LARGE OPERAND DIVISION AND AN ASYNCHRONOUS APPROACH TO FAULT DETECTION

Thesis by

Kathleen A. Kramer

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1991

(Submitted January 9, 1991)

To my father

Acknowledgements

First, I would like to express my gratitude to Professor Rodney M. Goodman for his generous support and guidance as my advisor.

Professor A. J. Martin classes taught me much that was useful in my work, for this and other things, I thank him. I also thank his student Tony Lee for helpful discussions in developing the divider.

I am extremely grateful to Masahiro Sayano for several days he donated to the write up of this work. I also thank R. Ramésh for his friendship and the hours he devoted to the writing of this work.

Silicon Compiler Systems generously provided the GDT software used in the development of the divider architecture; their support in this respect is gratefully acknowledged.

I owe a word of thanks to my officemate and dear friend John W. Miller who has helped me in my quest for a doctorate by serving as a good sounding board for my ideas and a willing audience for a loquacious and sometimes intolerable speaker.

I also thank Professors Edward C. Posner, Robert J. McEliece and Yaser S. Abu-Mostafa who graciously consented to be on my committee.

However, I owe my greatest debt to Dr. Anthony J. McAuley of Bell Communications Research for his support, guidance, and mentoring throughout my graduate studies. I also thank Bill Marcus for his work on the C program simulation of the divider. Dr. C. Cotton and Dr. D. Sincoskie of Bell Communications Research deserve a word of thanks for their understanding and support.

Abstract

Larger, faster ICs are creating a rash of new problems for the system designer. Designers faced with building larger and larger systems base their architectures on smaller systems that may scale poorly. As a result of VLSI, many new architectures are coming into favor, either because of the changing importance of design factors or because it is now possible to design bigger chips.

Efficient VLSI methods for implementing the basic arithmetic operations can push back many system-performance limitations. There is continued need for re-evaluation of arithmetic architectures, as the efficiency of implementation is related to both implementation technology and size of the operands. A new binary divider for n-bit integer operands, which produces the quotient and remainder in $O(n)$ time using $O(n)$ area, is presented. For very large operands, such as those required in Public Key Cryptography, the new divider is faster than comparable carry-save dividers and is more area-efficient than implementations using more redundant arithmetic.

A further problem faced by the designer of very large systems is their susceptibility to error. The system must be efficiently designed to function in the presence of errors, which become more likely as the size of the system increases. Qualities inherent in many asynchronous designs can be used to provide fault detection and therefore, fault tolerance. An approach to fault tolerance, one not possible with conventional, clocked, systolic arrays, is presented. This method of fault detection/correction exploits the inherent redundancy of architectures using four-state coding, a data-driven technique for implementing bit-level wave-front arrays.

TABLE OF CONTENTS

Chapter 1 Introduction

Larger, faster ICs are creating a rash of new problems for the system designer. Designers faced with building larger and larger systems base their architectures on smaller systems that may scale poorly. As a result of VLSI, many new architectures are coming into favor, either because of the changing importance of design factors or because it is now possible to design bigger chips. Efficient VLSI methods for implementing the four basic arithmetic operations have recently pushed back many system-performance limitations. There is continued need for re-evaluation of arithmetic architectures, since the efficiency of implementation can be divorced from neither the implementation technology nor the size of operand. However, when compared with improvements in performance of addition, subtraction, and multiplication, division has been relatively ignored [1].

A new binary divider for n-bit integer operands, which produces the quotient and remainder in $O(n)$ time using $O(n)$ area, is presented. For very large operands, such as those required in Public Key Cryptography, the new divider is faster than comparable carry-save dividers and is more area efficient than redundant arithmetic implementations.

Chapter 2 introduces the reader to the problem of division. Then, in Chapter 3, current division techniques are briefly reviewed. Chapter 4 presents a new divider, explaining how and why the algorithm works and comparing its performance to comparable dividers. Chapter 5 gives an implementation for the 16-bit version of the divider.

One further problem faced by the designer of very large systems is their susceptibility to error. The system must be efficiently designed to function in the presence of errors, which become more likely as the size of the system increases. Qualities inherent in many asynchronous designs can be used to provide fault detection and therefore, fault tolerance. Chapter 6 and Chapter 7 present the basis to such an approach to fault tolerance, one not possible with conventional, clocked, systolic arrays. This method of fault detection/correction exploits the inherent redundancy of architectures using four-state coding, a data-driven technique for implementing bit-level wave-front arrays.

Chapter 2 The Problem of Division

The division operation involves finding the quotient (Q) and remainder (R) of a dividend (N) and divisor (D). These four parameters satisfy the equation:

$$N = Q \times D + R, \tag{1}$$

where $|R|$ is less than $|D|$ and of the same sign as N. The usual method of division, the paper-and-pencil method, is illustrated with Example 2-1, which shows the division of 50333 by 93. Example 2-2 shows division of the same numbers using the same method, but it includes calculations and comparisons that are needed and implied in using the paper-and-pencil method, but are not explicitly included in most written calculations.

$$\begin{array}{rrl}
 & 541 & = \text{quotient} \\
\text{divisor} = 93 & \overline{)\,50333} & = \text{dividend} \\
 & \underline{-46500} & = 93 \times 500 \\
 & 3833 & \\
 & \underline{-3720} & = 93 \times 40 \\
 & 113 & \\
 & \underline{-93} & = 93 \times 1 \\
 & 20 & = \text{remainder}
\end{array}$$

Example 2-1

$$\begin{array}{rrl}
\text{divisor} = 93 & \overline{)\,50333} & = \text{dividend} = pr^0 \\
 & & (50333 < 93 \times 1000) \Longrightarrow q_3 = 0 \\
 & & (50333 > 93 \times 100) \\
 & & (50333 > 93 \times 200) \\
 & & (50333 > 93 \times 300) \\
 & & (50333 > 93 \times 400) \\
 & & (50333 > 93 \times 500) \\
 & & (50333 < 93 \times 600) \Longrightarrow q_2 = 5 \\
 & \underline{-46500} & = 93 \times 500 \\
 & 3833 & = pr^1 \\
 & & (3833 > 93 \times 10) \\
 & & (3833 > 93 \times 20) \\
 & & (3833 > 93 \times 30) \\
 & & (3833 > 93 \times 40) \\
 & & (3833 < 93 \times 50) \Longrightarrow q_1 = 4 \\
 & \underline{-3720} & = 93 \times 40 \\
 & 113 & = pr^2 \\
 & & (113 > 93 \times 1) \\
 & & (113 < 93 \times 2) \Longrightarrow q_0 = 1 \\
 & \underline{-93} & = 93 \times 1 \\
 & 20 & = \text{remainder} = pr^3
\end{array}$$

$$\begin{aligned}
\text{quotient} &= q_3 \times 10^3 + q_2 \times 10^2 + q_1 \times 10^1 + q_0 \times 10^0 \\
&= 0 \times 1000 + 5 \times 100 + 4 \times 10 + 1 \times 1 \\
&= 541
\end{aligned}$$

Example 2-2

For an n digit dividend and an m digit divisor, we must generate an $n-m+1$ digit quotient and a remainder of no more than m digits. To generate each quotient digit, one must find the greatest single-digit multiple of the divisor that when subtracted from the partial remainder yields a non-negative, new partial remainder. As Example 2-2 shows, some quotient digits may require several multiplications or additions and then comparisons to complete the search. In binary division this potentially troublesome search is curtailed because the quotient bit can be at most 1. Of course, more quotient bits need to be generated in binary division than in the case of decimal division, but the lack of complexity, reduced overall area, and faster cycle time make binary division preferable for a digital logic implementation. Complexity, as it is used in this thesis, refers to design complexity, not algorithmic complexity; a more complex design generally requires more area and sometimes more delay.

Example 2-3 illustrates the paper-and-pencil method with binary operands dividing 503 (= 010011) by 9 (= 01001) . As this example shows, each quotient bit is determined by comparing the current partial remainder with one multiple of the divisor that can be obtained by the shifting of bits, without additions.

$$\begin{array}{lrl}
\text{divisor} = 01001 & \overline{)\,0111110111} & = \text{dividend} = pr^0 = 503 \\
 & \underline{-0100100000} & \quad q_5 = 1 \\
 & 011010111 & = pr^1 = 503 - 9 \times 2^5 = 215 \\
 & \underline{-010010000} & \quad q_4 = 1 \\
 & 01000111 & = pr^2 = 215 - 9 \times 2^4 = 71 \\
 & \underline{-0100100} & \quad (71 < 9 \times 2^3) \Longrightarrow q_3 = 0, q_2 = 1 \\
 & 0100011 & = pr^3 = 71 - 9 \times 2^2 = 35 \\
 & \underline{-010010} & \quad q_1 = 1 \\
 & 010001 & = pr^4 = 35 - 9 \times 2^1 = 17 \\
 & \underline{-01001} & \quad q_0 = 1 \\
 & 01000 & = \text{remainder} = pr^5 = 17 - 9 \times 2^0 = 8
\end{array}$$

$$\begin{aligned}
\text{quotient} &= q_5 \times 10^5 + q_4 \times 10^4 + q_3 \times 10^3 + q_2 \times 10^2 + q_1 \times 10^1 + q_0 \times 10^0 \\
&= 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1 \\
&= 55
\end{aligned}$$

Example 2-3

Although the final remainder was calculated after five subtractions ($2^5 \times D$, $2^4 \times D$, $2^2 \times D$, $2^1 \times D$, and $2^0 \times D$), all six multiples of the divisor had to be compared with the current partial remainder.

Normalization, or sizing, of the divisor was also required to determine how many quotient bits had to be generated. In the paper-and-pencil method, this is accomplished by lining up the first non-zero digits of the divisor and dividend. In the examples trailing zeros were added to this shifted version of the divisor; written calculations often omit them.

Division and multiplication are in many respects dual operations. As a shift-and-subtract process, division superficially resembles the shift-and-add method of multiplication. Division, however, requires the results of one subtraction to determine the next quotient bit. This introduces a sequential ordering in the subtraction of multiples of the divisor from the partial remainders, which is not present in the addition of the partial products.

Chapter 3 Current Dividers

3.1 Introduction

This chapter contains a brief survey of conventional divider implementations, the algorithms they implement, their performance, and the area they require. The emphasis is on those that combine $O(n)$ delay with $O(n)$ area, where n is the number of bits of the operands. Area-efficient division architectures (those achieving $O(n)$ delay or less) are needed for the large operands required in Public Key Cryptography and many other digital signal-processing applications.

3.2 Commonly Implemented Dividers

Often, using iterative techniques [1], the faster multiplier has been used to speed up division. However, this is very area-inefficient, unless a multiplier is already required. A division algorithm based on multiplication that achieves $O(\log n)$ delay exists [2], but it requires great complexity and more than $O(n^2)$ area.

Other methods, notably the CORDIC algorithm, employ a fast adder in their design. This extremely versatile algorithm uses methods based on coordinate rotation to calculate trigonometric functions, multiplication, conversion between binary and mixed-radix number systems, and division [1]. Because of this versatility, variations of the CORDIC architecture are often used in general purpose computers and calculators. This method has an $O(n \log n)$ worst-case delay because it makes use of $O(n)$ iterations of three fast adders (each with $O(\log n)$ delay) operating in parallel.

3.3 Requirements for $O(n)$ Delay and $O(n)$ Area Dividers

Other conventional divider implementations use some variant of the paper and pencil method of division: either by doing explicit comparison and subtraction (non-restoring), or by using subtraction and compensating addition (restoring division). For binary radix arithmetic, the recursive loop at the center of this class of dividers is given by following equation.

$$R^{i+1} = R^i - q_i \times 2^i \times D \tag{2}$$

Although there are often initialization and rounding operations before and after the central loop, Equation 2is the key arithmetic operation. Therefore, the central problem for dividers of the same class as the paper and pencil method reduces to two related operations needed to implement Equation 2:

d1. Selecting q_i, by comparing partial remainder with the divisor (0 or 1 for the binary paper-and-pencil method).

d2. Adding/subtracting R^i and a multiple of the divisor.

How these two operations (*d1* and *d2*) are performed differentiates the performance of the various dividers based on the paper-and-pencil method.

Since the selection of q_i (step *d1*) and the addition (step *d2*) in the paper-and-pencil method are both at best $O(\log n)$ operations in VLSI [3], a straightforward implementation of the paper-and-pencil method requires n iterations and therefore $O(n \log n)$ delay. Altering these operations such that the delay for each of n iterations is constant, independent of n, would achieve $O(n)$ delay . By using a constant time adder that introduces redundancy into the partial remainder, the need for a complete $O(\log n)$ addition at each iteration is avoided.

3.3.1 Constant time selection of q_i

It will be shown by later examples that introducing redundancy into the representation of the quotient digit q_i can allow determination by a constant number of bits of the divisor and remainder. The redundancy allows subsequent quotient digits to make up for some error. A trivial example with $q_i \in \{+1, -1\}$, depending only on the sign bit of the current partial remainder, is shown in Example 3.3.1-1.

The series of +1's and -1's produced as a quotient in Example 3.3.1-1 must finally be recoded into the non-redundant form of 0's and 1's with an addition.

$$\begin{array}{rrl}
\text{divisor} = 01001 & \big)\overline{0111110111} & = \text{dividend} = pr^0 = 503 \\
& \underline{-0100100000} & (503 > 0 \Longrightarrow q_5 = 1 \\
& 011010111 & = pr^1 = 503 - 9 \times 2^5 = 215 \\
& \underline{-010010000} & (215 > 0 \Longrightarrow q_4 = 1 \\
& 01000111 & = pr^2 = 215 - 9 \times 2^4 = 71 \\
& \underline{-0100100} & (71 > 0) \Longrightarrow q_3 = 1 \\
& 11111111 & = pr^3 = 71 - 9 \times 2^3 = -1 \\
& \underline{+0100100} & (-1 < 0) \Longrightarrow q_2 = -1 \\
& 0100011 & = pr^4 = -1 + 9 \times 2^2 = 35 \\
& \underline{-010010} & (35 > 0) \Longrightarrow q_1 = 1 \\
& 010001 & = pr^5 = 35 - 9 \times 2^1 = 17 \\
& \underline{-01001} & (17 > 0) \Longrightarrow q_0 = 1 \\
& 01000 & = \text{remainder} = pr^6 = 17 - 9 \times 2^0 = 8
\end{array}$$

$$\begin{aligned}
\text{quotient} &= q_5 \times 10^5 + q_4 \times 10^4 + q_3 \times 10^3 + q_2 \times 10^2 + q_1 \times 10^1 + q_0 \times 10^0 \\
&= 1 \times 32 + 1 \times 16 + 1 \times 8 - 1 \times 4 + 1 \times 2 + 1 \times 1 \\
&= 55
\end{aligned}$$

Example 3.3.1-1

3.4 Constant Time Adders

In integer addition (as opposed to polynomial or finite field addition), the carry propagation slows the worst-case addition time, as shown in Example 3.4-2, where the carry bit of the least significant bits leads eventually to a carry out of the highest bit. A standard $O(n)$ delay and $O(n)$ area implementation of an n-bit carry-propagating adder is shown in Figure 3.4-2. This adder is a linear array of n full adders. Each full adder produces a two-bit total, SUM and $CARRYOUT$, of its three binary inputs, A, B, and $CARRYIN$. The equations relating the outputs of the full adder, shown in Figure 3.4-1, to the inputs are given with the following:

$$SUM = A + B + CARRYIN \pmod 2$$

$$= A \oplus B \oplus CARRYIN$$

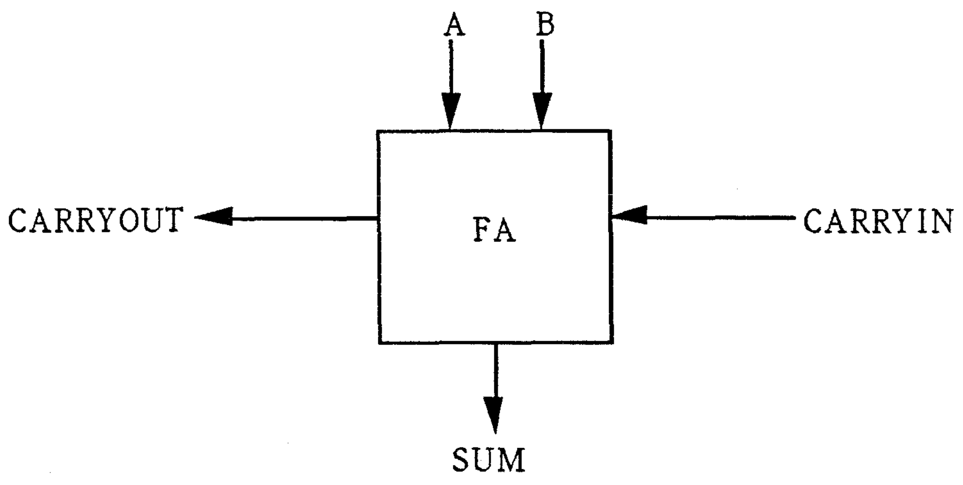

Figure 3.4-1: Full adder.

and

$$CARRYOUT = \lfloor (A + B + CARRYIN) \div 2 \rfloor$$

$$= (A \wedge B) \vee (A \wedge CARRYIN) \vee (B \wedge CARRYIN)$$

$$\begin{array}{ccccccccc}
 & \Leftarrow 1 & \Leftarrow 1 & \Leftarrow 1 & \Leftarrow 1 & \Leftarrow 1 & \Leftarrow 1 & \Leftarrow 1 & \Leftarrow 1 \\
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Example 3.4-2: Carry-Propagating Addition

Even the fastest non-redundant adder algorithms, such as the Brent-Kung adder [3], have delay $O(\log n)$ at the expense of area. Producing a constant time adder requires the introduction of some redundancy into the operands. Carry-save addition and signed-digit addition are two ways of doing this.

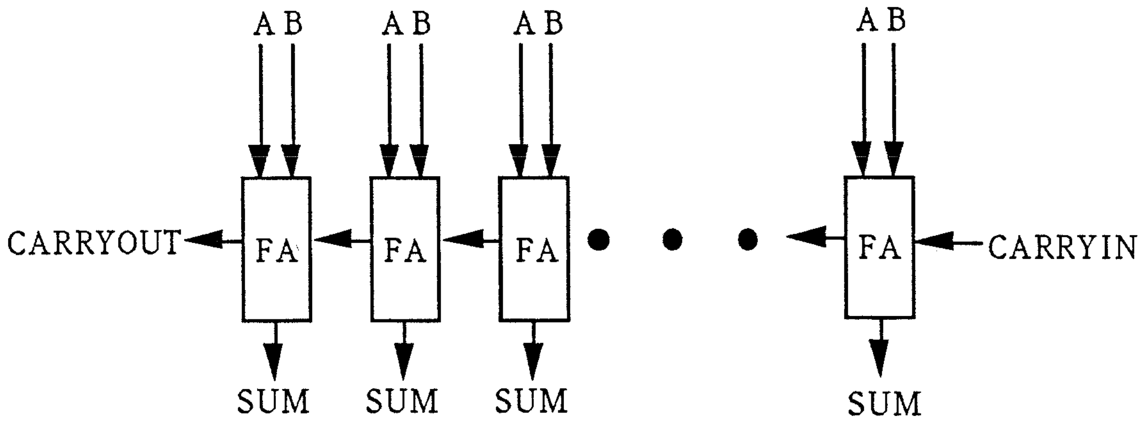

Figure 3.4-2: Carry-Propagating Adder.

3.4.1 Carry-Save Addition

In carry-save addition, the current total is in the form of two n-bit numbers, the carry and the sum. With each addition an n-bit number is added to the old sum and carry, and the result is in the form of a new sum and a new carry. Addition is performed as in standard addition, but the carry's of each bit are not propagated but are saved to be added in the next addition at the full adder to the left. Eventually the sum and carry must be added together in non-redundant addition; but when a series of m $n-$bit numbers are to be added, the whole addition can be performed in $m + n$ full-adder delays, rather than $m \times n$ full-adder delays. An example of the carry-save addition of a series of binary numbers is given in Example 3.4.1-3.

Addition of the binary numbers 00110011, 00010100, 00010001, and 01011100:

$$\begin{array}{rcll}
000110011 & = & \text{sum}^0 & = 99 \\
000000000 & = & \text{carry}^0 & = 0 \\
\underline{+\ 000010100} & = & \text{b} & = 36 \\
000100111 & = & \text{sum}^1 & = 71 \\
000100000 & = & \text{carry}^1 & = 64 \\
 & & & \\
000100111 & = & \text{sum}^1 & = 71 \\
000100000 & = & \text{carry}^1 & = 64 \\
\underline{+\ 000010001} & = & \text{b} & = 33 \\
000010110 & = & \text{sum}^2 & = 38 \\
001000010 & = & \text{carry}^2 & = 130 \\
 & & & \\
000010110 & = & \text{sum}^2 & = 38 \\
001000010 & = & \text{carry}^2 & = 130 \\
\underline{+\ 001011100} & = & \text{b} & = 172 \\
000001000 & = & \text{sum}^3 & = 8 \\
010101100 & = & \text{carry}^3 & = 332 \\
 & & & \\
000001000 & = & \text{sum}^3 & = 8 \\
\underline{+\ 010101100} & = & \text{carry}^3 & = 332 \\
010110100 & = & \text{total} & = 340
\end{array}$$

Example 3.4.1-3:

Carry-Save Addition

3.4.2 Constant time addition using more redundancy

Constant time addition can also be achieved using more redundancy, as in signed-digit addition. Each operand digit is allowed to take on any integer in the range $[-r, +r]$, where $r > 1$. When adding two numbers (Z and Y) together, we first form two intermediate results, the direct sum (W) and the transit (T), before forming the final sum (S). Because of the large operands, higher radix forms are not considered; although faster, they require significantly more area and complexity. The minimum-area, signed-digit adder with r = 2 is demonstrated in Example 3.4.2-

4.

$$\begin{aligned} \text{Augend} = \text{Z} = \ & 0\ 2\ 0\ 1\ 1\ 2\ 0 && = 2\times 3^5 + 1\times 3^3 + 1\times 3^2 + 2\times 3^1 = 528 \\ \text{Addend} = \text{Y} = \ & 0\ 0\ 1\ 2\ 0\ 1\ 1 && = 1\times 3^4 + 2\times 3^3 + 1\times 3^1 + 1\times 3^0 = 139 \end{aligned}$$

bit#	6	5	4	3	2	1	0	
Augend = Z =	0	2	0	1	1	2	0	
Addend = Y =	0	0	1	2	0	1	1	
Intermediate sum = N =	0	2	1	3	1	3	1	$n_i = z_i + y_i$
Carry = C =	0	1	0	1	0	1	0	$c_i = 0$ for $-1 < n_i < 1$
								$c_i = 1$ for $n_i > 1$
								$c_i = -1$ for $n_i < -1$
Direct Sum = W =	0	-1	1	0	1	0	1	$w_i = n_i - 3 \times c_i$
Transit = T =	1	0	1	0	1	0	0	$t_i = c_i - 1$
Sum = S =	1	-1	2	0	2	0	1	$s_i = w_i + t_i$

$$\begin{aligned} \text{Sum} &= s_6\times 3^6 + s_5\times 3^5 + s_4\times 3^4 + s_3\times 3^3 + s_2\times 3^2 + s_1\times 3^1 + s_0\times 3^0 \\ &= 1\times 3^6 - 1\times 3^5 + 2\times 3^4 + 2\times 3^2 + 1\times 3^0 \\ &= 667 \end{aligned}$$

Example 3.4.2-4

3.4.3 Comparison of carry-save and signed-digit addition

The two basic, constant-time add techniques have a real difference in redundancy and complexity. The carry-save adder is shown in Figure 3.4.3-3. The structure of the signed-digit adder is shown in Figure 3.4.3-4. A minimum of three bits are necessary to express the five different values possible (2 to -2) for each of two digits input-to and the one digit output-from each stage of the signed-digit adder. In comparison, each bit of the carry-save has three inputs ($SUMIN$, B, and $CARRYIN$), each expressed with only one bit, and two bits of output expressing four different values ($SUM = 0$ or 1, $CARRYOUT = 0$ or 1) ; thus the circuitry of the signed-digit adder must process six bits for input and contain latches for the three-bits output at each of n stages. This is the most important difference between implementations of the two constant time adders.

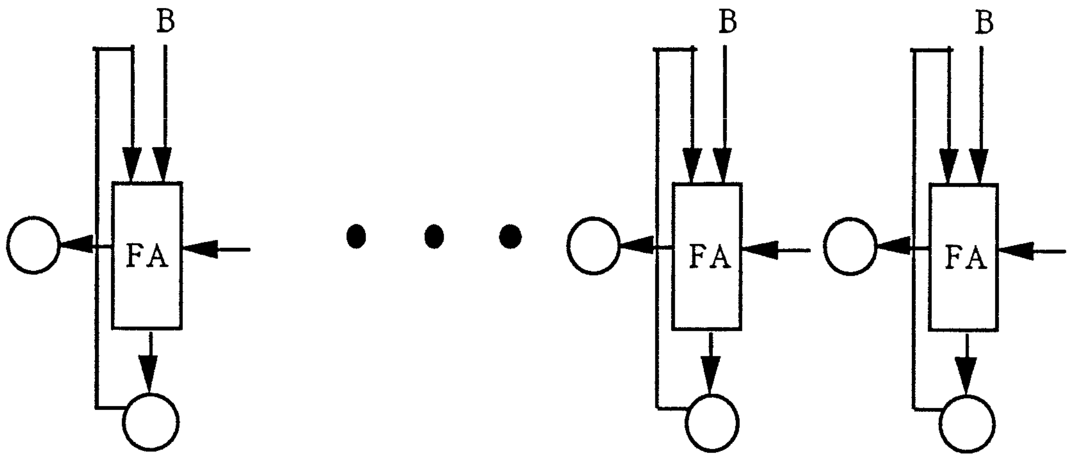

Figure 3.4.3-3: Carry Save Adder.

Since each stage of output must be latched, the signed-digit adder requires three registers per stage, while the carry-save needs only two. Significantly more area is required for the complex circuitry needed to process the six-bits input per stage of the signed-digit adder. To generate the transit bit, t_i, and direct sum, w_i, for each stage addition of the three-bit operands, complexity and area equivalent to three bits of binary addition are required (delay and area of three full adders, or some equivalent trade-off of increased area for delay reduction to that of a single full adder). The final combination of the two bit w_i's and two bit t_i's requires area and delay equivalent to two full adders. Thus, each stage of the signed-digit adder requires three latches and an area-delay trade-off equivalent to five full adders. Each bit of the carry save adder requires just one full adder and two latches. Although the increased range of the operands, -2 to 2, of the signed-digit adder means that only about half the number of stages is required to add over the same range (ignoring preprocessing to convert the usual binary operands), the total area cost of the

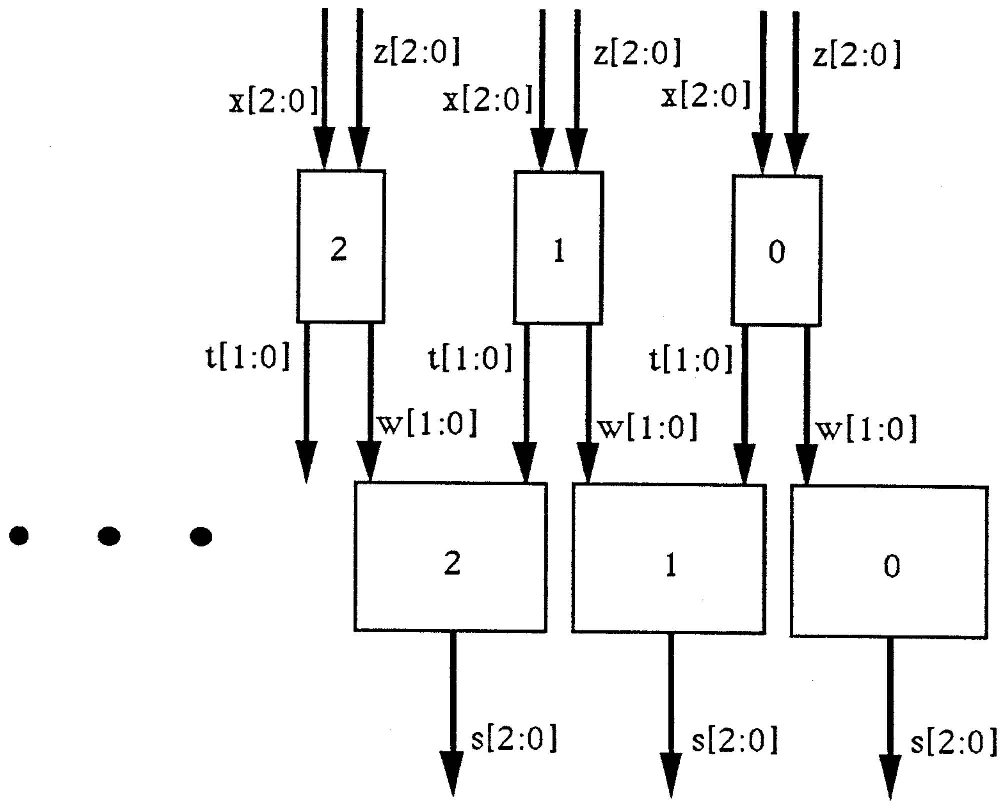

Figure 3.4.3-4: Signed Digit Adder.

redundant adder is about three times that of the carry-save [1]. Although the extra redundancy of the signed-digit adder might mean selection of q_i with less digits of information, the carry-save adder has clear complexity and area advantages which make it a better choice for large operand arithmetic.

3.5 Purdy and Purdy Divider

The Purdy and Purdy divider (PP) uses carry-save adders to achieve $O(n)$ delay and $O(n)$ area [4]. PP divides non-negative n-bit integers by positive n-bit integers in $O(n)$ time with $O(n)$ area. It differs from most dividers in that the remainder is calculated first. At each iteration three consecutive tests are performed on the most significant bits of a carry-save representation of the partial remainder. Each of these three tests relies on only one of the most significant two bits of the sum and one of the most significant two bits of the carry. A positive result causes the current, shifted version of the divisor to be subtracted. At most, two of these three tests will cause a subtraction to occur. The most significant bit of the carry and the most significant bit of the sum are eliminated with these tests/subtractions reducing them by one bit each iteratiion. After no more than n iterations, the final result of these successive reductions is a carry-save version of the partial remainder that is equal to $R + k \times D$, where $k \in \{0, 1, 2\}$. A non-redundant addition and two comparisons are made finally to produce the remainder. The remainder is then subtracted from the dividend to make a divisible dividend. The quotient is then generated from least to most significant bit by successively testing the even/oddness of the divisible dividend and subtracting. This difference in generation of the quotient is not fundamental; some form of the quotient is still needed to get the remainder, in this case $q_i \in \{+2, +1, 0\}$. An example taken from [4] is given in Example 3.5-5. It shows the contents of the sum register and the carry register, *prsum* and *prcarry*, which contain the partial remainder after each test/subtraction.

Initialize: load dividend into sum register
find the 2's complement representation of -9
normalize (multiply -9×16 in this case)

prsum= 503 = 0111110111
prcarry= 0
$B = -9 \times 16 = 101110000$

i = 0	B = 101110000	prsum =	010000111	010000111	000010111
		prcarry =	011100000	011100000	011000000
i = 1	B = 110111000	prsum =	001101111	001101111	001101111
		prcarry =	000100000	000100000	000100000
i = 2	B = 111011100	prsum =	000010011	000010111	000010111
		prcarry =	001011000	000110000	000110000
i = 3	B = 111101110	prsum =	000001001	000001011	000001011

	prcarry =	000101100	000011000	000011000
i = 4 B = 111110111	prsum =	000000100	000000101	000000101
	prcarry =	000010110	000001100	000001100

Example 3.5-5

At each of the three steps for each iteration, the sum and carry registers stay the same, or a carry-save addition is performed to combine *prsum*, *prcarry*, and B. At the end of each iteration, i, the first $i+1$ bits of the sum and carry registers are all zero. In Example 3.5-6, the total contained in *prsum* and *prcarry* after the last iteration is equal to 17 $(= 5 + 12)$ and the divisor must be subtracted to once to get the final remainder, $R = 17 - 9 = 8$.

3.6 Preparata and Vuillemin Dividers

There are those dividers that use more redundant arithmetic [5], [6], [7], [8], [2], and [9]. The Preparata and Vuillemin divider (PV) achieves $O(n)$ area and $O(n)$ delay [2]. It produces a redundant version of the remainder with approximately n redundant additions rather than the approximately $3 \times n$ required by PP. Functionally, the additions are quite similar to the standard carry-save in that transition bits or carries are saved for addition at the next add; but PV requires more redundant additions than PP. Even the least redundant version of PV requires $s_j \in \{-1, 0\}$ and $c_j \in \{-1, 0, 1\}$, meaning that more complex circuitry is needed to implement the adder for each of n stages and that two latches rather than one are needed for carry storage. Further, generation of the quotient is based on 5 bits of the partial remainder and 8 bits of the divisor.

Chapter 4 The Divider Algorithm

4.1 Introduction

This chapter describes a fast algorithm that offers divides in approximately n steps with n cells, to attain better performance than comparable proposed dividers, such as PV and PP. This chapter includes the division algorithm to be implemented, the way it works, and finally, the divider is compared with PP and PV.

4.2 The Algorithm

This algorithm is based on the paper-and-pencil method and therefore the partial remainder is generated in successive steps on the basis of the equation:

$$PR^i = PR^{i-1} - q_i \times D \times 2^i \tag{3}$$

where PR^i represents the partial remainder after iteration i, q_i is the quotient value for iteration i, and D is the divisor. The algorithm divides an n-bit dividend, N, by the n-bit divisor, D, to generate a quotient and a remainder. N and D can be either positive or negative integers.

The addition or subtraction for each iteration is performed on 2's complement operands, using carry-save addition, where the carry's are not propagated but stored as inputs to the adder during the next add. With each iteration of the algorithm the sum and carry registers containing the partial remainder, *prsum* and *prcarry*, decrease in size by one bit, until each are one bit smaller than the divisor, D. That is:

$$\lfloor \log_2(|prsum|) \rfloor < \lfloor \log_2(|D|) \rfloor$$

and

$$\lfloor \log_2(|prcarry|) \rfloor < \lfloor \log_2(|D|) \rfloor.$$

The two most significant bits of prsum and prcarry and the sign of the divisor determine the quotient value for each iteration. This quotient value, $q_i \in \{-2, -1, 0, 1, 2\}$, determines which multiple of the divisor is to be subtracted from the current partial remainder. All these multiples are readily obtained from D using only shifts and bitwise inversions.

In the following program text for the algorithm, the partial remainder is represented by the sum of the values in $PRSUM$ and $PRCARRY$; similarly, the sum of $QSUM$ and $QCARRY$ represents the current quotient. As shown in the program, the algorithm begins by assigning to the variable SD a normalized version of the divisor, where $|SD|$ is greater than or equal to 2^{n-2}. The number of shifts of the divisor required to find this normalized version is stored in the variable m.

In each iteration of Step 2, a quotient value, q, is determined from the first two bits of $PRSUM$ and $PRCARRY$. As per Equation 3, a multiple of the divisor, specifically, $q \times SD$, is subtracted from the current partial remainder to get a new one. The subtraction is done using carry-save addition designated in the program with the notation:

$$< sum, carry >:= sum + carry + b.$$

After the final reductions to $PRSUM$ and $PRCARRY$, a comparison must be made with the divisor to determine whether the partial remainder needs to be adjusted by the divisor. The adjustment is performed with a last carry save addition. Finally, non-redundant 2's complement addition is performed on $PRSUM$ and $PRCARRY$ to obtain the remainder.

```
INPUT    DIVIDEND[n ], DIVISOR[n];
OUTPUT   QUOTIENT[n], REMAINDER[n];

step1:

begin
    PRSUM:= DIVIDEND; PRCARRY:=0; SD:= DIVISOR;
    m:=0;
    while (SD[n-1] = SD[n-2] & m != n-1)
        do begin
```

```
            SD := SD<<1;
          m := m +1;
       end

step 2:
     i := 0;
    while (i != m+1)
     do begin

       sumq := -2*(PRSUM[n-1-i] + PRCARRY[n-1-i]) +
              (PRSUM[n-2-i] + PRCARRY[n-2-i]);

       if (sumq = -4) q:= -2*SIGN(DIVISOR);
       if (sumq = -3 || sumq = -2) q:= -1*SIGN(DIVISOR);
       if (sumq = -1) q:= 0;
       if (sumq = 0 || sumq = 1) q:= 1*SIGN(DIVISOR);
       if (sumq = 2) q:= 2*SIGN(DIVISOR);

       <PRSUM, PRCARRY> := PRSUM + PRCARRY -q*SD;
       truncate(PRSUM, PRCARRY);

       QSUM := QSUM <<1;
       QCARRY := QCARRY << 1;
       <QSUM, QCARRY> := QSUM + QCARRY + q;

       SD := SD >> 1;
    end;

Step 3:

    q1st := compare(PRSUM, PRCARRY, SIGN(DIVIDEND),SIGN(DIVISOR))

   <PRSUM, PRCARRY> := PRSUM + PRCARRY -q1st*DIVISOR;
    <QSUM, QCARRY> := QSUM + QCARRY + q1st;

    REMAINDER := PRSUM + PRCARRY;
    QUOTIENT := QSUM + QCARRY;

    end;
```

Example 4.2-1 shows the algorithm working to divide 503 by 9. In normalization the divisor, 9, is shifted over 5 times to form the initial 10 bit shifted divisor, 288. Successive tests on the most significant bits of *prsum* and *prcarry* and resulting additions reduce the partial remainder to -1. Since the sign of the remainder and the sign of the dividend, 503, need to be the same, the divisor is added to the partial remainder to form the final remainder, 8. This example of the algorithm, like the others that follow, was generated using a C program simulation of the algorithm. In Example 4.2-5, the PP algorithm was used to perform division on the same numbers and 15 tests/additions were required.

```
Dividing an 9 bit number by a 4 bit number:
  dividend =>        503 (00000000000000000000000111110111)
   divisor =>          9 (00000000000000000000000000001001)
-----------------------------------------------------------
```

```
        sd =>         288 (                          0100100000)
    pr_sum =>         503 (                          0111110111)
    pr_car =>           0 (                          0000000000)
    pr                    503
-----------------------------------------------------------------
At bit position 9  a=01 b=00 sum=1  So, pr = pr + -1*sd
-----------------------------------------------------------------
a=01 b=00 sum=1  So, pr = pr + -1*sd
-----------------------------------------------------------------
        sd =>         144 (                           010010000)
    pr_sum =>          41 (                           000101001)
    pr_car =>         174 (                           010101110)
    pr                    215
-----------------------------------------------------------------
At bit position 8  a=00 b=01 sum=1  So, pr = pr + -1*sd
-----------------------------------------------------------------
a=00 b=01 sum=1  So, pr = pr + -1*sd
-----------------------------------------------------------------
        sd =>          72 (                            01001000)
    pr_sum =>         105 (                            01101001)
    pr_car =>         -34 (                            11011110)
    pr                     71
-----------------------------------------------------------------
At bit position 7  a=01 b=11 sum=0  So, pr = pr + -1*sd
-----------------------------------------------------------------
a=01 b=11 sum=0  So, pr = pr + -1*sd
-----------------------------------------------------------------
        sd =>          36 (                             0100100)
    pr_sum =>         -63 (                             1000001)
    pr_car =>          62 (                             0111110)
    pr                     -1
-----------------------------------------------------------------
At bit position 6  a=10 b=01 sum=-1  So, pr = pr + 0*sd
-----------------------------------------------------------------
a=10 b=01 sum=-1  So, pr = pr + 0*sd
-----------------------------------------------------------------
        sd =>          18 (                              010010)
    pr_sum =>          31 (                              011111)
    pr_car =>         -32 (                              100000)
    pr                     -1
-----------------------------------------------------------------
At bit position 5  a=01 b=10 sum=-1  So, pr = pr + 0*sd
-----------------------------------------------------------------
a=01 b=10 sum=-1  So, pr = pr + 0*sd
-----------------------------------------------------------------
        sd =>           9 (                               01001)
    pr_sum =>          15 (                               01111)
    pr_car =>         -16 (                               10000)
    pr                     -1
-----------------------------------------------------------------
At bit position 4  a=01 b=10 sum=-1  So, pr = pr + 0*sd
-----------------------------------------------------------------
a=01 b=10 sum=-1  So, pr = pr + 0*sd
```

```
------------------------------------------------------------
       sd =>          4 (                             0100)
   pr_sum =>          7 (                             0111)
   pr_car =>         -8 (                             1000)
   pr                   -1
------------------------------------------------------------
Final Adjustment step:  pr adjusted by +divisor
------------------------------------------------------------
  quotient =>        55 (00000000000000000000000000110111)
 remainder =>         8 (00000000000000000000000000001000)
------------------------------------------------------------
Confirmation --> 503 / 9 = 55 remainder 8
------------------------------------------------------------
```

Example 4.2-1 Division of 503 by 9

Example 4.2-2 shows division of 503 by -9. Here the initial shifted version of the divisor is -288. As when the divisor was +9, the final partial remainder is -1. This partial remainder needs to be adjusted by subtracting the divisor to obtain the final remainder of +8.

```
Dividing an 9 bit number by a 4 bit number:
  dividend =>       503 (00000000000000000000000111110111)
   divisor =>        -9 (11111111111111111111111111110111)
------------------------------------------------------------
       sd =>       -288 (                       1011100000)
   pr_sum =>        503 (                       0111110111)
   pr_car =>          0 (                       0000000000)
   pr                  503
------------------------------------------------------------
At bit position 9  a=01 b=00 sum=1  So, pr = pr + 1*sd
------------------------------------------------------------
a=01 b=00 sum=1  So, pr = pr + -1*sd
------------------------------------------------------------
       sd =>       -144 (                        101110000)
   pr_sum =>         23 (                        000010111)
   pr_car =>        192 (                        011000000)
   pr                  215
------------------------------------------------------------
At bit position 8  a=00 b=01 sum=1  So, pr = pr + 1*sd
------------------------------------------------------------
a=00 b=01 sum=1  So, pr = pr + -1*sd
------------------------------------------------------------
       sd =>        -72 (                         10111000)
   pr_sum =>         39 (                         00100111)
   pr_car =>         32 (                         00100000)
   pr                   71
------------------------------------------------------------
```

```
At bit position 7  a=00 b=00 sum=0  So, pr = pr + 1*sd
------------------------------------------------------------
a=00 b=00 sum=0  So, pr = pr + -1*sd
------------------------------------------------------------
        sd =>        -36 (                        1011100)
    pr_sum =>         -1 (                        1111111)
    pr_car =>          0 (                        0000000)
    pr                     -1
------------------------------------------------------------
At bit position 6  a=11 b=00 sum=-1  So, pr = pr + 0*sd
------------------------------------------------------------
a=11 b=00 sum=-1  So, pr = pr + 0*sd
------------------------------------------------------------
        sd =>        -18 (                         101110)
    pr_sum =>         31 (                         011111)
    pr_car =>        -32 (                         100000)
    pr                     -1
------------------------------------------------------------
At bit position 5  a=01 b=10 sum=-1  So, pr = pr + 0*sd
------------------------------------------------------------
a=01 b=10 sum=-1  So, pr = pr + 0*sd
------------------------------------------------------------
        sd =>         -9 (                          10111)
    pr_sum =>         15 (                          01111)
    pr_car =>        -16 (                          10000)
    pr                     -1
------------------------------------------------------------
At bit position 4  a=01 b=10 sum=-1  So, pr = pr + 0*sd
------------------------------------------------------------
a=01 b=10 sum=-1  So, pr = pr + 0*sd
------------------------------------------------------------
        sd =>         -5 (                           1011)
    pr_sum =>          7 (                           0111)
    pr_car =>         -8 (                           1000)
    pr                     -1
------------------------------------------------------------
Final Adjustment step:  pr adjusted by +|divisor|
------------------------------------------------------------
  quotient =>        -55 (11111111111111111111111111001001)
 remainder =>          8 (00000000000000000000000000001000)
------------------------------------------------------------
Confirmation --> 503 / -9 = -55 remainder 8
------------------------------------------------------------
```

Example 4.2-2 Division of 503 by -9

4.3 Why the Algorithm Works

Step 1 of the algorithm normalizes the divisor by multiplying it by 2 (with shifts) m times. Thus, after Step 1,

$$SD = 2^m D.$$

At the end of Step 1, SD_{n-1} is not equal to SD_{n-2}, so for a negative divisor $SD_{n-1} = 1$, $SD_{n-2} = 0$, and for a positive divisor $SD_{n-1} = 0$, $SD_{n-2} = 1$. Since these bits represent -2^{n-1} and $+2^{n-2}$, respectively, and the lower $n-2$ bits can contribute at most $2^{n-2} - 1$, the following ranges apply for SD:

$$-2^{n-1} \leq SD < -2^{n-2}, D < 0$$

$$2^{n-2} \leq SD < 2^{n-1}, D > 0$$

or

$$2^{n-2} \leq |SD| \leq 2^{n-1},$$

and since $D = 2^{-m} \times SD$,

$$2^{n-m-2} \leq |D| \leq 2^{n-m-1}. \tag{4}$$

Since the basis of the algorithm, Step 2, is the ability to compress the partial remainder, prsum and prcarry, into 2's complement representations that are 1 bit smaller after each iteration, it must be shown that q_i and SD^i are chosen in such a way as to make this compression possible.

Initially, $prsum$ and $prcarry$ are n-bit 2's complement numbers. $prsum_{n-1}$ and $prcarry_{n-1}$, the sign bits, can each contribute -2^{n-1} to the partial remainder. $prsum_{n-2}$ and $prcarry_{n-2}$ can each contribute $+2^{n-2}$. The total contribution of the first two bits of $prsum$ and $prcarry$, sb, is therefore given by the following:

$$sb = (-2 \times prsum_{n-1} - 2 \times prcarry_{n-1} + prsum_{n-2} + prcarry_{n-2}) \times 2^{n-2}.$$

or, using *sumq* as defined in the program:

$$sb = sumq \times 2^{n-2}$$

Since $prsum_{n-1}$, $prcarry_{n-1}$, $prsum_{n-2}$, and $prcarry_{n-2}$ are each $\in \{0,1\}$, *sumq* is $\in \{-4,-3,-2,-1,0,1,2\}$.

The lower $n-2$ bits of *prsum* and *prcarry* will, after a carry-save addition with *any* $n-2$ bit binary number, produce a new *prsum* and *prcarry* with $n-2$ bits each and one carryout, *co*, with a potential contribution of $+2^{n-2}$.

Compression to two $n-1$ bit numbers is possible if the new partial remainder is in the range $[-2^n, 2^n - 2]$, the range that the sum of two $n-1$ bit 2's complement numbers can represent. Using only this range requirement might require adjustment of all the bits of both registers, however. The algorithm works by choosing q_i such that only the first bits need to be adjusted. This is possible because the algorithm ensures that each carry-save addition results in a total contribution by the upper bits, which is in the range of values that can be represented with the sign bits of $n-1$ bit 2's complement numbers, 0, -1, or $-2 \times 2^{n-2}$. That is,

$$sumq \times 2^{n-2} + co \times 2^{n-2} + bcont = sbnew = (0, -1, or - 2) \times 2^{n-2}, \qquad (5)$$

where *bcont* is the contribution from $-q_i \times SD$, and *sbnew* is the new value to be represented in the sign bits.

It will now be shown that every possible value of *sumq* and resulting $q_i \times SD$ achieves the appropriate range for *sbnew*, which can therefore be mapped into thenew sign bits.

For $sumq = -4$:

$q = -2, B = 2 \times |SD|$, an $n+1$ bit 2's complement number with $b_n = 0$, $b_{n-1} = 1$ and $b_{n-2} = sdnext = sd_{n-3}$, for $D > 0$ or $\overline{sd}_{n-3}$, for $D < 0$,

$$\begin{aligned} sbnew &= sumq \times 2^{n-2} + 1 \times 2^{n-1} + sdnext \times 2^{n-2} + co \\ &= [-4 + 2 + sdnext + co] \times 2^{n-2} \\ &= [-2 + sdnext + co] \times 2^{n-2}. \end{aligned}$$

As co and $sdnext$ are each $\in \{0, 1\}$, $sbnew$ can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = \overline{sdnext}$ and $prcarry_{n-2} = \overline{co}$.

For $sumq = -3$:

$q = -1$, and $B = |SD|$ with $b_{n-1} = 0$, $b_{n-2} = 1$

$$\begin{aligned} sbnew &= sumq \times 2^{n-2} + 1 \times 2^{n-2} + co \\ &= [-3 + 1 + co] \times 2^{n-2} \\ &= [-2 + co] \times 2^{n-2}. \end{aligned}$$

$sbnew$ can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = 1$ and $prcarry_{n-2} = \overline{co}$.

For $sumq = -2$:

$q = -1$, and $B = |SD|$ with $b_{n-1} = 0$, $b_{n-2} = 1$

$$\begin{aligned} sbnew &= sumq \times 2^{n-2} + 1 \times 2^{n-2} + co \\ &= [-2 + 1 + co] \times 2^{n-2} \\ &= [-1 + co] \times 2^{n-2}. \end{aligned}$$

$sbnew$ can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = 0$ and $prcarry_{n-2} = \overline{co}$.

For $sumq = -1$:

$q = 0$, and $B = 0$ with $b_{n-1} = 0$, $b_{n-2} = 0$

$$sbnew = sumq \times 2^{n-2} + co$$

$$= [-1 + co] \times 2^{n-2}.$$

sbnew can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = 0$ and $prcarry_{n-2} = \overline{co}$.

For $sumq = 0$:

$q = 1$, $B = -|SD|$, with $b_{n-1} = 1$, $b_{n-2} = 0$

$$sbnew = sumq \times 2^{n-2} - 2^{n-1} + co$$

$$= [-2 + co] \times 2^{n-2}$$

$$= [-2 + co] \times 2^{n-2}.$$

sbnew can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = 1$ and $prcarry_{n-2} = \overline{co}$.

For $sumq = 1$:

$q = 1$, $B = -|SD|$, with $b_{n-1} = 1$, $b_{n-2} = 0$

$$sbnew = sumq \times 2^{n-2} - 2^{n-1} + co$$

$$= [1 - 2 + co] \times 2^{n-2}$$

$$= [-1 + co] \times 2^{n-2}.$$

sbnew can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = 0$ and $prcarry_{n-2} = \overline{co}$.

For $sumq = 2$:

$q = 2$, $B = -2 \times |SD|$, with $b_n = 1$, $b_{n-1} = 0$, $b_{n-2} = 1 - sdnext$

$$sbnew = sumq \times 2^{n-2} - 2^(n) + (1 - sdnext) \times 2^{n-2} + co$$

$$= [2 - 4 + 1 - sdnext + co] \times 2^{n-2}$$

$$= [-1 - sdnext + co] \times 2^{n-2}.$$

sbnew can be apportioned to the new sign bits with the following assignments: $prsum_{n-2} = sdnext$ and $prcarry_{n-2} = \overline{co}$.

These assignments to the new sign bits are not unique, but do correctly apportion *sbnew*. The sign bit of *prcarry*, $prcarry_{n-2}$, is equal to $\overline{co}$ for every case of *sumq*. The results for the sign bit of *prsum*, $prsum_{n-2}$, are summarized in Table 4.3-1.

sumq	-4	-3	-2	-1	0	1	2
quotient	-2	-1	-1	0	1	1	2
sign of prsum	$\overline{\text{sdnext}}$	1	0	0	1	0	sdnext

Table 4.3-1: New Sign Bit

Thus, with an n-bit $|SD^0| \geq 2^{n-2}$, as ensured by the initial normalization of the divisor, a quotient value ranging from -2 to +2 can be chosen, based on just two bits of *prsum* and *prcarry*, which allows compression of any n-bit *prsum* and *prcarry* into $n-1$ bit registers after execution of Equation 3. By induction, at each iteration, i, the $n-i$ bit numbers *prsum* and *prcarry* can be reduced to $n-i-1$ bit numbers using an $n-i$ bit shifted divisor, SD^i, where $|SD^i| \geq 2^{n-i-2}$.

SD is reduced from its normalized n-bit form by one bit with each iteration until the final iteration, when SD is equal to the $n-m$ bit divisor, D, with a range as given in Equation 4.

After the mth and final iteration, $prsum$ and $prcarry$ are each $n-m-1$ bit 2's complement numbers and therefore:

$$-2^{n-m-2} \leq prsum \leq 2^{n-m-2} - 1 \tag{6}$$

and

$$-2^{n-m-2} \leq prcarry \leq 2^{n-m-2} - 1 \tag{7}$$

Combining Equations 6and 7with 4, the following ranges can be obtained for the final partial remainder:

$$-2^{n-m-1} \leq prsum + prcarry \leq 2^{n-m-1} - 2 \tag{8}$$

$$-2|D| \leq -2^{n-m-1} \leq prsum + prcarry \leq 2^{n-m-1} - 2 < 2|D| \qquad (9).$$

Since the final remainder, R, must have the same sign as the dividend, N, and $|R| < |D|$, the range of the partial remainder shows that an adjustment of at most twice the divisor at Step 3 obtains R.

4.4 Comparison with Other Dividers

This section will show that implementation considerations favor this approach over both PP and PV.

4.4.1 Comparison with PP

The proposed algorithm is more flexible than PP in that it permits division by and of negative numbers, while PP is restricted to non-negative dividends and positive divisors. At each addition, the algorithm must process a total of 4 sum and carry bits to generate the three-bit quotient, while PP looks at just 2 bits to generate 1 of 3 parts of the quotient. The additional processing for the quotient bit has, as n gets large, a negligible effect on area because the quotient is generated only once, regardless of the size of the divider. As for delay, there is a cost in generating from more inputs and processing from more outputs, but this delay is more than offset by the two more additions and quotient selections required for each iteration of PP. PP also discards the quotient values used to generate the remainder, and computes the quotient after the remainder in approximately n steps, while the new divider is more suited to generate the quotient in parallel with the remainder. The result is that the remainder and the quotient are computed more than 2 times faster. The reduction in adders results in an area reduction by more than a factor of 2 over PP. The $O(n)$ area of the dividers translates this decrease in area into an ability to divide numbers of twice as many bits.

4.4.2 Comparison with PV

The comparison with PV dividers will be restricted to the least redundant version because greater redundancy can improve delay only at a significant cost of area and complexity [1]. In the minimum-area implementation of PV, the addition operation is similar to carry-save arithmetic, but the operands at each digit are allowed a greater range: $s_j \in \{-1, 0\}, c_j \in \{-1, 1\}, d_j \in \{-1, 0\}, q_j \in \{-2, -1, 0, 1, 2\}$ [2]. Since each bit slice of the adder is to implement $< s_j, c_j > := s_j + c_j - q_j \times dj$, each bit slice of the redundant adder must receive a minimum of 1 bit for the s_j

input, 2 bits for the c_j input, and 2 bits for the $d_j \times q_j$. Each bit slice of the adder must then output and latch at least 1 bit for the s_j output and 2 for bits for the c_j output. The standard carry-save adder must process only 3 bits of input and latch only 2 bits of output, and therefore requires less area and less delay. The quotient selection in PV also compares unfavorably. It is more complex and is based on the first 5 bits of the divisor and 8 bits of the partial remainder [2]. Despite the greater complexity and area of PV, it has a significant delay advantage over PP because the remainder is computed with approximately n additions rather than the $3 \times n$ required with PP. The new divider, like PV, computes the remainder in approximately n additions, but the n additions performed require less delay and less area.

Chapter 5 Architecture of the Divider

5.1 Introduction

This chapter contains the main structural information for a 16-bit synchronous implementation of the algorithm.

The overall architecture is shown in Figure 5.1-1, a diagram of the main modules that shows where the main operands of the algorithm are being input and output. The divisor is received by the input module. The input module outputs the shifted divisor to the pr_adder module which receives the quotient bits from the controller and the dividend that is input to the chip. The pr_adder outputs the remainder. The q_adder module receives the quotient bits generated at each iteration in order to finally output the total quotient. The controller generates the

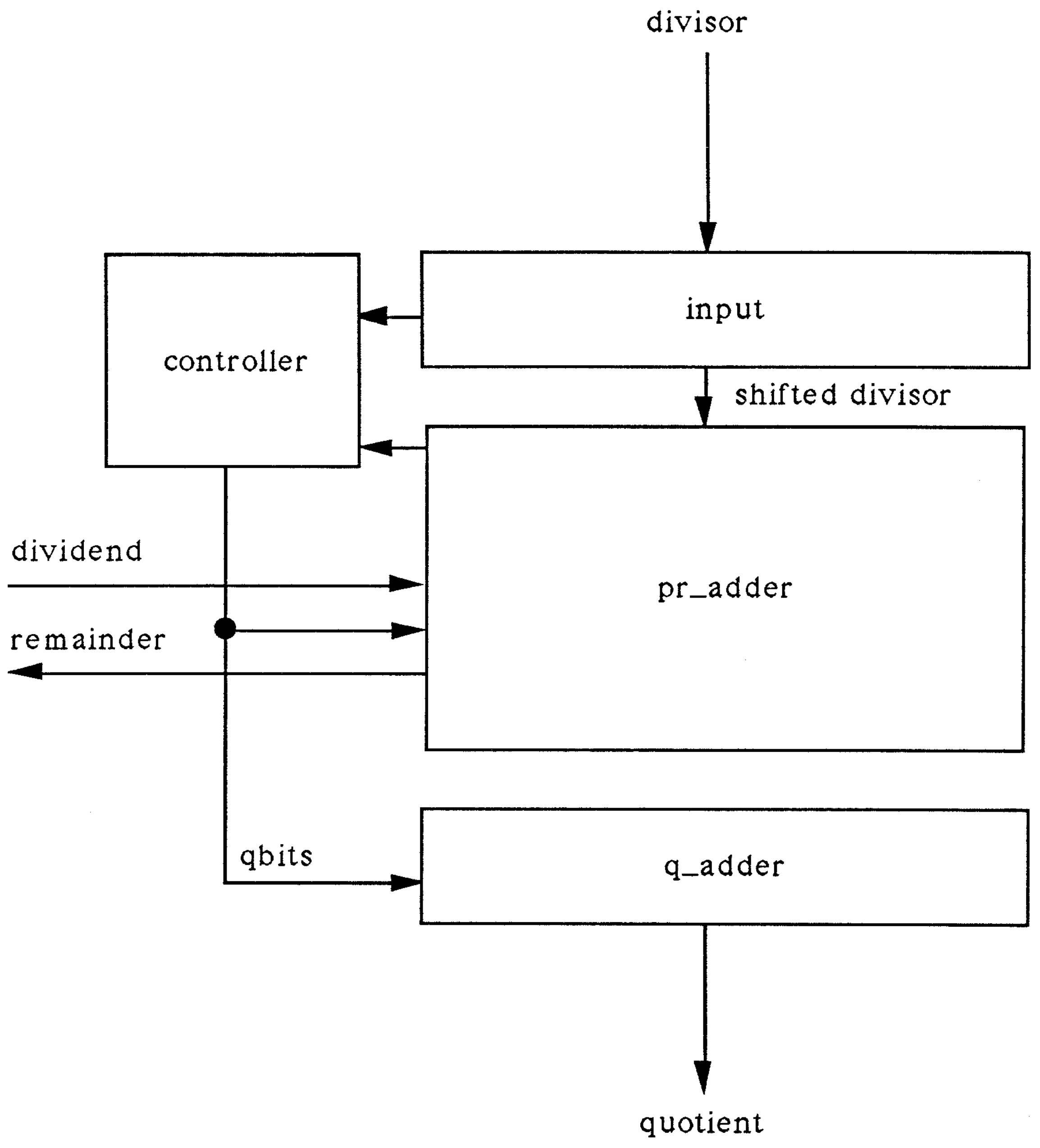

Figure 5.1-1: Divider Architecture.

quotient bits from unmarked inputs from the input module and pr_adder. It also controls which kind of addition, carry-save or ripple carry, is being performed by pr_adder and q_adder. Subsequent sections will describe the structure and functions of input, pr_adder, and q_adder in more detail.

5.2 Normalization Circuitry

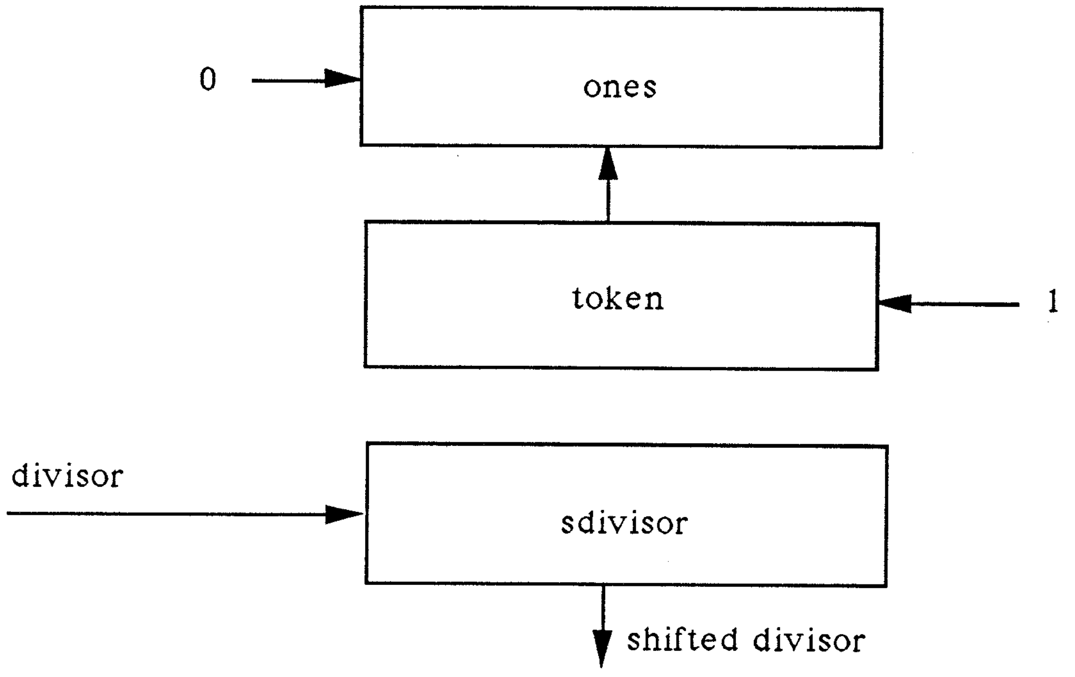

Figure 5.2-2: Input Module.

The n-bit divisor must be shifted over until its shifted version, SD, has an absolute value of at least 2^{n-2}. This is implemented with a shift register initially loaded with the divisor and then shifted left as many times as necessary. The number of shifts that are required is stored in an accompanying shift register, called token, which is initially loaded with a one and is shifted left with the divisor with ones being loaded into the left. After the shift is completed, the value in token, containing $m+1$ ones, is loaded into another register called ones. As the central loop implementing the carry-save additions is performed, the ones register will be shifted to the right (with zeros being input at the left) until it contains only a single one, signalling the end of the main loop. The token register, containing $n - (m + 1)$ zeros, is used to store the final size of the partial remainder sum and carry registers, prsum and prcarry. Obviously, both the token register and the ones register could have been implemented with $\log n$-size counters, but n-bit registers were used instead to maintain the bit slice nature of the architecture and to lower complexity. These three n-bit registers and the multiplexors associated with the inputs to each bit form the $O(n)$-size module shown in Figure 5.2-2 . The schematic diagram of the bit slice of this module is shown in Figure 5.2-3 ; an n size array of these, with only nearest neighbor connections, forms the input module.

The carry input to each bitslice allows the contents of prcarry to be loaded into the sdivisor register, needed for the final ripple carry addition of the remainder.

5.3 Carry-Save Adder

The standard n-bit, carry-save adder is a linear array of n full adders with n latches for the output sum bits and n latches for the output carry bits. The standard configuration is to have the latched sum fed back as input to its accompanying full adder and the latched carry used as input to the next full adder to the left. Using a standard carry-save adder would require complicated n-dependent, select circuitry to find the current, high-order bits of prsum and prcarry, since quotient bits are determined by, and truncation circuitry must act on, the $(n-i)$th bits of the prsum and prcarry (i being the current iteration).

This complicated circuitry is avoided by propagating the results of each addition to the left one bit. This can be done without losing data, because the new prsum and prcarry are one bit smaller with each iteration.

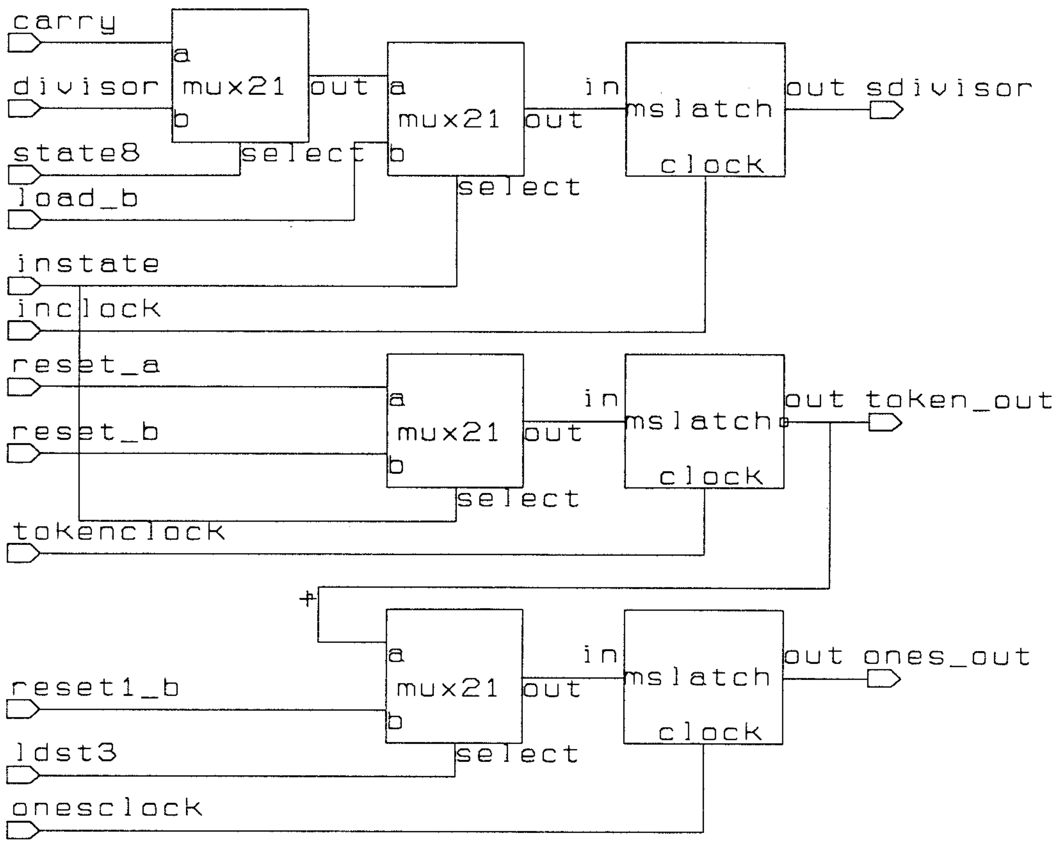

Figure 5.2-3: Bit Slice of Normalization Circuitry

A three-bit quotient determines which multiple of the shifted divisor is to be added to (or subtracted from) the current partial remainder. When the quotient is +2 or -2 and twice the shifted divisor is to be subtracted/added, the inputs to the full adders are $sd \times 2$, or sd shifted left one bit. A multiplexor is used to determine whether the potential input to bit j of the adder is sd_j or sd_{j-1}. Another multiplexor determines whether $+sd$, $-sd$, or 0 is added to the partial remainder.

A multiplexor whose output determines the sum input to each full adder allows an initial parallel load of the sum register with the dividend, and subsequent inputs from the latched output of the full adder to the right.

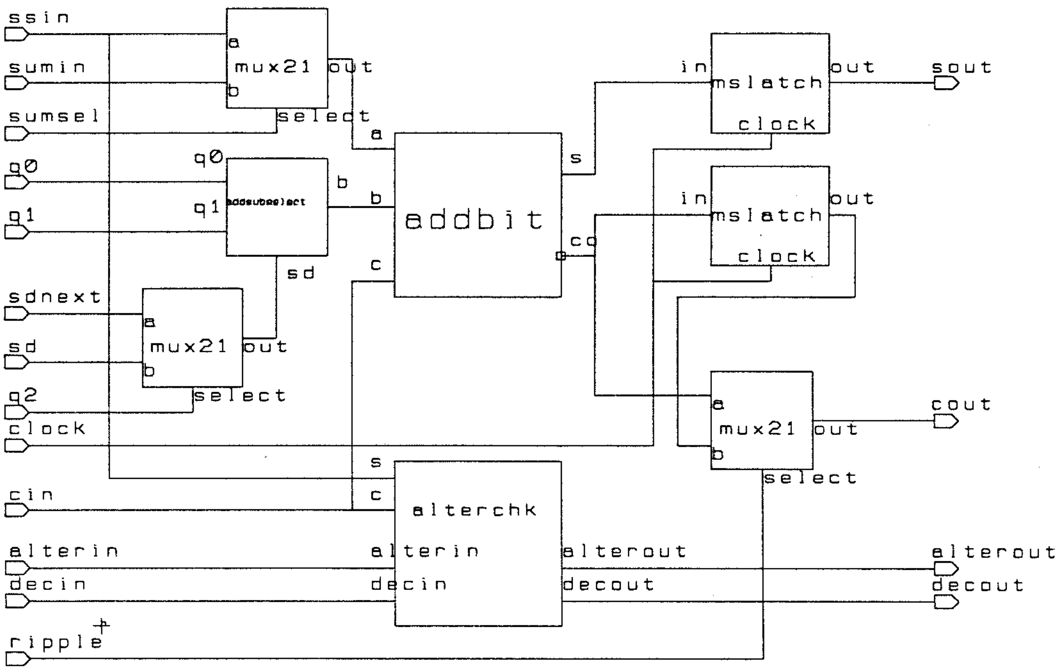

Figure 5.3-4: Bit Slice of pr_adder

The final step of the algorithm requires a standard, carry-propagating addition. The same full adders used for the carry-save additions can be used for the carry-propagating addition when a multiplexor is used to determine whether the carryin input to each full adder is the carryout of the full adder to the right or the latched carry of the full adder, two to the right. The additional multiplexor on each carry input reduces area by avoiding n additional full adders (a 2:1 multiplexor has significantly less area than a full adder) and associated latches at no delay cost, because there is already a delay on the inputs to each full adder by the two multiplexors required to process the quotient bits.

The schematic diagram of the bit slice of this is shown in Figure 5.3-4 . As can be seen from Figure 5.3-4 the bit slice contains one full adder, four multiplexors, and two latches. The additional logic pictured is used for comparison after the main loop. The computational delay is that of two multiplexors and one full adder. The critical path delay of pr_adder during the main loop is through two multiplexors, one full adder, and a final inverter for the sign bit of the carry register, as discussed in Chapter 4.

5.4 Quotient Generation

At each iteration i, the quotient q_i represents a $q_i \times 2^{m-1-i}$ addition to the total quotient, Q. This is implemented at each iteration by shifting the old quotient one bit to the left (multiplying by two) and adding q_i.

$$Q_{-1} = 0$$

$$Q_i = 2 \times Q_{i-1} + q_i$$

After m iterations, Q is given by the following equation:

$$Q_m = q_0 \times 2^{m-1} + q_1 \times 2^{m-2} + ... + q_{m-1} \times 2^0. \tag{10}$$

The quotient update is implemented with a carry-save adder whose output is shifted one bit to the left (as in the partial remainder update) with the B input to the adder being determined by the current quotient, q_i. The n-bit 2's complement of q_i is given by 111...11110, 111...11111, 000...00000, 000...00001, or 000...00010 . Table 5.4-1 gives the three-bit mapping of q_i into q_2, q_1, and q_0. The b input to each full adder, except the rightmost two, is therefore non-zero only when q_i is less than 0. It can be seen from the table that b can therefore be set equal to q_1. The least significant b input, b_0, is non-zero only for $q_i = -1$ or 1, and is given by $\overline{q_2} \wedge (q_1 \vee q_0)$. The next to last b, b_1, is non-zero for $q = -2$, $q = -1$, or $q = 1$; and is given by $q_1 \vee (q_0 \wedge \overline{q_2})$.

The additional circuitry to implement b_0 and b_1 does not add to the latency of the divider because q_adder is being updated at the same time as pr_adder, which requires more delay for its inputs.

qi	q2	q1	q0
-2	1	1	0
-1	0	1	0
0	0	0	0
1	0	0	1
2	1	0	1

Table 5.4-1

The resulting architecture of a bit slice of the q_adder is shown in Figure 5.4-5 .

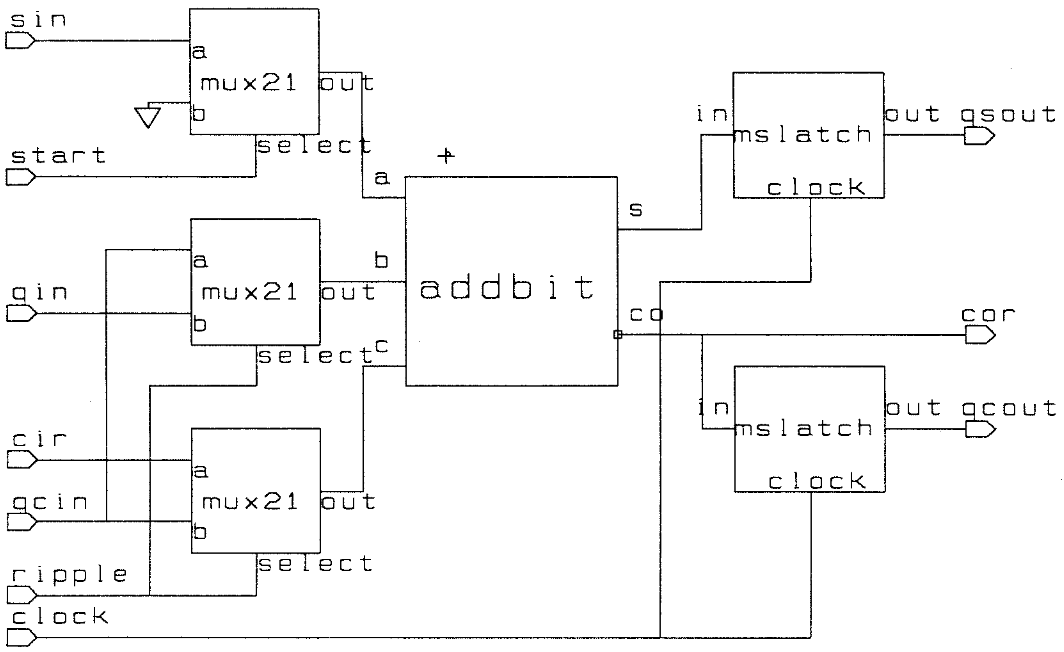

Figure 5.4-5: Bit Slice of q_adder.

Chapter 6 Faults in Asynchronous Arrays

6.1 Introduction

Architectures using four-state coding, a data-driven technique for implementing bit-level wave-front arrays and their responses to various faults are described. They are shown to be resistant to soft and hard error propagation. This resistance to propagation of errors is the basis of a simplified approach to fault tolerance not possible with conventional, clocked systolic arrays. In Chapter 7, architectures that implement this approach will be presented.

6.2 Four-State Coding

With clockless (self-timed, or asynchronous) wave-front arrays, the timing is controlled by the elements themselves, not by a common clock [10]. In order to keep computations ordered, it is necessary to employ more than two states to represent a single bit of information. Four-state, asynchronous communication code employs two states to represent logical 1 (P1 and Q1) and two states for logical 0 (P0 and Q0), where P and Q can be thought of as two phase representations. These states can be coded in binary logic using two bits: 00 = logical 0 (P0), 01 = logical 1 (Q1), 10 = logical 0 (Q0), and 11 = logical 1 (P1). If outputs and inputs alternate between the P and Q phase representations, there is always exactly one bit changed for every transition in output or input, avoiding the problem of races when the output changes [11].

The major drawback of four-state coding is area (true for asynchronous designs in general); but it claims three important advantages over the equivalent synchronous systolic array: faster throughput, reduced design complexity and greater reliability. Four-state asynchronous coding will be demonstrated using a simple, serial shift register.

Figure 6.2-1 and Figure 6.2-2 show three stages in the middle of a longer shift register. It consists of three asynchronous delay cells (D-cells), whose truth table is shown in Table 6.2-1. Of the 8 possible input states, half change the output state and the other half leave it in its existing state. The D-cell passes Q-phase data only when the next element has P-phase data, and passes P-phase data only when the next element has Q-phase data. Each cell in the shift register will alternate between P and Q phase as stored values are shifted through from left to right. Figure 6.2-2 shows a simplified view of Figure 6.2-1, the type of view that will be used in the rest of this chapter. Here, each line represents three wires: two for the flow of

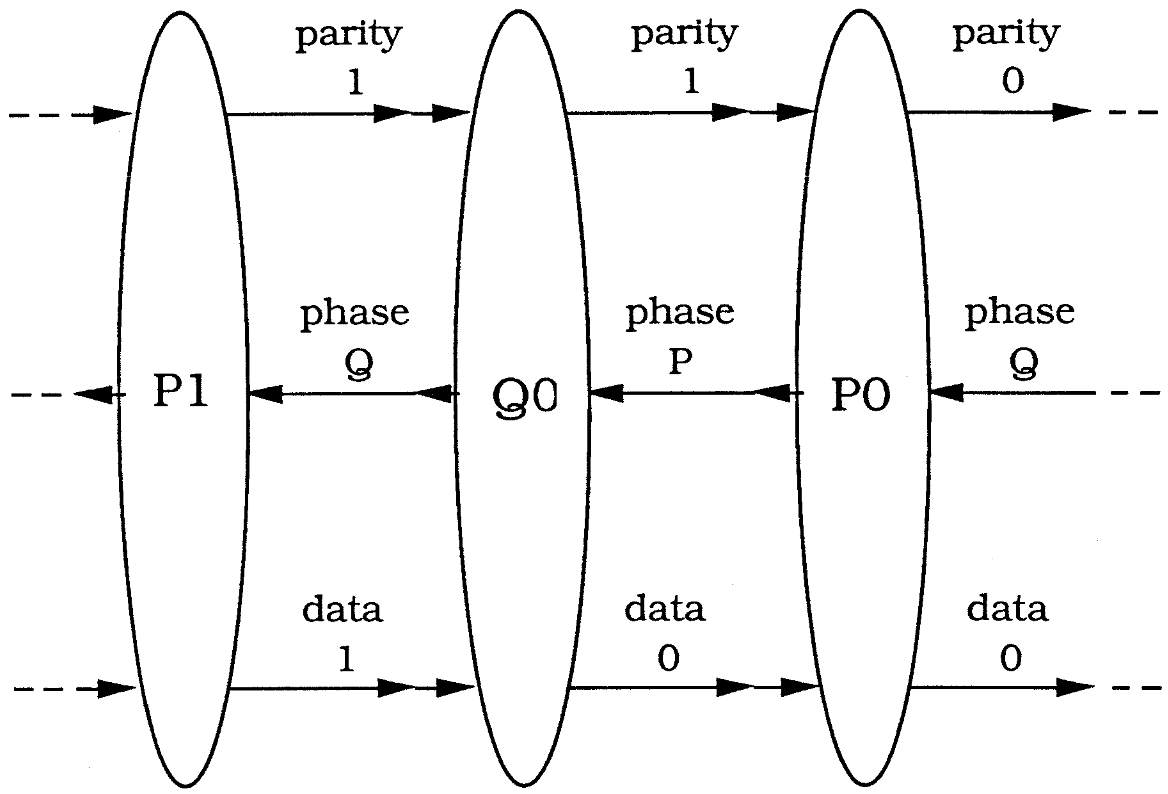

Figure 6.2-1: Four-State Shift Register.

information (P0, Q0, P1 or Q1) going in the direction of the arrow and one for the phase acknowledgment (P or Q) going in the opposite direction.

6.3 Faults in MOS Circuits and Their Detection

The need for increased processing speeds (in communications, defense, complex modeling and real-time image processing) is coupled with a need for increased reliability. For many applications, processing errors in the individual processing elements must be detected and corrected in order to provide assurance that the results are correct.

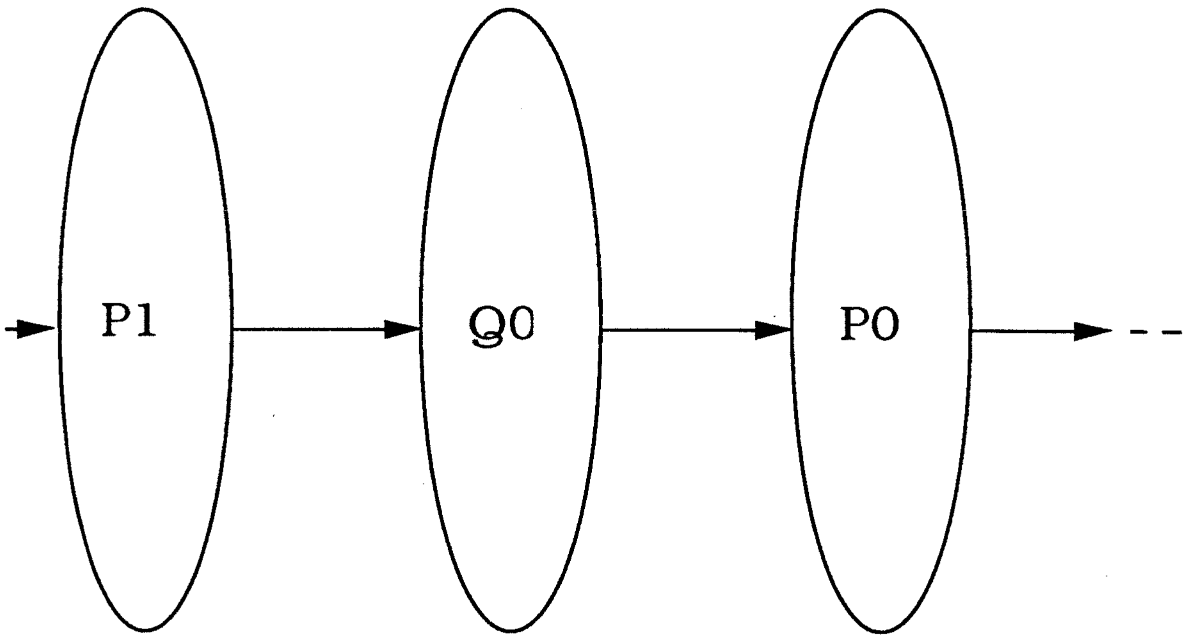

Figure 6.2-2: Four-State Shift Register - Simplified View.

On-line detection of both permanent (or hard) errors and temporary (or soft) errors is necessary in order to provide fault-tolerant computing [12]. The continuing decrease in the minimum feature sizes has tended to increase the severity and frequency of both hard and soft errors [13]. The impact of failures upon the validity of the circuit operation may vary from none to catastrophic. While hard errors may seriously affect system performance, soft errors tend to be more difficult to detect and fix, potentially causing more harm in an application. It has been estimated that in most systems 80-90% of physical faults are soft [14]. It is in mitigating the effects of these faults that asynchronous logic can have an important impact.

Input phase and data	Acknowledge Phase	Output phase and data
00 (P0)	P	No change
00 (P0)	Q	00 (P0)
01 (Q1)	P	01 (Q1)
01 (Q1)	Q	No change
10 (Q0)	P	10 (Q0)
10 (Q0)	Q	No change
11 (P1)	P	No change
11 (P1)	Q	11 (P1)

Table 6.2-1: Four-State, Shift-Register Truth Table

6.3.1 Transient faults or soft errors

Environmental factors, such as electromagnetic interference and radiation, cause all circuits to receive undesirable charges in the form of alpha particles and cosmic rays [12]. The soft-error rate in MOS circuits is almost exponentially dependent on Q_{crit}, the critical charge necessary to change the logic value [15]. As processes are scaled Q_{crit} tends to decrease by the square of the scale factor. The decrease in Q_{crit} more than compensates for any improvement in hit probability caused by smaller devices (assuming device area scales by the square of the minimum-feature size). If the improved density is used to make larger arrays, keeping the active area unchanged, then scaling down by a factor of 2 increases the soft-error rate by a factor of between 2 and 4 [15].

6.3.2 Detecting and correcting soft errors

The most straightforward approach to detecting and correcting soft errors is replication. It is applicable to most circuits; and, although it costs a great deal of area, it is not very complex to implement. Triplication, like triple-repetition coding, enables single-error correction. An erroneous output from one circuit will be corrected if the other two circuits are outputting the correct value. With duplication, only error detection can be done, since there is no way to decide correctly between different outputs. The ability to mark one circuit as having gone wrong can, like an erasure in a communication channel, allow for single-error correction. The result would be error correction with only two-thirds the area required by triplication. Asynchronous logic can be made to output erasures rather than errors allowing for error correction with duplication.

More sophisticated detecting schemes have been employed in some applications [16]. Examples of this include single-error correcting and double-error detecting codes for memories, parity bits for data buses, residue codes for ALU's, watchdogs and redundant links in switches [17] [18] [19] [20]. In general, a circuit is self-testing for a set of faults F, if for every fault in F, the circuit produces a non-code output [21] [22]. Although these schemes are more area-efficient, they lack the simplicity and generality of replication coding.

6.4 Faults in a Four-State Shift Register

Section 6.2 described how a four-state shift register operates in the absence of faults. Although the resulting shift register is functionally similar to a clocked shift register, it is affected by faults very differently. In particular, the asynchronous shift register can react to an error by either "sticking" or "slipping."

6.4.1 Sticking

Figure 6.4.1-3 illustrates how a hard error could cause the register to "stick." The marked cell in Figure 6.4.1-3 has a permanent fault, causing the data bit to be "stuck-at-zero." There are no problems as long as this bit is not supposed to change; but when P1 is input as in Figure 6.4.1-3 IV, the cell cannot acknowledge the data, and no further data can be received from, or sent to, this cell. Although sticking may appear to be undesirable behavior, it can, in fact, act as error detection. Regular circuit behavior would result in a parity transition, a new output, within some reasonable interval of time. Here, there is no such transition, so it can be recognized that no new output has been received from this circuit. In effect, the circuit has output an erasure. Circuit duplication would result in a repetition code of length two. Duplication, not triplication, is thus sufficient for single-error correction, as only erasure correction is required.

6.4.2 Slipping

Figure 6.4.2-4 illustrates how a soft error could cause the register to "slip." The correct shift-register contents of Figure 6.4.2-4 I are changed by a "hit" into those of Figure 6.4.2-4 II. Now the cells to the left of the error, because of the new phase acknowledgement, think that the data they sent had been received, when

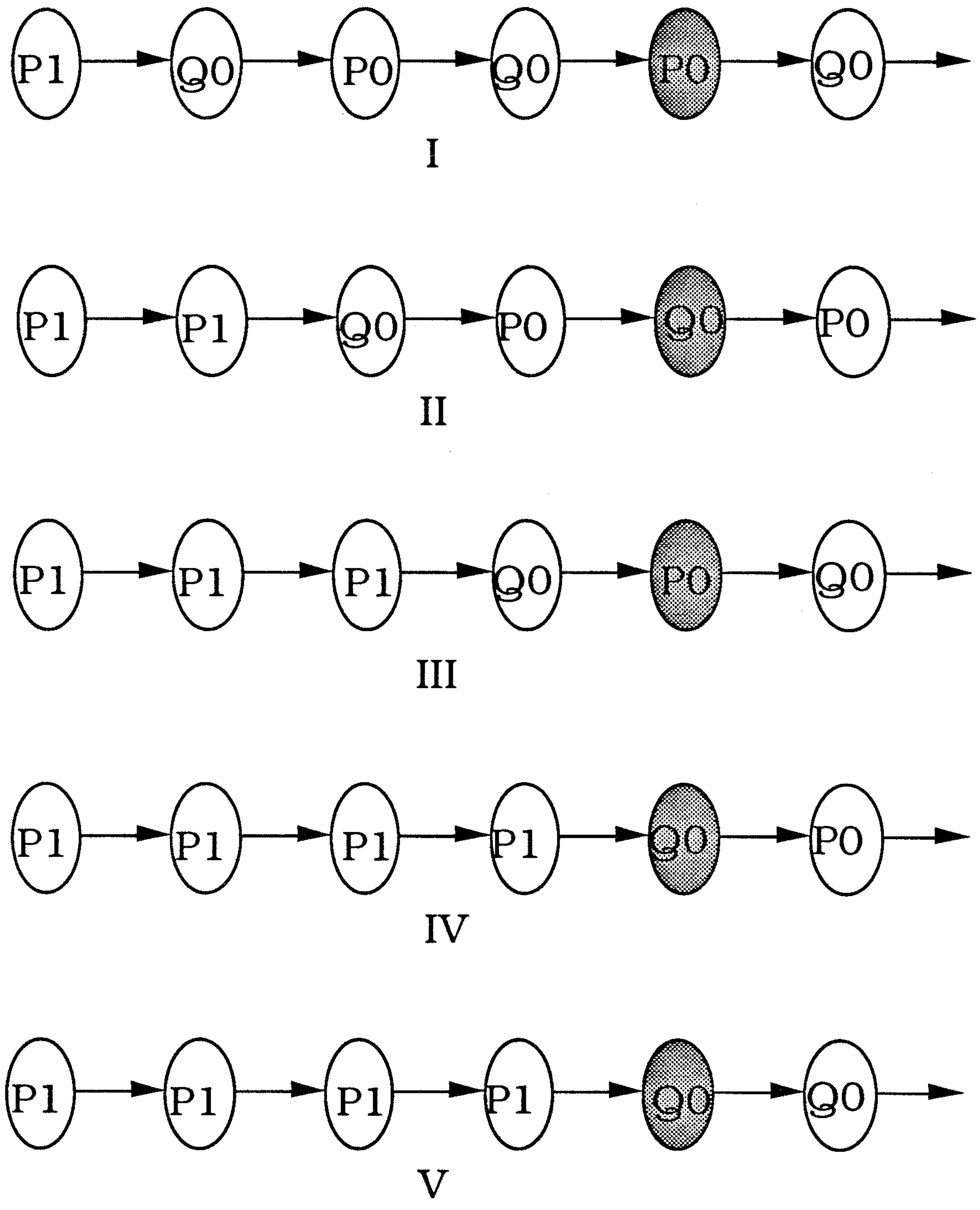

Figure 6.4.1-3: Shift Register with a Hard Error.

actually it had not; and the the cells to the right would never see the data that had

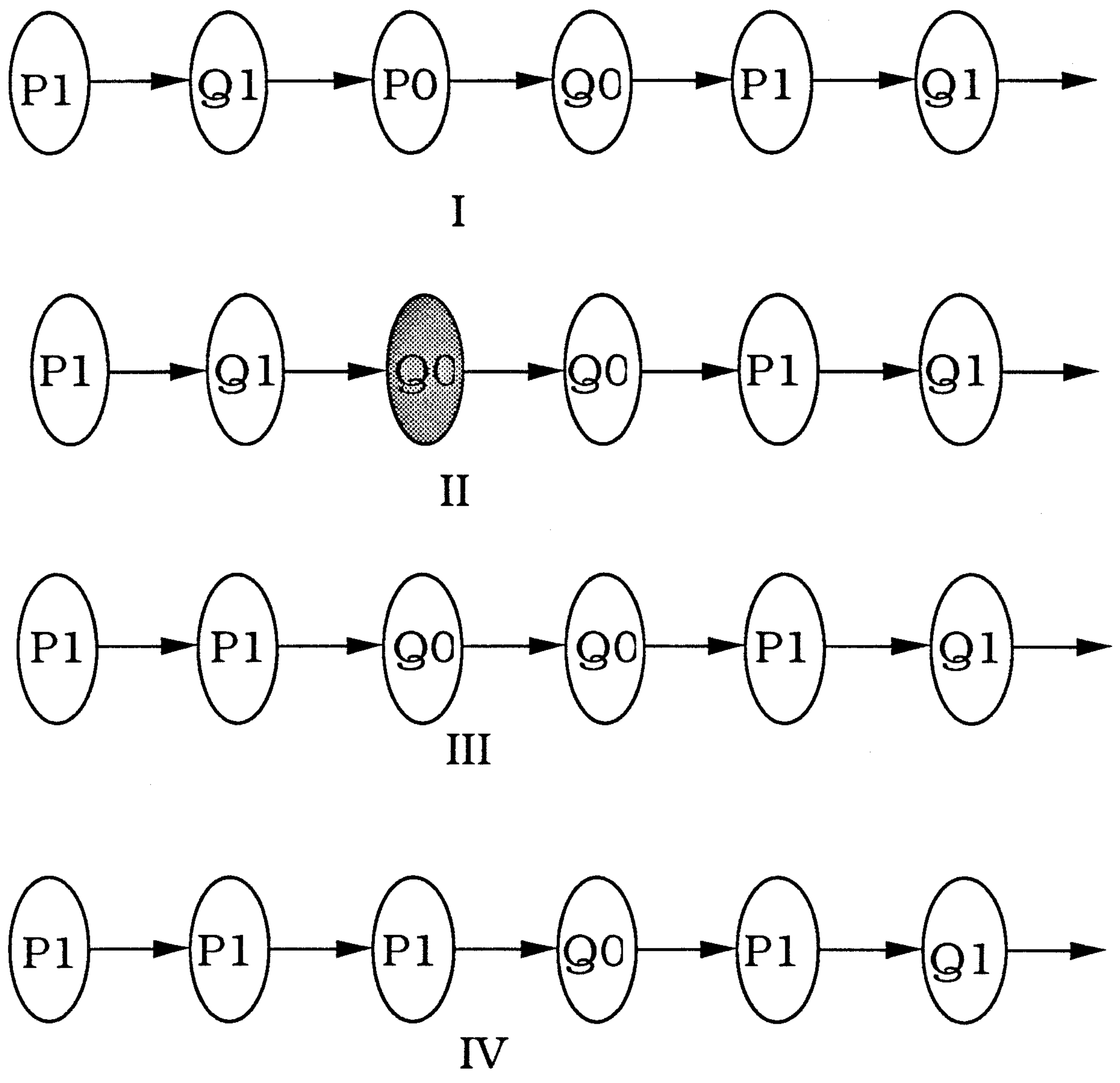

Figure 6.4.2-4: Shift Register with a Soft Error.

been changed in error. The eventual result is shown in Figure 6.4.2-4 IV, with the leftmost P1 allowed to propagate two cells to the right, while the Q1 and P0 to its right were effectively erased. This behavior is likely to be catastrophic and needs to be avoided.

Chapter 7 Exploiting the Inherent Fault Tolerance of Asynchronous Arrays

7.1 Introduction

In this chapter, the responses of two different convolutional encoders are compared in order to demonstrate which types of asynchronous arrays will slip and which will stick in the presence of faults. Finally, the fault responses of more general four-state architectures are demonstrated.

7.2 Faults in a Four-State, Convolutional Encoder

An n-stage, convolutional encoder is a polynomial multiplier, where an input stream is multiplied by a fixed coefficient. The function is illustrated by Figure 7.2-

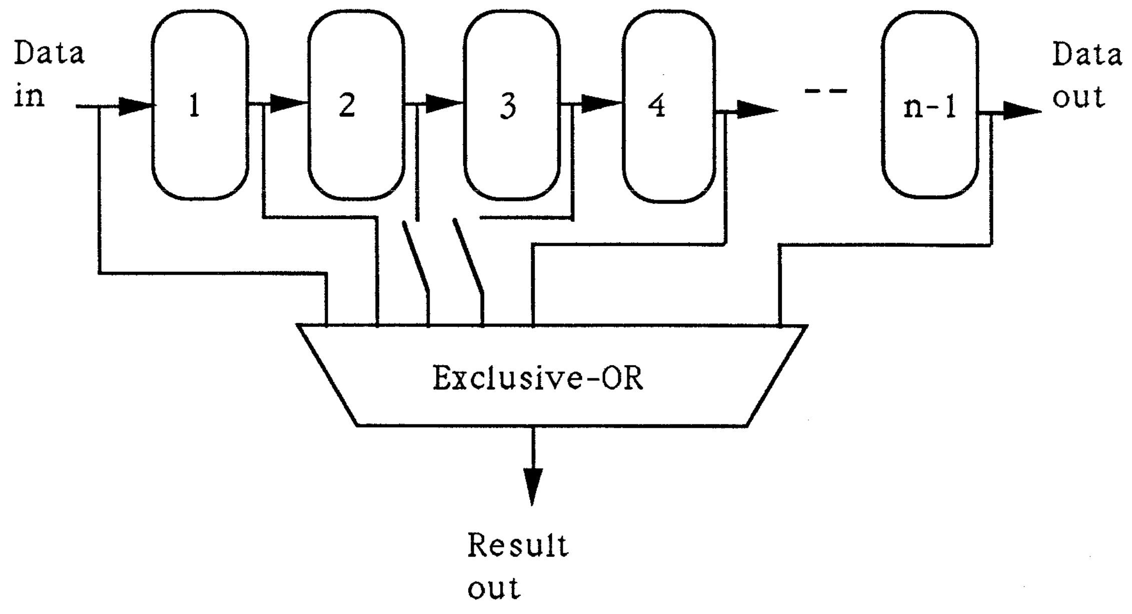

Figure 7.2-1: Convolutional Encoder.

1, where the input stream is stored in latches, and the coefficient associated with each latch determines whether the latch output is included in the exclusive-OR operation. For large n the encoder would be built using a more regular systolic structure, consisting of n identical elements chained together. Two such elements, each employing four-state cells, are shown in Figure 7.2-2 and Figure 7.2-3 . It is instructive to consider these two functionally equivalent architectures, since they exhibit different behavior under error conditions. First, a fault-prone architecture will be discussed, and then an alternate fault-resistant architecture will be described.

7.2.1 Version 1 - a fault-prone, four-state convolutional encoder architecture

Three elements from a chain that make up a convolutional encoder are shown in Figure 7.2.1-2 . It is a fairly straightforward implementation, optimum in terms of the throughput achievable. The top latch performs the same function as a latch

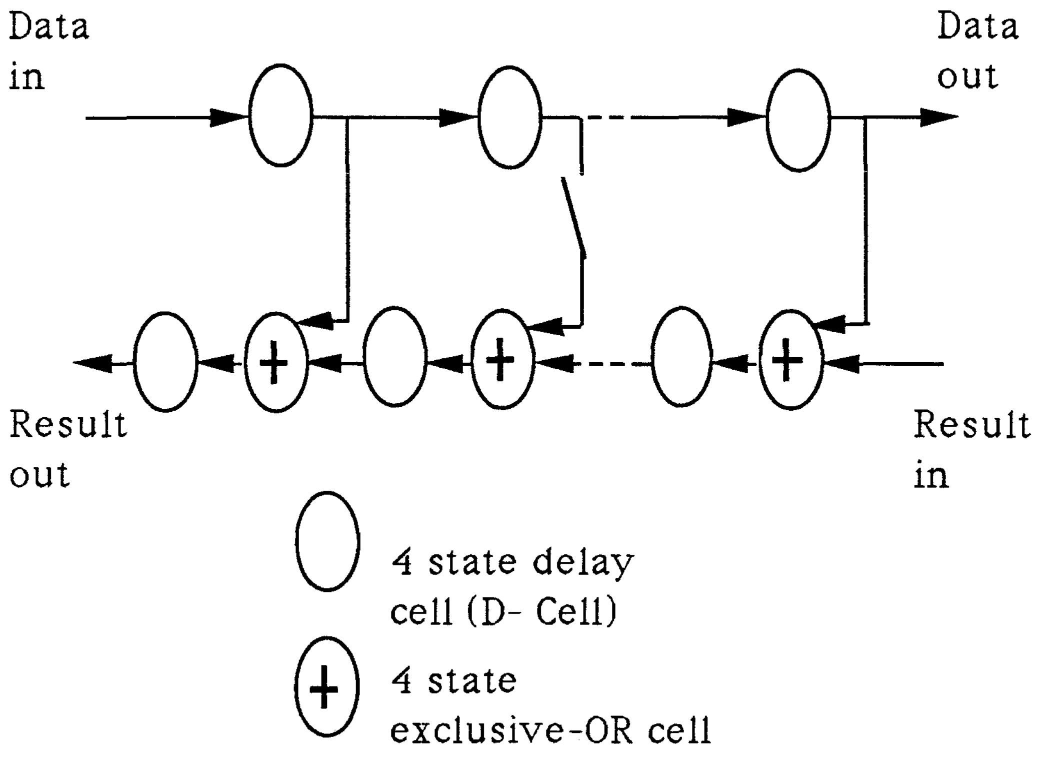

Figure 7.2.1-2: Convolutional Encoder - Version 1.

in Figure 7.2.1-2, delaying the input as it passes from left to right. The bottom cell on the right-hand side performs part of the overall exclusive-OR operation as the partial result is passed from right to left. The final latch in the bottom left is employed to improve throughput. Thus, the top cells act as a shift register, with each cell receiving its inputs from the previous element, and outputting its information both to the next element and to the exclusive-OR cell. The bottom cells act to accumulate the result, with the right-hand cell doing the actual convolution and the left-hand cell acting as a buffer.

If one of the cells that make up this convolutional encoder receives enough radiation to change its output spontaneously, its behavior will depend on the state

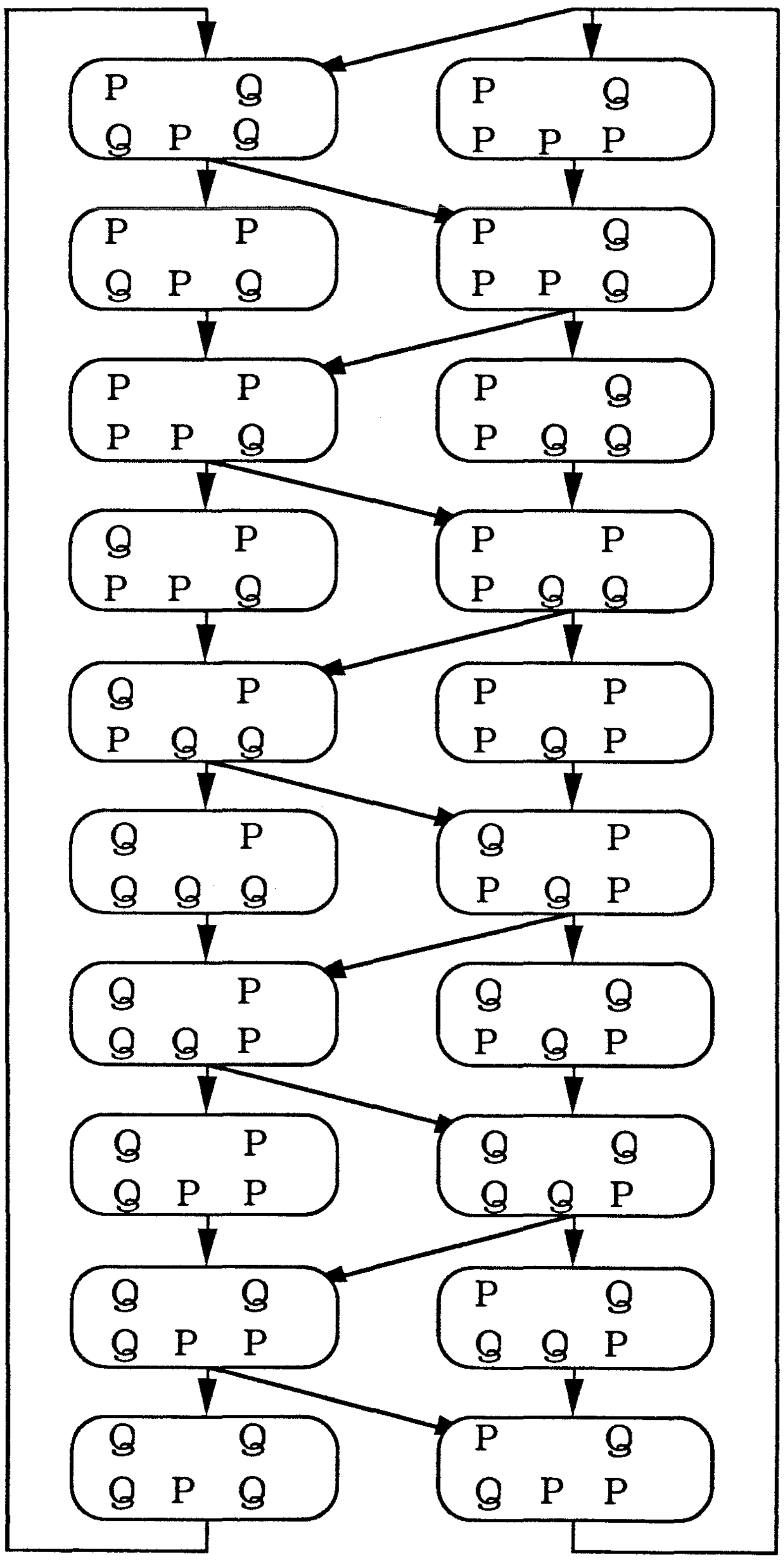

Figure 7.2.1-4: Version 1 without Errors.

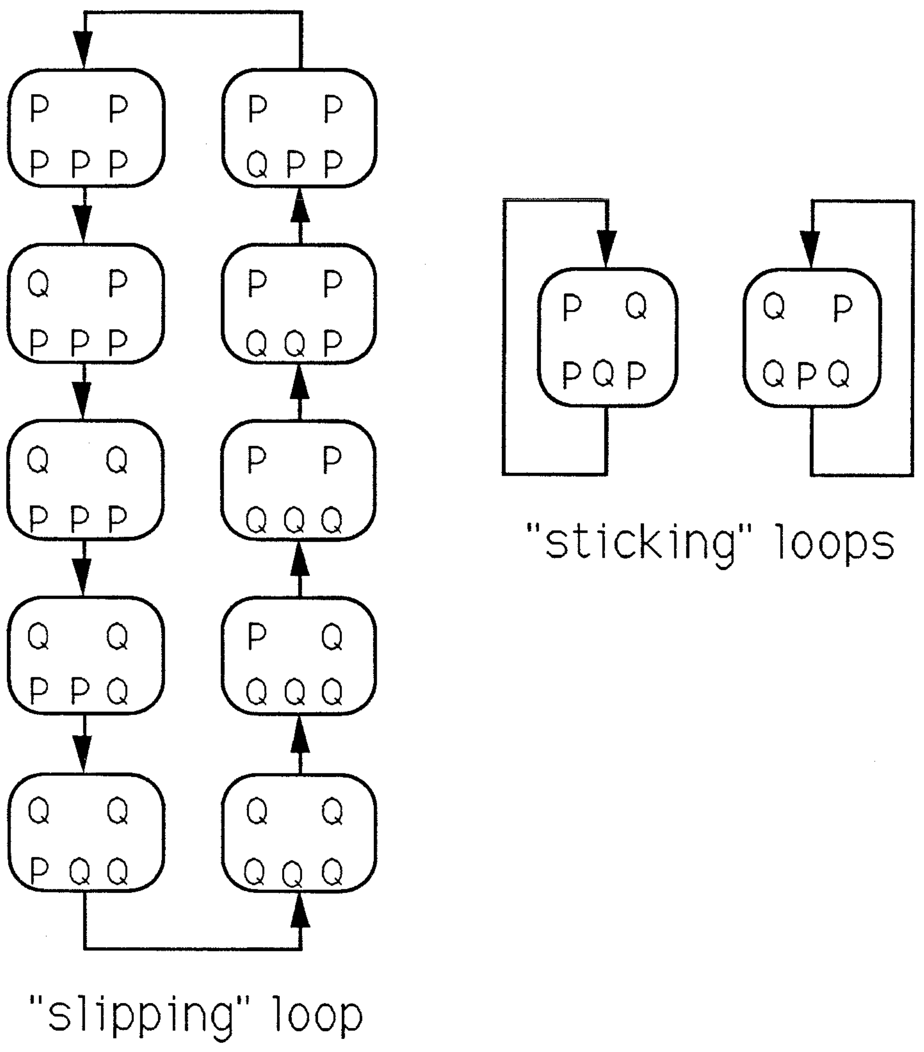

Figure 7.2.1-4: Version 1 with Errors.

of its neighbors. A single-bit error will change the phase (i.e., P to Q or Q to P) of the element receiving the error. In order to illustrate this point consider five of the cells in isolation. Specifically, pick three cells from an element and the two right-hand cells from the element to its left. Now these five cells will each be in one of four-states (P0, Q0, P1 or Q1); however, for this analysis just the phase, P or Q, will be noted. There will then be 25 possible phase states. If each cell obeys its truth table, then the legal transitions from a given state will be very limited. A convenient way of describing the possible sequences is by means of a de Bruijn diagram [23]. Figure 7.2.1-4 and Figure 7.2.1-5 show the resulting de Bruijn diagram for the five cells. It has four loops: two of size one (which will be termed the "sticking" loops), one of size ten (which will be called the "slipping" loop), and one of size 20 (the error free loop).

Assume initially that the encoder has been reset so that it is in the error-free loop. Then, provided there are no errors in the system, the five cells will remain in this loop. However, an error can cause the system to jump spontaneously into any possible state, including those of the "slipping" and "sticking" loops. If an error moves the system into either of the "sticking" loops, the system will come to a complete standstill and the error will be detected. But if an error moves the system into the "slipping" loop, two very undesirable, non-reversible events occur. First, two bits of data will be totally obliterated. This is easiest to see if the middle cell, initially in the Q state, was changed by radiation into the P state, and the remaining cells were already in the P state. Then, not only is the bit of Q phase information lost, but also one of the bits of phase P information to its left or right will be lost, since there is no longer any way of distinguishing them. The rest of the system will then have no way of knowing of this lost data, leaving the errors undetected. A second problem that occurs is that the new loop introduces a bottleneck. This "slipping" loop allows data to pass through at only half the rate of the "desired" loop.

Although this encoder is optimal in terms of throughput under normal conditions, it can exhibit some changes catastrophic to system function as a result of a soft error.

7.2.2 Version 2 - a fault-detecting convolutional encoder architecture

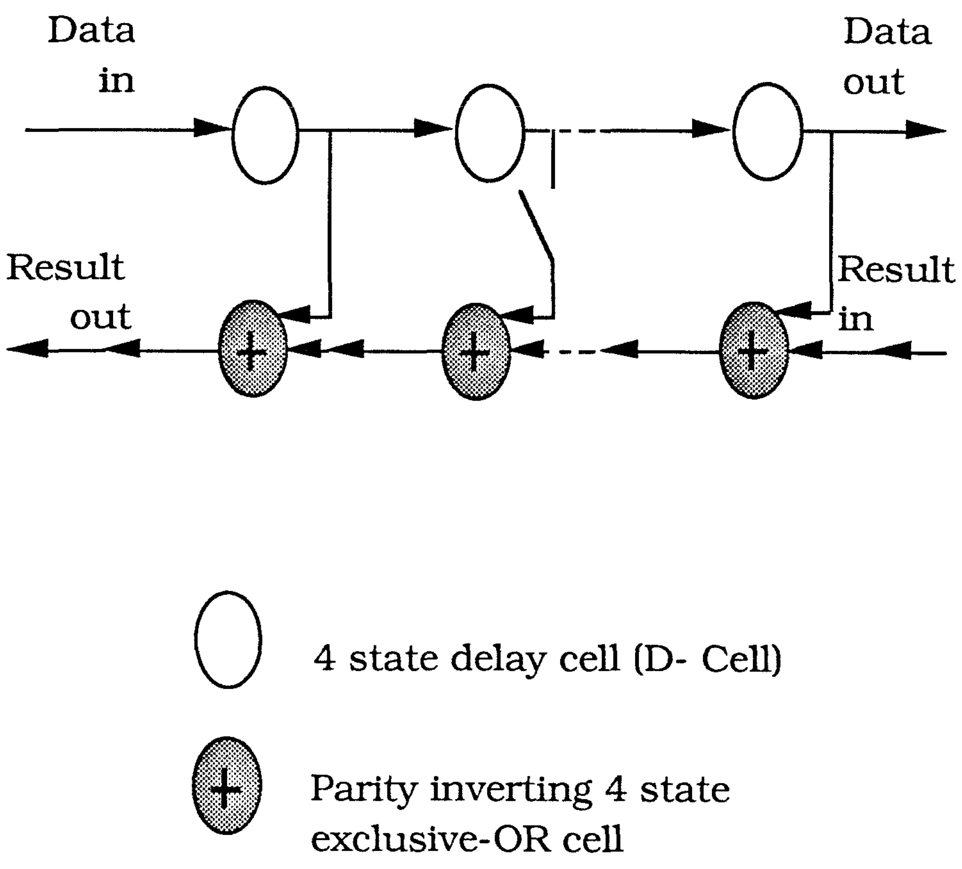

Figure 7.2.2-3: Convolutional Encoder - Version 2.

The convolutional encoder shown in Figure 7.2.2-3 makes use of a technique called phase inversion. That is, cell A will input information from cell B with the parity bit inverted, making cell B appear to cell A as if it were in the opposite phase from its true state; and cell B will see inverted versions of cell A's acknowledgment, so that it receives the acknowledgment it is expecting. In Figure 7.2.2-3, the exclusive-OR cell is marked in black because it uses phase inversion. This encoder works in a manner similar to the previous one and is functionally identical.

It, too, is optimal in terms of throughput and is actually more efficient in area, since it employs one less cell. Its response to soft errors is, however, very different.

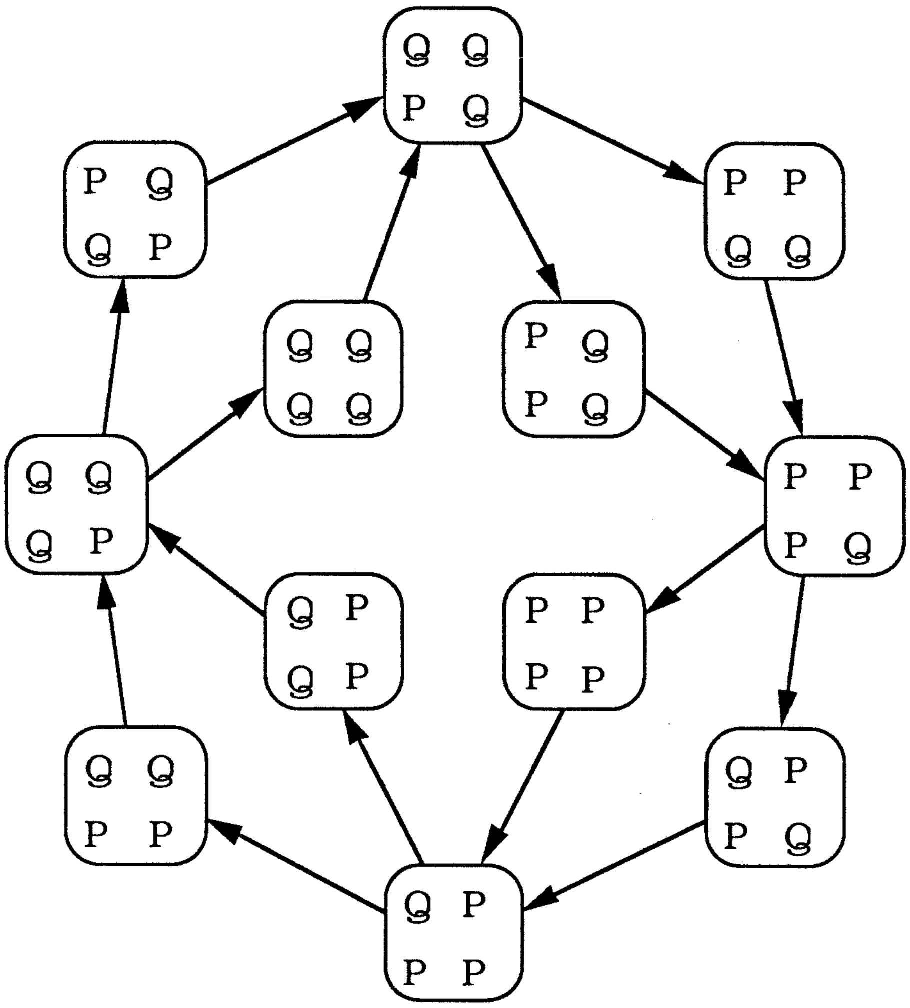

Figure 7.2.2-6: Version 2 without Errors.

A de Bruijn diagram for two elements of this encoder is shown in Figure 7.2.2-6 and Figure 7.2.2-7 . These diagrams show five loops: four of size one ("sticking" loops) and one of size twelve (error-free loop). Again assuming that the system is initialized to start in the error-free loop, then, provided there are no system errors,

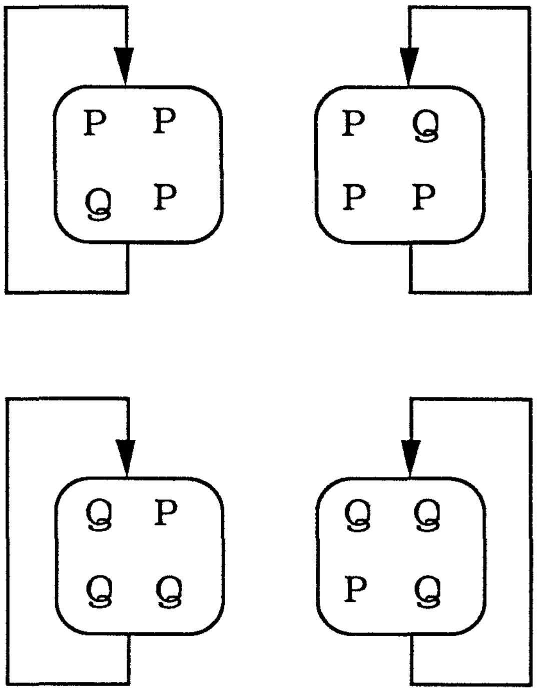

Figure 7.2.2-7: Version 2 with Errors.

it will remain permanently in this loop. With this encoder, unlike the previous one, errors can move the system only into a "sticking" loop.

This encoder is to be preferred because it does not exhibit the potentially catastrophic changes (slipping) as a result of soft errors that were exhibited by the previous architecture.

7.3 More General Architectures

Some general comments can be made regarding the relationship between architecture and error characteristics. First, slippage will always be possible when a cell is receiving from only one cell and outputting to only one cell. This results in the shift register case and is also true for the delay cell in the bottom left of Figure 7.3-8 . Any configuration in which cells receive input from only one cell and output to cells whose outputs are independent, as in the tree-type configuration shown in Figure 7.3-9 will have the potential for slipping.

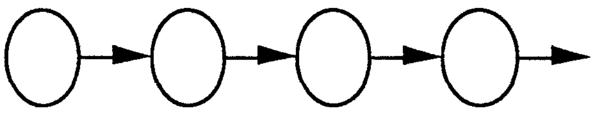

Figure 7.3-8: Simple Slipping Configuration.

In a stable state, when there is no drive to change the phase of a cell, C, the phase information is being stored by the cell and is therefore vulnerable to soft error. One type of stable state occurs when the cell receiving the information, R, is not acknowledging the new phase thaat C is sending. The other type occurs when no new phase input is available from the input cell, I. When C is in the simple configuration of Figure 7.3-10, sticking results from soft errors received during either type of stable state because the other cell, D, acts as a fault detector.

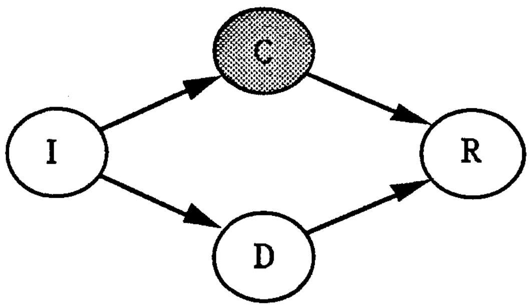

Figure 7.3-10: Simple Sticking Configuration.

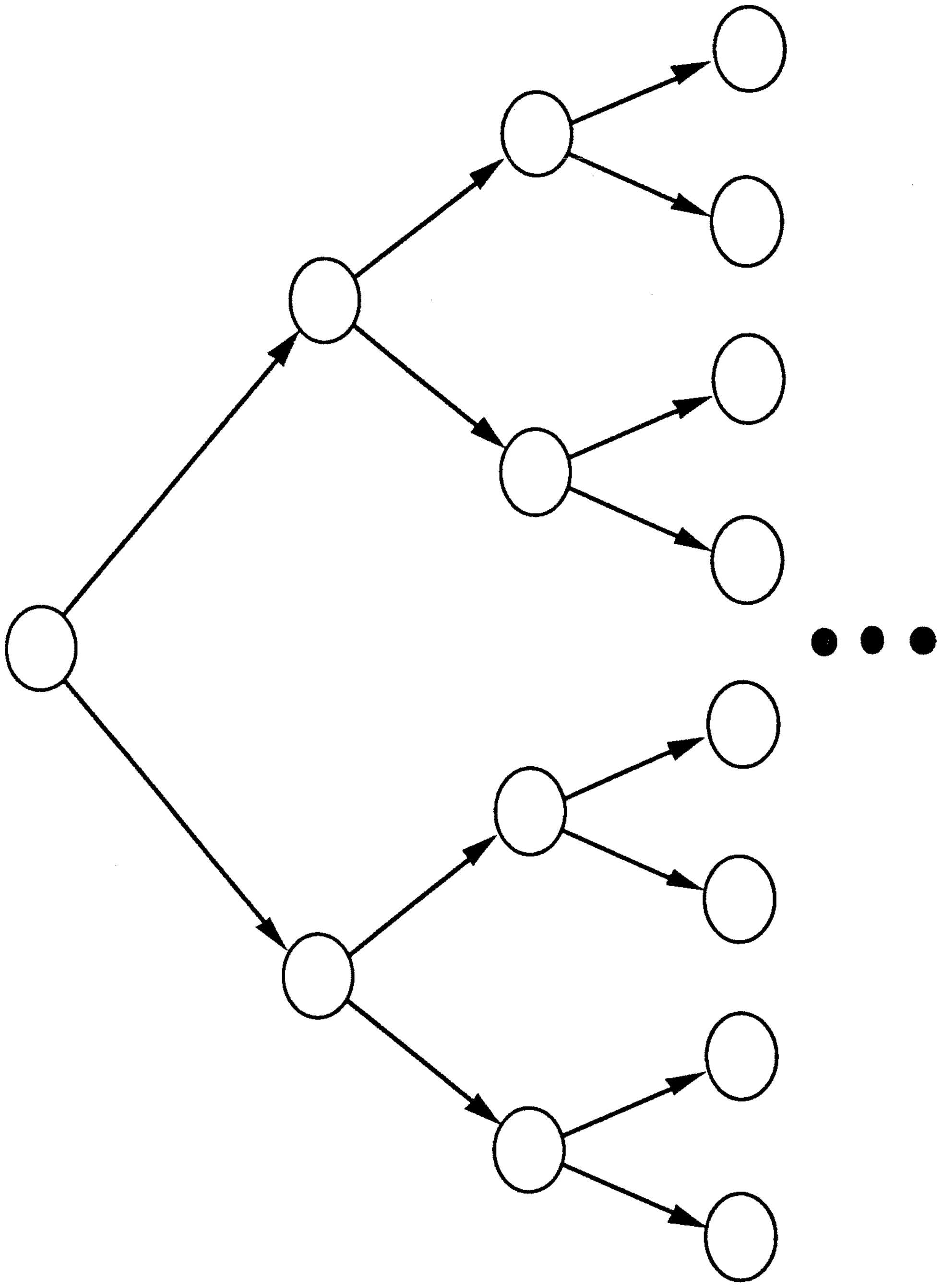

Figure 7.3-9: More General Slipping Configuration.

In Figure 7.3-11, cell C is shown in a stable state where the cell R is not acknowledging its output. After being hit by a soft error, the phase of C is changed from P to Q. The phase of cell D, however, remains unchanged and will not change until cell R acknowledges its P phase. Cell R cannot change phase because cell C is of the wrong phase and does not appear to hold new information. Cell C can change phase because its old P input from cell I is still stored at cell I. This error correction is the only change that can occur in this phase state.

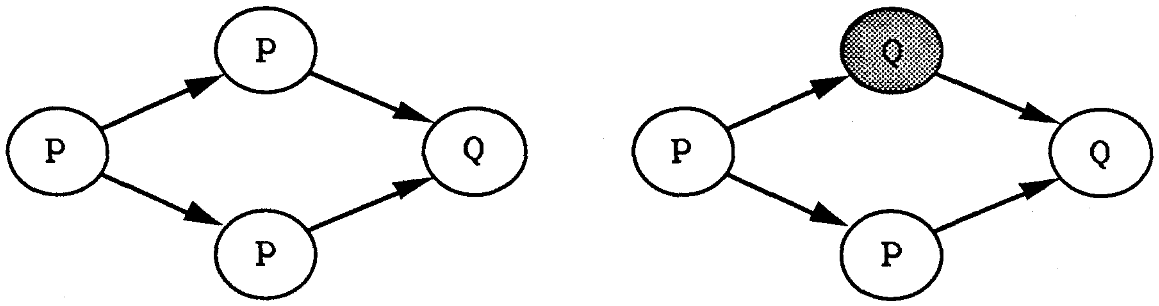

Figure 7.3-11: Correcting Phase State.

Figure 7.3-12 shows the same phase configuration except that cell I has already received new information. In this phase configuration, cell R cannot change because its inputs are of a different phase. Cell D cannot change because its output is not being acknowledged by cell R. Cell I cannot change because its Q phase is not acknowledged by cell D. Finally, cell C cannot change because it is not receiving new phase information from cell I.

Figure 7.3-13 shows cell C waiting for a new input from cell I. After an error changes the phase of cell C, cell I is no longer receiving an acknowledgement of its information and cannot change phase. Cell R cannot receive cell C's error because cell D is still in phase P. Cell D cannot change phase because it is still waiting for a new input from cell I. Additional inputs to the cell C or outputs from the cell would not change this sticking response, because they can only add more phase

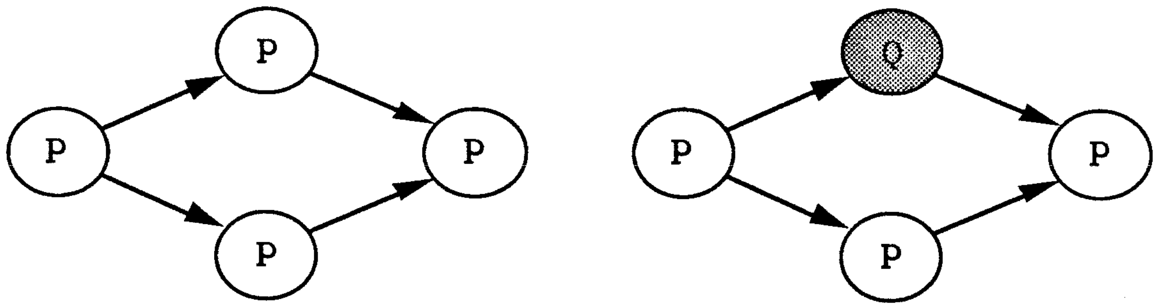

Figure 7.3-12: Waiting for New Input.

restrictions. In Figure 7.3-14, every cell in the mesh, except the upper left and lower right endpoint, is in the configuration of Figure 7.3-10. The entire mesh will therefore stick if any one of the cells gets an error. Further, a sticking architecture results when any of the cells of the mesh is replaced by a configuration of cells which itself sticks in response to errors.

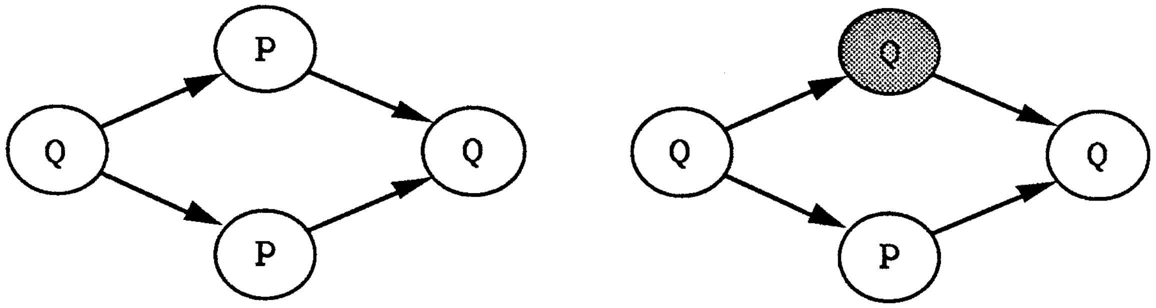

Figure 7.3-13: Waiting for Acknowledgment.

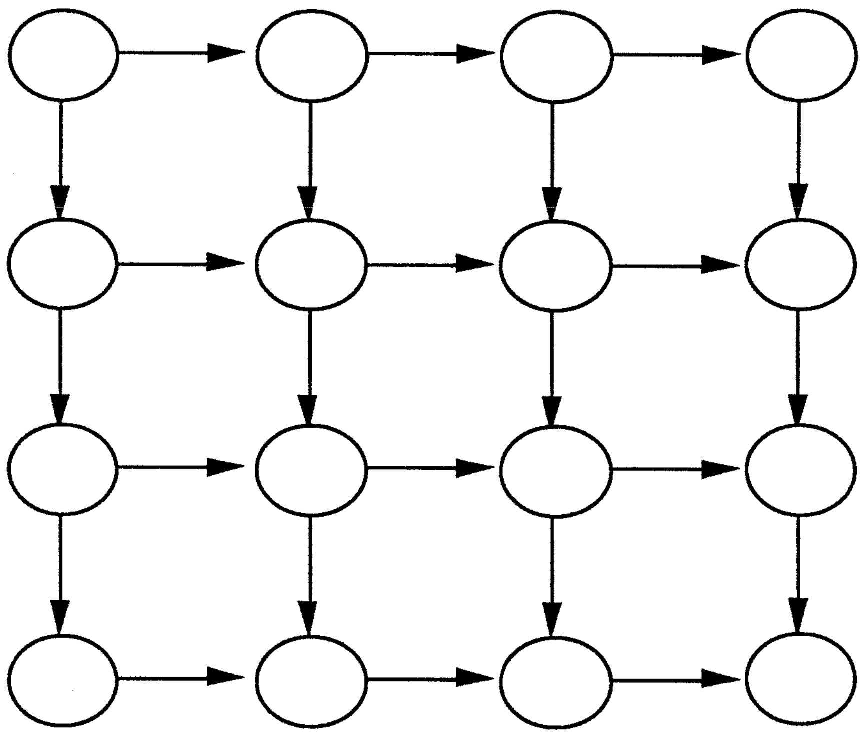

Figure 7.3-14: More General Sticking Architecture.

Chapter 8 Conclusion

As ICs become denser and faster, the system designer is able to use very large operands in their arithmetic units. These architectures, such as those required for Public Key Cryptography (1024 bit operands), push back system performance limits. As the size increases, however, there is need to re-evaluate the scalability of the architecture and the susceptibility to errors.

As an example of the need to change the architecture, a new binary divider for large operand division has been proposed. This architecture, one which can be applied to Public Key Cryptography, produces the quotient and remainder in $O(n)$ time using $O(n)$ area, where n is the operand size. The new divider is faster than comparable carry save dividers, and more area-efficient than implementations using more redundant arithmetic.

Besides architecture scalability and area, key design complexity issues were also addressed. A small 16 bit synchronous version of our divider showed that the architecture is suitable for VLSI implementation. Future research on the division architecture will include implementing a large 1024 bit asynchronous version, with a fault tolerant architecture.

The error susceptibility of denser, faster IC's motivated the research for a simplified approach to fault tolerance. A novel method was proposed for asynchronous arrays; the method exploits the inherent fault tolerance of asynchronous arrays to achieve fault tolerance with duplication rather than triplication. Errors always resulted in sticking, rather than slipping that masks the errors, in architectures of a general form to which non-sticking architectures can be easily converted.

Although the architectures discussed all used four-state coding, this approach is not necessarily limited to only these asynchronous circuits. Other types of wavefront arrays could be analyzed to find, or even altered to achieve, the sticking or deadlock response to faults that would allow error correction with only duplication.

References

[1] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design.* New York: Wiley, 1979.

[2] F. P. Preparata and J. E. Vuillemin, "Practical Cellular Dividers," *IEEE Trans. on Computers*, vol. C-39, pages 605 - 614, May 1990.

[3] R. P. Brent and H.T. Kung, "A regular layout for Parallel Adders," *IEEE Trans. on Computers*, vol. C-31, pages 260 - 264, March 1982.

[4] C. N. Purdy and G. B. Purdy, "Integer Division in Linear Time with Bounded Fan-In," *IEEE Trans. on Computers*, vol. C-36, pages 640 - 644, May 1987.

[5] J. E. Robertson, "A new class of Digital Division Methods," *IEEE Trans. on Computers*, vol. C-07, pages 218 - 222, September 1958.

[6] D. E. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Trans. on Computers*, vol. C-17, pages 925 - 234, September 1968.

[7] Atkins, "Design of the Arithmetic Units of ILLIAC III: Use of Redundancy and Higher Radix Methods," *IEEE Trans. on Computers*, vol. C-19, pages 720 - 733, August 1970.

[8] K. S. Trivedi and M.D. Ercegovac, "On-Line Algorithms for Division and Multiplication," *IEEE Trans. on Computers*, vol. C-26, pages 681 - 687, July 1977.

[9] M. D. Ercegovac and T. Lang, "Simple Radix-4 Division with Operand Scaling" IEEE Computer, vol. 15, pages 37-46, Jan. 1982.

[10] R. M. F. Goodman and A. J. McAuley, "An Efficient Asynchronous Multiplier," *International Conference on Systolic Arrays*, San Diego, California, USA,

May 25-26, 1988.

[11] A. J. McAuley, "Four State Asynchronous Archtectures," submitted to the *IEEE Trans. on Computers.*

[12] T. J. Brosnan and N.R. Strader II, "Modular Error Detection for Bit-Serial Multiplication," *IEEE Trans. on Computers*, vol. 37, No. 9, 1043- 1052, September 1988.

[13] Niraj K. Jha, "Multiple Stuck-Open Fault Detection in CMOS Logic Circuits," *IEEE Trans. on Computers*, vol. 37, No. 4, pages 426-432, April 1988.

[14] O. Tasar and V. Tasar, "A Study of Intermittent Faults in Digital Computers," in *AFIPS Conf. Proc.*, vol. 46, pp 807- 811, 1979.

[15] B. Chappell, S. Schuster, G. A. Sai-Halasz , "Stability and SER Analysis of Static RAM Cells", *IEEE Trans. on Electron Devices*, vol. ED-32, No. 2, pages 463 - 470 , Feb. 1985.

[16] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors – A Survey," *IEEE Trans. on Computers*,vol. 37, No. 2, 160 - 174, Feb. 1988.

[17] I. Gazit and M. Malek, "Fault Tolerance Capabilities in Multistage Network-Based Multicomputer Systems," *IEEE Trans. on Computers*, vol. 37, No. 7, pages 788 - 798, July 1988.

[18] J. R. Connet, E. J. Pasternak, and B.D. Wagner, "Software Defenses in Real Time Control Systems," in *Dig. Int. Symp. Fault Tolerance Comput.*, FTCS-2 , Newton, MA, 94-99, June 19-21, 1972.

[19] J. S. Novak and L. S. Tuomenoksqua, "Memory Mutilation in Stored Pro-

gram Controlled Telephone Systems," in *Conf. Rec. 1970 Int. Conf. Commun.*, vol. 2, 43-32 to 43-45, 1970.

[20] S. M. Ornstein, "Pluribus-A Reliable Multiprocessor," in *Proc. AFIPS Conf.*, vol 44, Anaheim, CA May 19-22, 1975 551-559.

[21] T. Nanya and T. Kawamura, "Error Secure Propagating Concept and its Application to the Design of Strongly Fault-secure Processors," *IEEE Trans. on Computers*, vol. 37, No. 1, pages 14 - 24, Jan. 1988.

[22] D. Nikolos, A. M. Paschalis, and G. Philokyprou, "Efficient Design of Totally Self-Checking Checkers for all Low-Cost Arithmetic Codes,"*IEEE Trans. on Computers*, vol. 37, No. 7, 807 - 811, July 1988.

[23] B. Bannister and D. Whitehead, *Fundamentals of Digital Systems*, McGraw Hill, London, 1973.