widely varying technologies available. For example, there are no full 53x53, ECL multipliers described in the literature, so comparable, rather than identical, designs must be used instead.

Table 5.8 summarizes the important parameters for the multiplier design described here, and also for comparable designs described by Adiletta *et al.*[1], Mori *et al.* [19], Goto *et al.*[11], and Elkind *et al.*[8]. The table shows that the ECL based design described here is

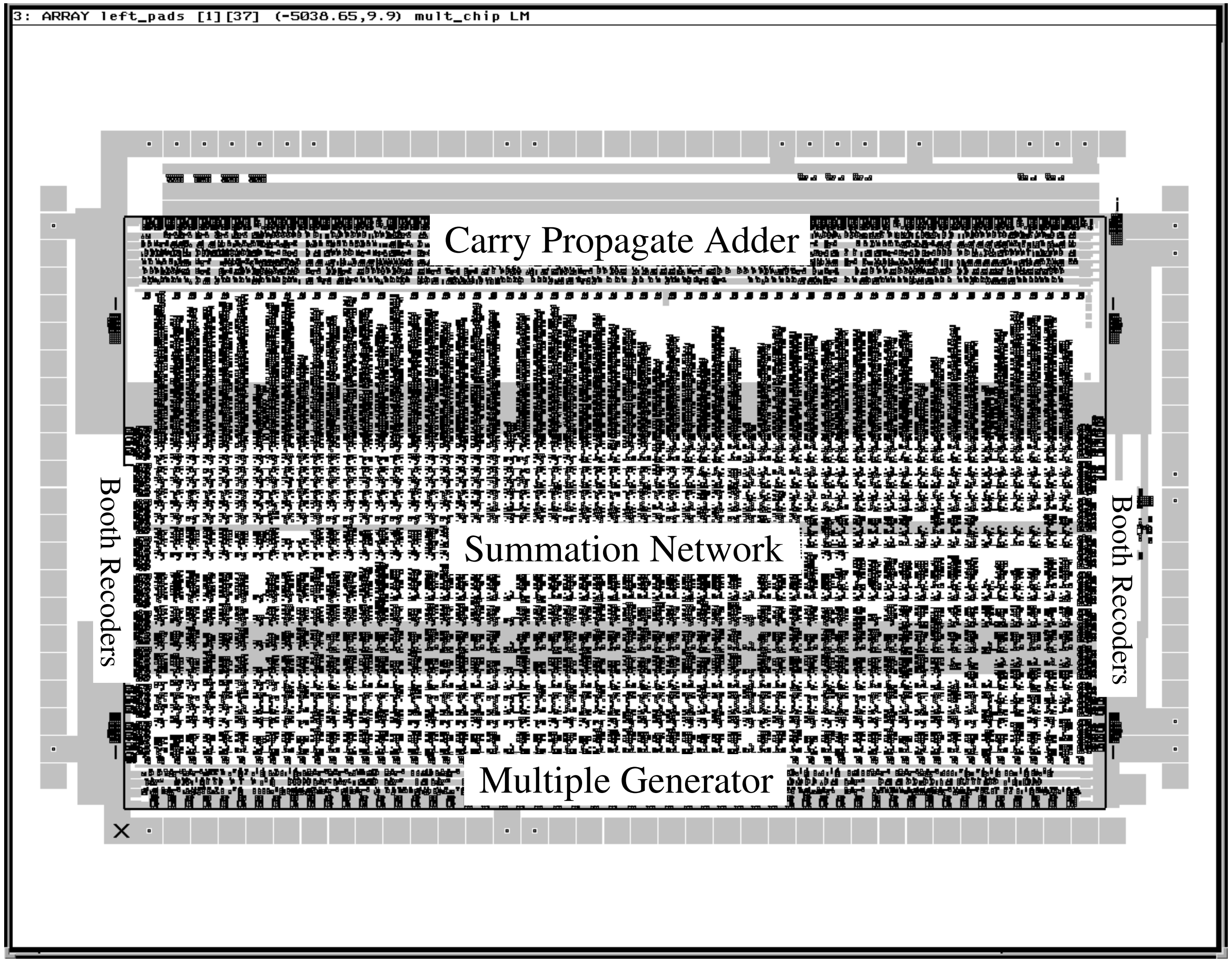


Figure 5.20: Floor plan of multiplier chip

Figure 5.21: Photo of 53x53 multiplier chip. Die size is 5mm x 3mm.

	This thesis	Adiletta	Mori	Goto	Elkind
Technology	ECL	ECL	CMOS	CMOS	ECL
Multiplier Size	53x53	32x32	54x54	54x54	56x56
Design Style	Custom	Gate array	Custom	Custom	Custom
Type	Full Tree	Full Tree	Full Tree	Full Tree	Iterative
Lithography	0.6μ	2μ	0.5μ	0.8μ	2μ
Area	$10mm^2$	$96mm^2$	$12.5mm^2$	$12.9mm^2$	$13.7mm^2$
Delay	3.5 nsec	9 nsec	10 nsec	13 nsec @40MHz	22.4 nsec 14 clocks @600MHz
Power	4.5 W	30.9 W	0.87 W	0.875 W	Not Stated
Power x Delay	15.7	278	8.7	11.4	

Table 5.8: Multiplier Designs

faster than comparable CMOS designs and also competitive in area. ECL designs consume high power, so careful circuit design is necessary to minimize the power consumption. With such care, the power-delay product of ECL designs can be less than a factor of two larger than CMOS designs.

5.7 Improvements

There are a couple of simple improvements that could be made to the multiplier designs presented in this chapter. First, because of limitations in the tools, only a 2 layer channel router was available. A good 3 layer router would have reduced the layout area of all the multipliers by 10-20%. Second, the power consumption could be reduced by the addition of a second power supply, with a voltage of -2.5. Many output emitter followers could be terminated to this supply rather than the -5V supply, reducing the power consumption in these drivers by 50%. In general about $\frac{1}{3}$ of the current in the designs could be terminated to this reduced voltage. This would reduce the total power consumption of all of the designs by about 15%.

5.8 Delay and Wires

Finally, in order to explore the effects of wires and asymmetric input delays in multiplier design, a summation network for a special Booth 3-14 multiplier was produced by the layout tool. All power ramping and the use of differential wiring was turned off in order to preserve the critical paths through the summation network and make all the CSAs in the summation network identical (except for any needed level shifting).

The critical path, through the resulting summation network, had a delay of 2.0 nsec and went through 9 levels of CSAs. This compares to 8 levels of CSAs if straight Wallace tree wiring is used. Of the total delay, 1.25 nsecs is used just for driving the wires between CSAs, leaving only 750 psec explicitly contributed by the intrinsic delays of the CSAs. Even though the wire component is large, counting levels of CSAs might still be useful in estimating performance if all of the wires were about the same length. Then the average wire delay (in this case 125 psec) could just be added to the CSA delay, becoming part of the delay associated with the "gate delay" of the CSA. In actuality, the wire delays along the critical path varied wildly, from 7 psec to 331 psec, with a standard deviation of 119 psec. This would seem to indicate that the efforts of the layout tool to take into consideration wire length when it places CSAs and multiplexers, is valuable in improving the performance of the summation network.

Another indication that considering wire and asymmetric input delays is important in improving the performance, can be seen from looking at other non-critical paths in the summation network. Some delay paths go through 11 CSA levels, and yet are faster than the critical path of 9 CSA levels. This is not even the most extreme case. The 11 level path has a delay of 1.4 nsec, yet is faster than another path that goes through only 8 levels (in 1.85 nsec). The path through more CSA levels is faster because of wires and also because it goes through faster inputs on the CSAs. Wires and asymmetric input delays matter, and need to be taken into consideration as the summation network is designed.

5.9 Summary

Of the conventional partial product generation algorithms considered in this chapter, the Booth 3 algorithm is the most efficient in power and area, but is slower due to the need for an expensive carry propagate addition when computing "hard" multiples. The Booth 2 algorithm is the fastest, but is also quite power and area hungry. Other conventional algorithms, such as Booth 4 and simple multiplication, do not seem to be competitive with the first two.

Implementations using the redundant Booth 3 algorithm compare quite favorably with the conventional algorithms. The fastest version of this algorithm is as fast as the Booth 2 algorithm, but provides modest decreases in both power ($\sim$ 25%) and area ($\sim$ 15%). The redundant Booth algorithm also compares favorably with the conventional(nonredundant) Booth 3 in both area and power, but is faster. Serious consideration should be given to this class of algorithms.

Wires and input delay variations are important when designing summation networks, if the highest possible performance is desired. Ignoring these effects can lead to designs that are not as fast as they could be.

Chapter 6

Conclusions

The primary objective of this thesis has been to present a new type of partial product generation algorithm (Redundant Booth), to reduce the implementation to practice, and to show through simulation and design that this algorithm is competitive with other more commonly used algorithms when used for high performance implementations. Modest improvements in area (about 15%) and power (about 25%) over more conventional algorithms have been shown using this algorithm. Although no performance increment has been demonstrated, this is not terribly surprising given the logarithmic nature of the summation network which adds the partial products.

Secondarily, this thesis has shown that algorithms based upon the Booth partial product method are distinctly superior in power and area when compared to non-Booth encoded methods. This result must be used carefully if applied to other technologies, since different trade-offs may apply. Partial product methods higher than Booth 3 do not seem to be worthwhile, since the savings due to the reduction of the partial products do not seem to justify the extra hardware required for the generation and distribution of the "hard multiples". This conclusion may not apply for multiplication lengths larger than 64 bits. The reason for this is that the "hard multiple" logic increases linearly with the multiplication length, but the summation network hardware increases with the square of the multiplication length.

The use of carry save adders in a Wallace tree (or similar configuration) is so very fast at summing partial products, that it seems that there is very little performance to be gained by trying to optimize the architecture of this component further. The delay effects of the other pieces of a multiplier are at least as important as the summation network in determining the final performance. Figure 6.1 shows the breakdown of delays of a multiplier using the redundant Booth 3 algorithm, with 14 bit small adders. The summation network is less than

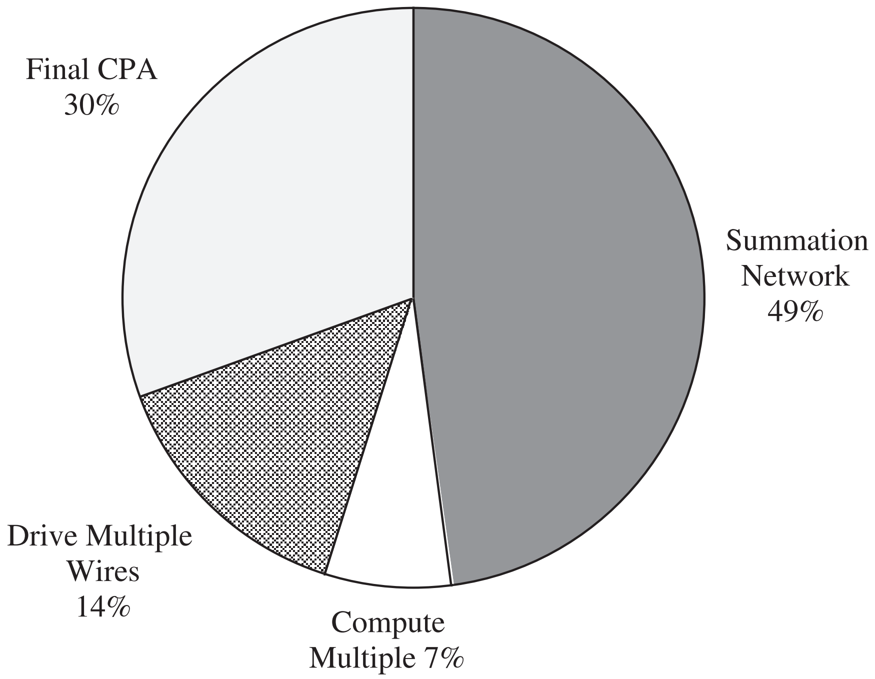


Figure 6.1: Delay components of Booth 3-14 multiplier.

$\frac{1}{2}$ of the total delay. This reduces the performance sensitivity of the entire multiplier to small changes in the summation network delay. As a result, somewhat slower, but more compact structures (such as linear arrays or hybrid tree/array structures) may be competitive with the faster tree approaches. Significant improvements in multiplier performance will come only from using faster circuits, or by using a completely different approach.

The summation network and partial product generation logic consume most of the power and area of a multiplier, so there may be more opportunities for improving multipliers by optimizing summation networks to try to minimize these factors. Reducing the number of partial products and creating efficient ways of driving the long wires needed in controlling and providing multiples to the partial product generators are areas where further work may prove fruitful.

Since wire delays are a substantial fraction of the total delay in both the summation network and the carry propagate adder, efforts to minimize the area may also improve the performance. Configuring the CSAs in a linear array arrangement is smaller and has shorter wires than tree configurations. In the future, if wires become relatively more expensive, such linear arrays may become competitive with tree approaches. At the present time trees still seem to be faster.

Finally, good low level circuit design seems to be very important in producing good multiplier designs. A modest improvement in the design of CSAs is important, because so many of them are required. From 900 to 2500 carry save adders were used in the designs presented in this thesis. This thesis has presented a power efficient circuit which can be used to drive a group of long wires, when it is known that exactly 1 of the wires can be high at any time. This single circuit reduces the power of the entire multiplier by about 8%, which seems modest, but it is only a single circuit. Another example where concentrating on the circuits can pay off is in power ramping non-critical paths, which saves about 30% of the power at virtually no performance cost. These examples illustrate that good circuit design, as well as good architectural decisions, are necessary if the best performing multipliers are to be built.

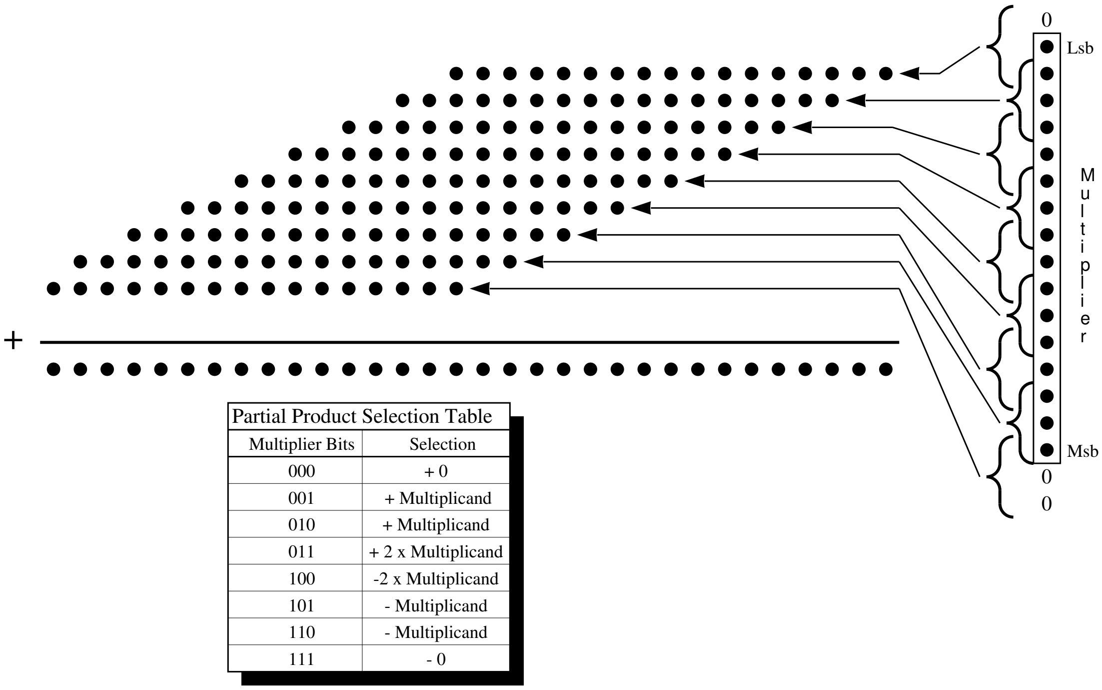
Appendix A

Sign Extension in Booth Multipliers

This appendix shows the sign extension constants that are needed when using Booth's multiplication algorithm are computed. The method will be illustrated for the 16x16 bit Booth 2 multiplication example given in Chapter 2. Once the basic technique is understood it is easily adapted to the higher Booth algorithms and also to the redundant Booth method of partial product generation. The example will be that of an unsigned multiplication, but the final section of this appendix will discuss the modifications that are required for signed arithmetic.

A.1 Sign Extension for Unsigned Multiplication

The partial products for the 16x16 multiply example, assuming that all partial products are positive, are shown in Figure A.1. Each partial product, except for the bottom one, is 17 bits long, since numbers as large as 2 times the multiplicand must be dealt with. The bottom partial product is 16 bits long, since the multiplier must be padded with 2 zeroes to guarantee a positive result. Figure A.2 shows the partial products if they all happen to be negative. Using 2's complement representation, every bit of the negated partial products is complemented, including any leading zeroes, and 1 is added at the least significant bit. The bottom partial product is never negated, because the 0 padding assures that it is always positive. The triangle of 1's on the left hand side can be summed to produce Figure A.3, which is exactly equivalent to the situation shown in Figure A.2. Now, suppose that a



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 A.1: 16 bit Booth 2 multiplication with positive partial products.

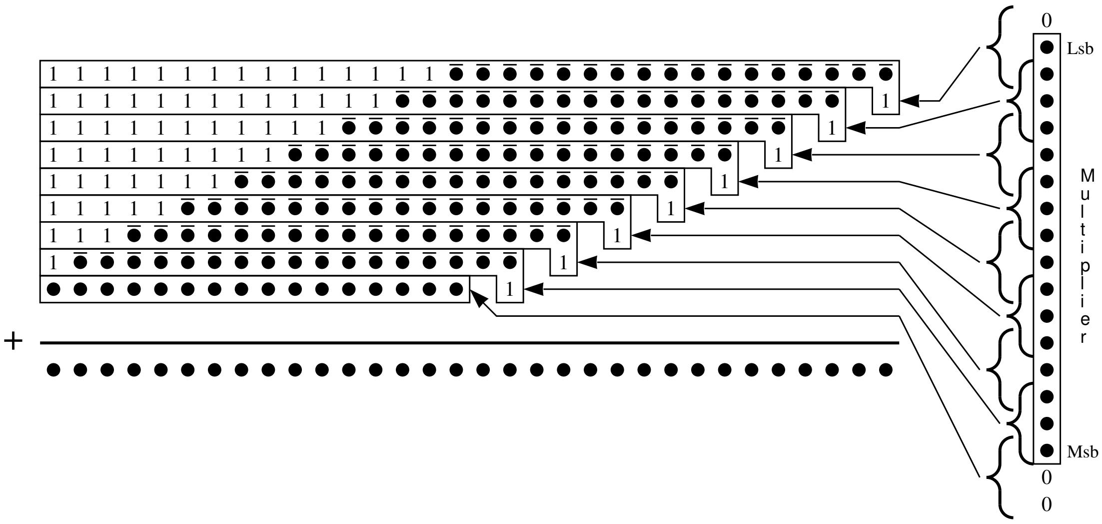


Figure A.2: 16 bit Booth 2 multiplication with negative partial products.

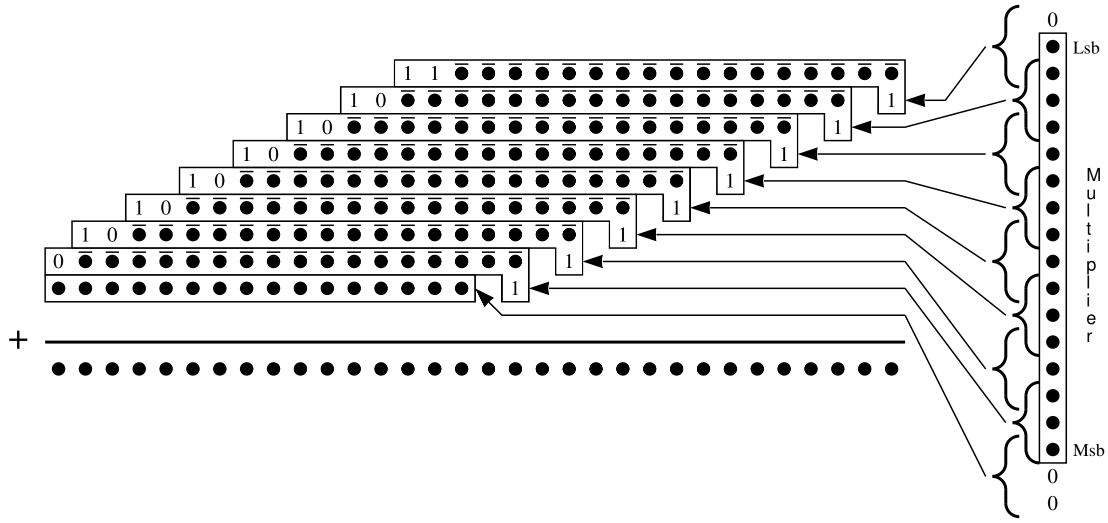
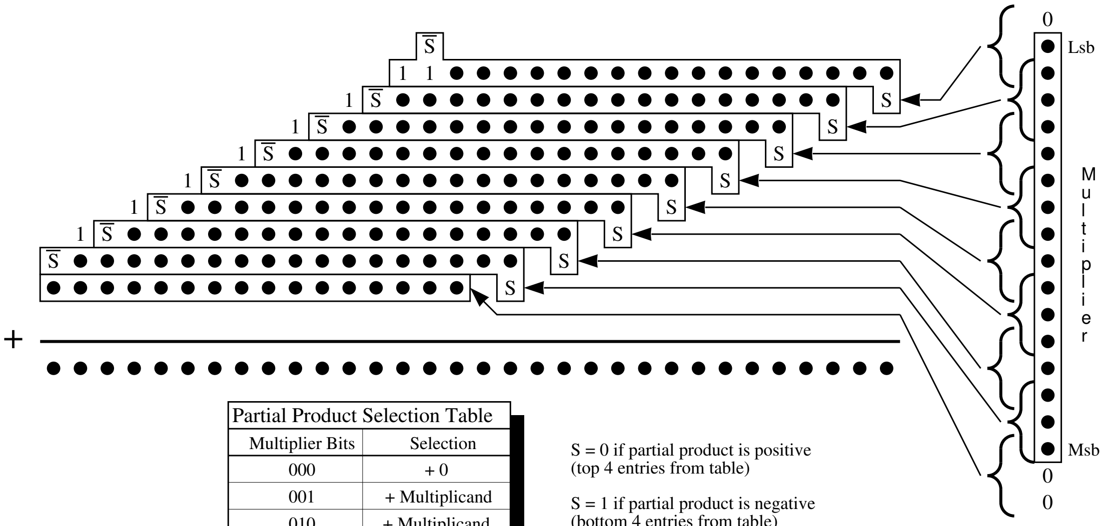


Figure A.3: Negative partial products with summed sign extension.

particular partial product turns out to not be negative. The leading string of ones in that particular partial product can be converted back to a leading of zeroes, by adding a single 1 at the least significant bit of the string. Referring back to the selection table shown in Figure A.1, a partial product is positive only if the most significant bit of the select bits for that partial product is 0. Additionally, a 1 is added into the least significant bit of a partial product only if it is negative. Figure A.4 illustrates this configuration. The $\overline{\mathrm{S}}$ bits represent the 1's that are needed to clear the sign extension bits for positive partial products, and the S bits represent the 1's that are added at the least significant bit of each partial product if it is negative.

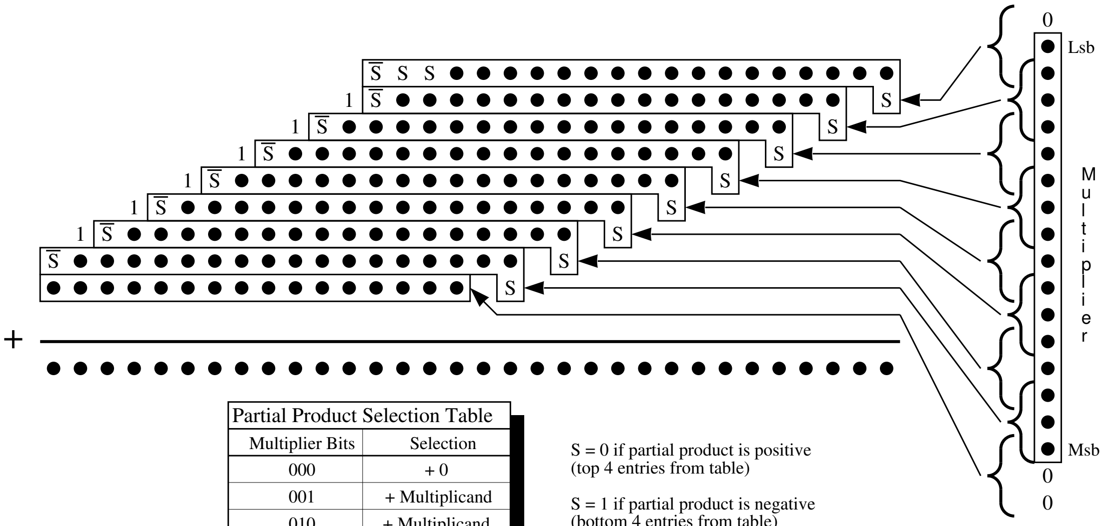
A.1.1 Reducing the Height

Finally, the height (maximum number of items to be added in any one column) of the dot diagram in Figure A.4 can be reduced by one by combining the $\overline{\mathrm{S}}$ term of the top partial product with the two leading ones of the same top partial product, which gives the final result, shown in Figure A.5 (this is the same as Figure 2.4).



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 A.4: Complete 16 bit Booth 2 multiplication.



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

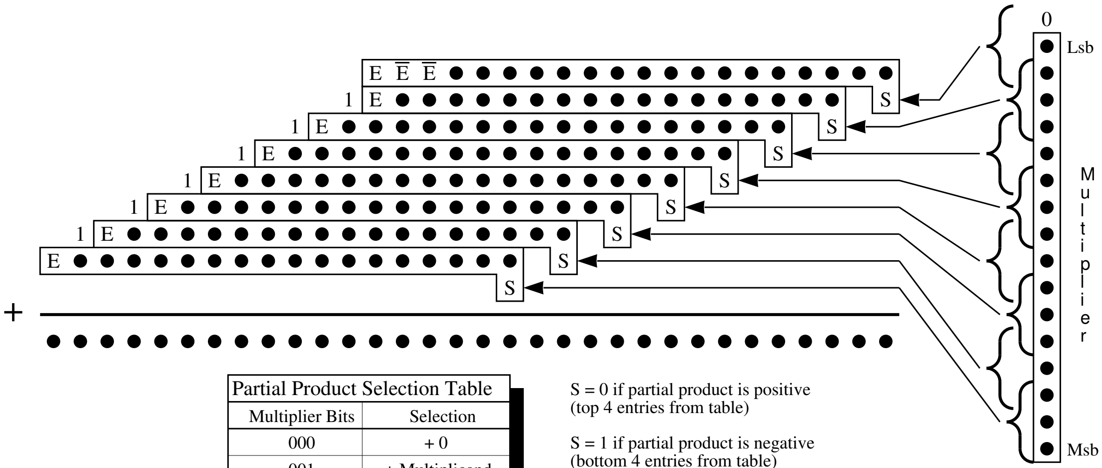
Figure A.5: Complete 16 bit Booth 2 multiplication with height reduction.

A.2 Signed Multiplication

The following modifications are necessary for 2's complement, signed multiplication.

- The most significant partial product (shown at the bottom in all of the preceding figures), which is necessary to guarantee a positive result, is not needed for signed multiplication. All that is required is to sign extend the multiplier to fill out the bits used in selecting the most significant partial product. For the sample 16x16 multiplier, this means that one partial product can be eliminated.
- When $\pm$Multiplicand (entries 1,2,5 and 6 from the partial product selection table) is selected, the 17 bit section of the effected partial product is filled with a sign extended copy of the multiplicand. This sign extension occurs before any complementing that is necessary to obtain $-$Multiplicand.
- The leading 1 strings, created by assuming that all partial products were negative, are cleared in each partial product under a slightly different condition. The leading 1's for a particular partial product are cleared when that partial product is positive. For signed multiplication this occurs when the multiplicand is positive and the multiplier select bits chooses a positive multiple, and also when the multiplicand is negative and the multiplier select bits choose a negative multiple. A simple EXCLUSIVE-NOR between the sign bit of the multiplicand and the high order bit of the partial product selection bits in the multiplier generates the one to be added to clear the leading 1's correctly.

The complete 16x16 signed multiplier dot diagram is shown in Figure A.6



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

S = 0 if partial product is positive (top 4 entries from table)

S = 1 if partial product is negative (bottom 4 entries from table)

E = 1 if multiplicand is positive and partial product is positive, or if multiplicand is negative and partial product is negative or if partial product is +0.

E = 0 if multiplicand is positive and partial product is negative, or if multiplicand is negative and partial product is positive or if partial product is -0.

Figure A.6: Complete signed 16 bit Booth 2 multiplication.

Appendix B

Efficient Sticky Bit Computation

B.1 Rounding

The material in the preceding chapters of this thesis have dealt with methods and algorithms for implementing integer multiplication. Chapter 5 briefly explained the format of IEEE double precision floating point numbers. To convert an integer multiplier into a floating point multiplier requires 2 modifications to the multiplication hardware :

- Exponent adder - This involves a short length (12 bits or less) adder.
- Rounding logic - The rounding logic accepts the 106 bit integer product and uses the low order 53 bits of the product to slightly modify the high order 53 bits of the product, which then becomes the final 53 bit fraction portion of the product.

The actual rounding process is quite involved, and methods for high speed rounding can be found in Santoro, Bewick, and Horowitz[23]. The purpose of this appendix is to discuss an efficient method for computing the "sticky bit" which is required for correct implementation of the IEEE round to nearest rounding mode, and is also required for computation of the IEEE "exact" status signal.

B.2 What's a Sticky Bit?

The sticky bit is a status bit that is derived from the low order bits of the final 106 bit product. The sticky bit is high if every bit of the low order bits of the 106 bit final product is zero. The actual number of bits involved in the sticky computation can depend on the rounding mode, and also can depend on whether the high order bit of the 106 bit product is a "1". Whatever the precise definition, the value of the sticky bit is based upon some number of low order product bits that need to be tested to determine if they are all zero. For IEEE double precision numbers, the number of low order bits that need to be tested is in the vicinity of 50 or so. The sticky bit, then, reduces to a large length zero detect.

B.3 Ways of Computing the Sticky

The obvious method of computing the sticky bit is with a large fan-in OR gate on the low order bits of the product. This method has the disadvantage that the sticky bit takes a long time to compute because the low order bits of the product must be available and then must propagate through a large fan-in OR gate. Because the largest practical OR gate that can be built with commonly available technology is limited to 4 or 5 inputs, the sticky bit must be created through a number of smaller length OR gates wired in series. This places the sticky bit squarely in the critical path of the multiplier. High speed implementations compute the final product using a 106 bit carry propagate adder, but the low order bits of the result affect high order bits of the result only through carries propagated into the high order bits, and the value of the sticky bit. It seems very inefficient to actually compute the low order product bits, determine the sticky, and then throw the low order product bits away.

To remove the sticky bit from the critical path of the multiplier, different methods may be used. One method involves determining the number of trailing zeros in the two input operands to the multiplier. It can be proven that the number of trailing zeros in the product is exactly equal to the sum of the number of trailing zeros in each input operand. The sticky bit can then be determined by checking to see if the trailing zero sum is greater than or equal to the number of low order bits involved in the sticky computation. If true, then the sticky bit will be a 1, otherwise it will be 0. Notice that this method doesn't require the actual low order product bits, just the input operands, so the determination can occur in parallel with the actual multiply operation, removing the sticky computation from the critical path. The disadvantage of this method is that significant extra hardware is required. This hardware includes 2 long length priority encoders to count the number of trailing zeros in the input operands, a small length adder, and a small length comparator. Some hardware is eliminated, though, in that the actual low order bits of the product are no longer needed, so part of the carry propagate adder hardware can be eliminated.

The Santoro rounding paper describes a very clever method of sticky computation which involves examining the low order bits of the product while it is still in a redundant form, i.e. before the carry propagate add which computes the final product. If all of the low order bits of the redundant form are zero, then the sticky bit must be 1, else it must be 0. This overlaps the sticky computation with the final carry propagate add which computes the product, removing the sticky from the critical path. Unfortunately, this method only works for non-Booth encoded multipliers, a significant disadvantage given the results of Chapter 5. Like the previous method, this scheme also avoids the actual computation of the low order bits, which provides hardware savings in the final carry propagate adder.

B.4 An Improved Method

The improved sticky method described in this appendix was inspired by the Santoro sticky scheme. While not quite as efficient as the Santoro method, the improved method also works for Booth encoded multipliers. A small amount of extra hardware is required, however the extra hardware requirement is significantly reduced if this method is used in conjunction with the redundant Booth multiplication algorithm described in Chapter 2. Like the previous methods, direct computation of the low order product bits is avoided, providing hardware savings in the carry propagate adder.

The idea is to inject the constant -1 into the multiplier summation network which is responsible for summing the partial products. That is the summation network computes the value (operand 1) $\times$ (operand 2) - 1, instead of (operand 1) $\times$ (operand 2). The proper result is then obtained by modifying the final carry propagate add so that it computes A+B+1 instead of A+B. This is easily accomplished by just forcing the carry-in to the adder to 1.

It will be shown that a group of low order result bits will be all zeros if and only if :

- The carry-in of 1 is propagated from the lowest order bit across all low order bits involved in the sticky bit computation AND
- A carry is not generated anywhere by the low order bits involved in the sticky computation.

Using the terminology of Chapter 3, these two conditions can be stated in a more precise manner. Assume that the number of low order bits involved in the sticky computation is n. Then the sticky bit, ST_n is :

$$ST_n = \overline{s_{n-1}}\,\overline{s_{n-2}}\ \ldots\ \overline{s_1}\,\overline{s_0} \tag{B.1}$$

Where $s_{n-1} \ldots s_0$ are the low order product bits. The conjecture is that :

$$ST_n = p_0^{n-1}\overline{g_0^{n-1}} \tag{B.2}$$

Proof: By induction n, the number of bits involved in the sticky computation. For n=1, Equation 3.1 gives s_0, and with a carry-in of 1, s_0 becomes :

$$\begin{aligned} s_0 &= a_0 \oplus b_0 \oplus 1 \\ &= \overline{a_0 \oplus b_0} \end{aligned}$$

From Equation B.1 the sticky bit, ST_0, is :

$$\begin{aligned} ST_0 &= \overline{s_0} \\ &= a_0 \oplus b_0 \end{aligned} \tag{B.3}$$

This is the same as Equation 3.6 which defines ${p^{\oplus}}_0$. To allow the use for general p_0, that is where p_0 is computed as either ${p^{\oplus}}_0$ (EXCLUSIVE-OR) or p_0^{+} (OR), p_0 must be ANDed with the inversion of g_0 :

$$\begin{aligned} ST_0 &= a_0 \oplus b_0 \\ &= p_0\overline{g_0} \end{aligned}$$

This establishes the result for n=1. To prove for n bits, the result is assumed to be true for n-1 bits, and then shown to be true for n bits. From Equation B.1 :

$$ST_n = \overline{s_{n-1}}\, ST_{n-1} \tag{B.4}$$

From Equation 3.1:

$$\begin{aligned} s_{n-1} &= a_{n-1} \oplus b_{n-1} \oplus c_{n-1} \\ s_{n-1} &= (p_{n-1}\overline{g_{n-1}}) \oplus c_{n-1} \end{aligned} \tag{B.5}$$

Where c_{n-1} is the carry-in to bit n-1. Equations 3.7, 3.8, and 3.9 give c_{n-1}, and since $c_0 = 1$, c_{n-1} can be written as

$$c_{n-1} = g_0^{n-2} + p_0^{n-2} \tag{B.6}$$

Substituting Equations B.5 and B.6 into Equation B.4 gives :

$$ST_n = \overline{[(p_{n-1}\overline{g_{n-1}}) \oplus (g_0^{n-2} + p_0^{n-2})]}\, ST_{n-1}$$

The induction hypothesis is :

$$ST_{n-1} = p_0^{n-2}\overline{g_0^{n-2}}$$

When this is substituted for ST_{n-1} it gives :

$$\begin{aligned} ST_n &= \overline{[(p_{n-1}\overline{g_{n-1}}) \oplus (g_0^{n-2} + p_0^{n-2})]}\, (p_0^{n-2}\overline{g_0^{n-2}}) \\ &= [p_{n-1}\overline{g_{n-1}}(g_0^{n-2} + p_0^{n-2}) + \overline{p_{n-1}\overline{g_{n-1}}}\;\overline{(g_0^{n-2} + p_0^{n-2})}]\, (p_0^{n-2}\overline{g_0^{n-2}}) \\ &= [p_0^{n-1}\overline{g_{n-1}} + (\overline{p_{n-1}} + g_{n-1})(\overline{g_0^{n-2}})(\overline{p_0^{n-2}})]\, (p_0^{n-2}\overline{g_0^{n-2}}) \\ &= (p_0^{n-1})(\overline{g_0^{n-2}})(\overline{g_{n-1}}) \\ &= (p_0^{n-1})(\overline{p_{n-1}} + \overline{g_0^{n-2}})(\overline{g_{n-1}}) \\ &= p_0^{n-1}\overline{g_0^{n-1}} \end{aligned}$$

Which completes the proof by induction.

The actual sticky bit computation can be simplified even further if the individual bit propagates,p_k are generated using the EXCLUSIVE-OR form, $p_k^{\oplus}$. Then the $\overline{g_0^{n-1}}$ term can be dropped, because p_0^{n-1} and g_0^{n-1} are mutually exclusive. If a carry is propagated across the entire group of low order bits, no carry can be generated in those bits. Then ST_n becomes :

$$ST_n = p_0^{n-1}$$

B.5 The -1 Constant

The -1 constant that is required to be added into the summation network is a full length (106 bit) string of ones, assuming two's complement representation. For Booth encoded multipliers, there is usually a constant that must be added for sign extension reasons, which is non-zero only beginning at about bit 53. Therefore 53 extra ones's must be added into the summation network. The hardware requirement for this is about the same as 53 half adders. Since the actual summation network consists of 1000 or more carry save adders, each of which is about twice as complex as a single half adder, the extra hardware can be seen to be quite small. The carry generate and carry propagate signals are already required to propagate carries from the low order product bits to the high order bits. Elimination of the summation hardware from the low order bits of the carry propagate adder saves about the same amount of hardware, so there is approximately no net change in the hardware requirements. However the sticky bit has been eliminated from the critical path.

This sticky method is particularly efficient when combined with the redundant Booth algorithm described in Chapter 2. This is because of the existence of the *Compensation Constant* which already must be added into the summation network. This compensation constant is guaranteed to have a 1 in a bit position that is about the length of the small adders used in the hard multiple generator. This is usually in the range of 8-14 bits. Therefore at most 8-14 extra half adders will be needed to add the -1 constant into the summation network. Eliminating the computation of the low order summation bits saves more hardware than this, giving a net reduction in the total hardware requirements.

Appendix C

Negative Logic Adders

The purpose of this appendix is to provide a proof of Theorem 1 of Chapter 3. Although there are other ways of proving this particular theorem, the proof illustrates the simple manner in which many of the relationships used in Chapter 3 can be proven using mathematical induction. It is also possible to prove the correctness of the algorithms presented in Chapter 2, by induction on the number of "digits" in the multiplier. A digit is a single group of bits from the multiplier which are recoded using Booth's algorithm and then used to select a particular partial product.

Theorem 1 *Let A and B be positive logic binary numbers, each n bits long, and* c_0 *be a single carry bit. Let S be the n bit sum of A, B, and* c_0*, and let* c_n *be the carry out from the summation. That is :*

$$2^n \cdot c_n \overset{\text{sum}}{+} S = A \overset{\text{sum}}{+} B \overset{\text{sum}}{+} c_0$$

Then :

$$2^n \cdot \overline{c_n} \overset{\text{sum}}{+} \overline{S} = \overline{A} \overset{\text{sum}}{+} \overline{B} \overset{\text{sum}}{+} \overline{c_0}$$

Proof: (by induction on n, the length of the operands)**:**

The case n=0 gives (using equations 3.1 and 3.2):

$$\begin{aligned}
2 \cdot \overline{c_1} \overset{\text{sum}}{+} \overline{s_0} &= 2 \cdot \overline{(a_0\, b_0 + a_0\, c_0 + b_0\, c_0)} \overset{\text{sum}}{+} \overline{(a_0 \oplus b_0 \oplus c_0)} \\
&= 2 \cdot \left[\left(\overline{a_0\, b_0}\right) \left(\overline{a_0\, c_0}\right) \left(\overline{b_0\, c_0}\right) \right] \overset{\text{sum}}{+} \left(\overline{a_0} \oplus \overline{b_0} \oplus \overline{c_0}\right)
\end{aligned}$$

$$
\begin{aligned}
&= 2 \cdot \left[\left(\overline{a_0} + \overline{b_0}\right) \left(\overline{a_0} + \overline{c_0}\right) \left(\overline{b_0} + \overline{c_0}\right)\right] \overset{\text{sum}}{+} \left(\overline{a_0} \oplus \overline{b_0} \oplus \overline{c_0}\right) \\
&= 2 \cdot \left[\overline{a_0}\,\overline{b_0} + \overline{a_0}\,\overline{c_0} + \overline{b_0}\,\overline{c_0}\right] \overset{\text{sum}}{+} \left(\overline{a_0} \oplus \overline{b_0} \oplus \overline{c_0}\right) \\
&= \overline{A} \overset{\text{sum}}{+} \overline{B} \overset{\text{sum}}{+} \overline{c_0}
\end{aligned}
$$

To prove by induction on n (the length of the A and B operands), the theorem is assumed for all operands of length n-1, and with this assumption it is proven to be true for operands of length n. Proceeding:

$$
\begin{aligned}
2^n \cdot \overline{c_n} \overset{\text{sum}}{+} \overline{S} &= 2^n \cdot \left[\overline{(a_{n-1}\, b_{n-1} + a_{n-1}\, c_{n-1} + b_{n-1}\, c_{n-1})}\right] \overset{\text{sum}}{+} \sum_{k=0}^{n-1} \overline{s_k} \cdot 2^k \\
&= 2^n \cdot \left[\overline{a_{n-1}}\,\overline{b_{n-1}} + \overline{a_{n-1}}\,\overline{c_{n-1}} + \overline{b_{n-1}}\,\overline{c_{n-1}}\right] \overset{\text{sum}}{+} 2^{n-1} \cdot \overline{s_{n-1}} \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{s_k} \cdot 2^k \\
&= 2^n \cdot \left[\overline{a_{n-1}}\,\overline{b_{n-1}} + \overline{a_{n-1}}\,\overline{c_{n-1}} + \overline{b_{n-1}}\,\overline{c_{n-1}}\right] \\
&\quad \overset{\text{sum}}{+} 2^{n-1} \cdot \overline{a_{n-1} \oplus b_{n-1} \oplus c_{n-1}} \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{s_k} \cdot 2^k \\
&= 2^n \cdot \left[\overline{a_{n-1}}\,\overline{b_{n-1}} + \overline{a_{n-1}}\,\overline{c_{n-1}} + \overline{b_{n-1}}\,\overline{c_{n-1}}\right] \\
&\quad \overset{\text{sum}}{+} 2^{n-1} \cdot \left(\overline{a_{n-1}} \oplus \overline{b_{n-1}} \oplus \overline{c_{n-1}}\right) \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{s_k} \cdot 2^k \\
&= 2^{n-1} \left(\overline{a_{n-1}} \overset{\text{sum}}{+} \overline{b_{n-1}} \overset{\text{sum}}{+} \overline{c_{n-1}}\right) \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{s_k} \cdot 2^k \\
&= 2^{n-1} \left(\overline{a_{n-1}} \overset{\text{sum}}{+} \overline{b_{n-1}}\right) \overset{\text{sum}}{+} 2^{n-1}\overline{c_{n-1}} \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{s_k} \cdot 2^k
\end{aligned}
$$

Now use the induction hypothesis to replace the last two terms by the sums of the first n-1 bits of $\overline{A}$, $\overline{B}$ and a single bit carry-in $\overline{c_0}$:

$$
\begin{aligned}
2^n \cdot \overline{c_n} \overset{\text{sum}}{+} \overline{S} &= 2^{n-1} \left(\overline{a_{n-1}} \overset{\text{sum}}{+} \overline{b_{n-1}}\right) \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{a_k} \cdot 2^k \overset{\text{sum}}{+} \sum_{k=0}^{n-2} \overline{b_k} \cdot 2^k \overset{\text{sum}}{+} \overline{c_0} \\
&= \sum_{k=0}^{n-1} \overline{a_k} \cdot 2^k \overset{\text{sum}}{+} \sum_{k=0}^{n-1} \overline{b_k} \cdot 2^k \overset{\text{sum}}{+} \overline{c_0} \\
&= \overline{A} \overset{\text{sum}}{+} \overline{B} \overset{\text{sum}}{+} \overline{c_0}
\end{aligned}
$$

which completes the inductive proof.

Bibliography

[1] Matthew J. Adiletta, Richard L. Doucette, John H. Hackenberg, Dale H. Leuthold, and Dennis M. Litwinetz. Semiconductor Technology in a High-Performance VAX System. *Digital Technical Journal*, 2(4):43–60, Fall 1990.

[2] A. R. Alvarez, editor. *BiCMOS Technology and Applications*. Kluwer Academic Publishers, 1989.

[3] G. Bewick, P. Song, G De Micheli, and M. J. Flynn. Approaching a Nanosecond : A 32 Bit Adder. In *Proceedings of the 1988 IEEE International Conference on Computer Design*, pages 221–226, 1988.

[4] Gary Bewick and Michael J. Flynn. Binary Multiplication Using Partially Redundant Multiples. Technical Report CSL-TR-92-528, Stanford University, June 1992.

[5] A. D. Booth. A Signed Binary Multiplication Technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, June 1951.

[6] C. T. Chuang. NTL with Complementary Emitter-Follower Driver : A High-Speed Low-Power Push-Pull Logic Circuit. In *1990 IEEE Symposium on VLSI Circuits*, pages 93–94. IBM Research Division, Thomas J. Watson Research Center, 1990.

[7] L. Dadda. Some Schemes for Parallel Multipliers. *Alta Frequenza*, 36(5):349–356, May 1965.

[8] Bob Elkind, Jay Lessert, James Peterson, and Gregory Taylor. A sub 10ns Bipolar 64 Bit Integer/Floating Point Processor Implemented on Two Circuits. In *IEEE 1987 Bipolar Circuits and Technology Meeting*, pages 101–104, 1987.

[9] Mohamed I. Elmasry. *Digital Bipolar Integrated Circuits*. John Wiley & Sons, 1983.

[10] M. J. Flynn, G. De Micheli, R. Dutton, R. F. Pease, and B. Wooley. Subnanosecond Arithmetic : Second Report. Technical Report CSL-TR-91-481, Stanford University, June 1991.

[11] G. Goto, T. Sato, M. Nakajima, and T. Sukemura. A 54*54-b Regularly Structured Tree Multiplier. *IEEE Journal of Solid-State Circuits*, 27(9):1229–1236, September 1992.

[12] IEEE Standard for Binary Floating-Point Arithmetic, 1985. ANSI/IEEE Std 754-1985.

[13] Norman P. Jouppi. MultiTitan Floating Point Unit. In *MultiTitan: Four Architecture Papers*. Digital Western Research Laboratory, April 1988.

[14] Earl E. Swartzlander Jr., editor. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, 1990.

[15] J. Kirchgessner, J. Teplik, V. Ilderem, D. Morgan, R. Parmar, S. R. Wilson, J. Freeman, C. Tracy, and S. Cosentino. An Advanced 0.4μ BiCMOS Technology for High Performance ASIC Applications. In *International Electron Devices Meeting Technical Digest*, pages 4.4.1–4.4.4, 1991.

[16] H. Ling. High-Speed Binary Adder. *IBM Journal of Research and Development*, 25(2 and 3):156–166, May 1981.

[17] O. L. MacSorley. High-Speed Arithmetic in Binary Computers. *Proceedings of the IRE*, 49(1):67–91, Jan 1961.

[18] Meta-Software. *HSPICE User's Manual - H9001*. Meta-Software Inc., 1990.

[19] J. Mori, M. Nagamatsu, M. Hirano, S. Tanaka, M. Noda, Y. Toyoshima, K. Hashimoto, H. Hayashida, and K. Maeguchi. A 10ns 54x54-bit Parallel Structured Full Array Multiplier with 0.5μm CMOS Technology. In *1990 Symposium on VLSI Circuits*, pages 125–126, 1990.

[20] Motorola. *MECL System Design Handbook*. Motorola Semiconductor Products Inc., 1988.

[21] Michael S. Paterson and Uri Zwick. Shallow Multiplication Circuits. In *10th Symposium on Computer Arithmetic*, pages 28–34, 1991.

[22] Marc Rocchi, editor. *High Speed Digital IC Technologies*. Artech House, 1990.

[23] M. R. Santoro, G. Bewick, and M. A. Horowitz. Rounding Algorithms for IEEE Multipliers. In *Proceedings of 9th Symposium on Computer Arithmetic*, pages 176–183, 1989.

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

[25] Mark Santoro and Mark Horowitz. SPIM: A Pipelined 64x64b Iterative Array Multiplier. *IEEE International Solid State Circuits Conference*, pages 35–36, February 1988.

[26] N. R. Scott. *Computer Number Systems & Arithmetic*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[27] C. E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. Am. Inst. Electr. Eng.*, 57:713–723, 1938.

[28] C. E. Shannon. The Synthesis of Two-Terminal Switching Circuits. *Bell Syst. Tech. J.*, 28(1), 1949.

[29] J. Sklansky. Conditional Sum Addition Logic. *Transactions of the IRE*, EC-9(2):226–230, June 1960.

[30] 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.

[31] Jeffery Y.F. Tang and J. Leon Yang. Noise Issues in the ECL Circuit Family. Technical report, Digital Western Research Laboratory, January 1990.

[32] Stamatis Vassiliadis. Six-Stage 64-Bit Adder. *IBM Technical Disclosure Bulletin*, 30(6):208–212, November 1987.

[33] Stamatis Vassiliadis. Adders With Removed Dependencies. *IBM Technical Disclosure Bulletin*, 30(10):426–429, March 1988.

[34] Stamatis Vassiliadis. A Comparison Between Adders with New Defined Carries and Traditional Schemes for Addition. *International Journal of Electronics*, 64(4):617–626, 1988.

[35] C. S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, EC-13:14–17, February 1964.

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

[37] A. Weinberger. 4-2 Carry-Save Adder Module. *IBM Technical Disclosure Bulletin*, 23(8):3811–3814, January 1981.

[38] A. Weinberger and J. L. Smith. A One-Microsecond Adder Using One-Megacycle Circuitry. *IRE Transactions on Electronic Computers*, EC-5:65–73, June 1956.

[39] S. Winograd. On the Time Required to Perform Addition. *Journal of the ACM*, 12(2):227–285, 1965.

[40] S. Winograd. On the Time Required to Perform Multiplication. *Journal of the ACM*, 14(4):793–802, 1967.