Lehigh University
Lehigh Preserve

Theses and Dissertations

2001

Barrel shifter design, optimization, and analysis

Matthew Rudolf Pillmeier
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Pillmeier, Matthew Rudolf, "Barrel shifter design, optimization, and analysis" (2001). *Theses and Dissertations.* Paper 714.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Pillmeier, Matthew Rudolf

Barrel Shifter Design, Optimization, and Analysis

January 2002

Barrel Shifter Design, Optimization, and Analysis

by

Matthew Rudolf Pillmeier

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Engineering

Lehigh University

January 2002

This thesis is accepted in partial fulfillment of the requirements for the degree of Master of Science.

12/5/01

(Date)

Prof. Michael J. Schulte
(Thesis Advisor)

Prof. Bruce D. Fritchman
(Chairperson of Department)

Acknowledgements

First and foremost, I would like to thank Prof. Michael J. Schulte for his support, direction, and guidance, in this project. In addition, I would like to thank my family and friends for giving me the support I needed to finish this project.

Table of Contents

Acknowledgements iii

Table of Contents iv

List of Tables vii

List of Figures ix

Abstract 1

1. Introduction 2

1.1. Operations 2

1.1.1.Rotate 2

1.1.2.Shift Right Logical 3

1.1.3.Shift Left Logical 4

1.1.4.Shift Right Arithmetic 4

1.1.5.Shift Left Arithmetic 5

1.2. Flags 6

1.2.1.Zero Flag 6

1.2.2.Overflow Flag 6

2. Previous Research 7

2.1. Uni-directional Shift/Rotate Mechanisms 7

2.1.1.Logical Right Shifter 8

2.1.2.Right Rotator 9

2.2. Bi-directional Shift/Rotate Mechanisms 10

2.2.1.Series Bi-directional Logical Shifter 10

2.2.2.Parallel Bi-directional Logical Shifter 12

2.2.3.Data Reversal Bi-directional Logical Shifter 14

2.2.4.One's Complement Bi-directional Rotator 16

2.2.5.Masking Rotating Shifter 17

2.3. Shift Operation Zero Flag 19

2.4. Overflow Flag 20

3. Current Research 23

3.1. Mux-based Data Reversal 23

3.1.1.Design Overview 23

3.1.2.Zero Flag 26

3.1.3.Overflow Flag 27

3.1.4.Examples 30

3.2. Mask-based Data Reversal 33

3.2.1.Design Overview 33

3.2.2.Mask F Generator 35

3.2.3.Zero Flag 37

3.2.4.Overflow Flag 38

3.2.5.Examples 41

3.3. Mask-based Two's Complement 44

3.3.1.Design Overview 44

3.3.2.Two's Complement 46

3.3.3.Zero Flag 47

3.3.4.Register Load Optimized 48

3.3.5.Examples 50

3.4. Mask-based One's Complement 53

3.4.1.Design Overview 53

3.4.2.Zero Flag 56

3.4.3.Examples 56

4. Results 61

4.1. Estimates for Component Count and Number of Components on Critical Path 61

4.2. Synthesis Results for Area and Delay 61

4.2.1.Designs without Flags 66

4.2.2.Designs with Zero Flag 66

4.2.3.Designs with Zero and Overflow Flags 66

4.2.4.Register Load Optimized Mask-based Two's Complement 67

5. Conclusions and Future Research 71

5.1. Conclusions 71

5.2. Future Research 72

Bibliography 74

Vita 75

List of Tables

2.1 Component Counts and Critical Delay Paths for Bi-directional Logical Shifters 12

2.2 Overflow Flag Calculation Method 1 Example 22

2.3 Overflow Flag Calculation Method 2 Example 22

3.1 Mux-based Data Reversal Rotate Right Example 31

3.2 Mux-based Data Reversal Rotate Left Example 31

3.3 Mux-based Data Reversal Shift Right Logical Example 32

3.4 Mux-based Data Reversal Shift Left Logical Example 32

3.5 Mux-based Data Reversal Shift Right Arithmetic Example 32

3.6 Overflow Method Comparison 41

3.7 Mask-based Data Reversal Rotate Right Example 41

3.8 Mask-based Data Reversal Rotate Left Example 42

3.9 Mask-based Data Reversal Shift Right Logical Example 42

3.10 Mask-based Data Reversal Shift Left Logical Example 43

3.11 Mask-based Data Reversal Shift Right Arithmetic Example 43

3.12 Rotator Cost 49

3.13 Two's Complement and Shift/Rotate Mux Cost 49

3.14 Mask-based Two's Complement Rotate Right Example 50

3.15 Mask-based Two's Complement Rotate Left Example 51

3.16 Mask-based Two's Complement Shift Right Logical Example 51

3.17 Mask-based Two's Complement Shift Left Logical Example 1 52

3.18 Mask-based Two's Complement Shift Right Arithmetic Example 52

3.19 Mask-based Two's Complement Shift Left Logical Example 2 53

3.20 Mask-based One's Complement Rotate Right Example 58

3.21 Mask-based One's Complement Rotate Left Example 58

3.22 Mask-based One's Complement Shift Right Logical Example 59

3.23 Mask-based One's Complement Shift Left Logical Example 59

3.24 Mask-based One's Complement Shift Right Arithmetic Example 60

4.1 Mux-based Data Reversal Theoretical Gate Count and Delay Measure 62

4.2 Mask-based Data Reversal Theoretical Gate Count and Delay Measure 63

4.3 Mask-based Two's Complement Theoretical Gate Count and Delay Measure 64

4.4 Mask-based One's Complement Theoretical Gate Count and Delay Measure 65

4.5 Designs without Flags Area and Delay Synthesis Results 68

4.6 Designs with Zero Flag Area and Delay Synthesis Results 69

4.7 Designs with Zero and Overflow Flags Area and Delay Synthesis Results 70

4.8 Register Load Optimized Mask-based Two's Complement Area and Delay Synthesis Results 70

List of Figures

1.1	Rotate Right	2
1.2	Rotate Left	2
1.3	Rotate Right by 8 Example	3
1.4	Rotate Left by 8 Example	3
1.5	Shift Right Logical	3
1.6	Shift Right Logical by 8 Example	3
1.7	Shift Left Logical	4
1.8	Shift Left Logical by 8 Example	4
1.9	Shift Right Arithmetic	5
1.10	Shift Right Arithmetic by 8 Example	5
1.11	Shift Left Arithmetic	6
1.12	Shift Left Arithmetic by 8 Example	6
2.1	Logical Right Shifter	9
2.2	Right Rotator	10
2.3	Series Bi-directional Logical Shifter	11
2.4	Parallel Bi-directional Logical Shifter	13
2.5	Data Reversal Bi-directional Logical Shifter	15
2.6	One's Complement Bi-directional Rotator	17
2.7	Masking Rotating Shifter	18
2.8	Zero Flag Calculation	20
2.9	Previous Overflow Flag Calculation	21
3.1	Mux-based Data Reversal	24
3.2	Right Shifter/Rotator	25

3.3 Mux-based Data Reversal with Zero Flag 27

3.4 Mux-based Data Reversal with Zero and Overflow Flags 28

3.5 Right Shifter/Rotator with Overflow Flag 29

3.6 Mask-based Data Reversal 34

3.7 Mask F Generator 36

3.8 Mask-based Data Reversal with Zero Flag 38

3.9 Mask-based Data Reversal with Zero and Overflow Flags 39

3.10 Current Overflow Flag Calculation 40

3.11 Mask-based Two's Complement 44

3.12 Two's Complement Unit 47

3.13 Mask-based Two's Complement with Zero Flag 48

3.14 Mask-based One's Complement 54

3.15 Mask F Generator for One's Complement 55

3.16 Mask-based One's Complement with Zero Flag 57

Abstract

Barrel shifters are arithmetic and logic circuits that may be utilized to shift or rotate data in a general-purpose microprocessor or digital signal processor. This thesis proposes and analyzes various methods of implementing barrel shifters. The purpose of this thesis is to understand the tradeoffs of various barrel shifter design approaches in order to recognize where each may be most useful.

Each design is a compromise between gate count and critical path latency. In an attempt to reduce both, the proposed designs utilize a number of innovative design techniques. The techniques can be divided into two categories: those addressing uni-directional result computation and those providing the logic necessary to implement all operations with uni-directional hardware support. Four design schemes were employed to test each of the techniques; Mux-based Data Reversal, Mask-based Data Reversal, Mask-based Two's Complement, and Mask-based One's Complement. The mux-based and mask-based descriptor indicates the uni-directional result computation method, while the rest specify the mechanism used to emulate bi-directional operations with uni-directional hardware support.

Analysis of each design reveals some unique findings. First of all, the designs using the two's complement and one's complement mechanisms were found to have a critical path latency much higher than expected, thus they are of very limited use unless the shift/rotate amount arrives earlier than the data to be shifted or rotated. Second, the optimal designs were found to be the Mux-based Data Reversal and Mask-based Data Reversal approaches. Each had comparable area-delay products. If gate count minimization is the primary concern, then the mux-based approach is preferred. Likewise, critical path latency minimization is achieved with the mask-based approach. Thus, no single design is preferred for all circumstances. Instead, use is highly dependent on the particular demands placed on the circuit.

Chapter 1 - Introduction

1.1 Operations

A barrel shifter primarily offers five operations; rotate right, rotate left, shift right logical, shift left logical, and shift right arithmetic. Occasionally, the shift left arithmetic operation is also included, but it is not supported in the designs detailed here due to its infrequent use.

1.1.1 Rotate

A rotate is a cyclic shift either to the left or right. This means that as bits are shifted out of the data vector on one side, they are shifted into the data vector on the other side. During this process, all bits from the input are routed to the output. Their position in the output, however, is not necessarily the same as it was in the input.

As shown in Figure 1.1, a *k*-bit rotate right moves *k* low order bits to the most significant end of the bit vector. Likewise, as shown in Figure 1.2, a *k*-bit rotate left moves *k* high order bits to the least significant end of the bit vector. The remaining (*n*-*k*) bits are shifted so as to fill the void left by the *k* bits shifted in a cyclic manner.

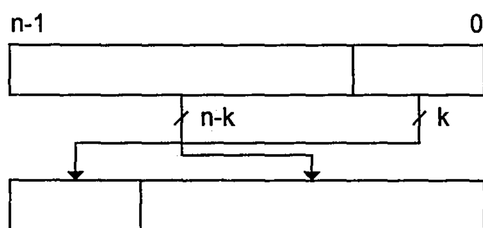

Figure 1.1: Rotate Right

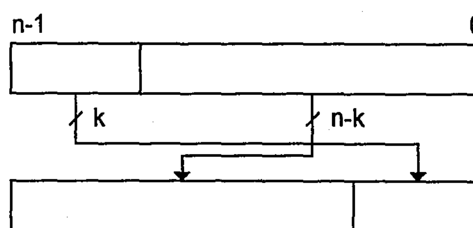

Figure 1.2: Rotate Left

An example of the rotate right can be seen in Figure 1.3 where a 32-bit word of data has a low order byte rotated right. As can be seen, this byte is moved into the most significant portion

of the word. The remaining bits are simply shifted to the right. This fills the void left by the byte moved and simultaneously creates a void in the high order bit region for the rotated byte to be placed. In a similar fashion, Figure 1.4 demonstrates the manner in which a high order byte is moved into the low order region of the word during a rotate left. In both instances, a byte of data is moved from one end of the word to the other with the rest of the word reorienting itself to accommodate this alteration.

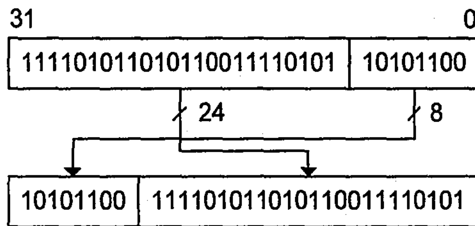

Figure 1.3: Rotate Right by 8 Example

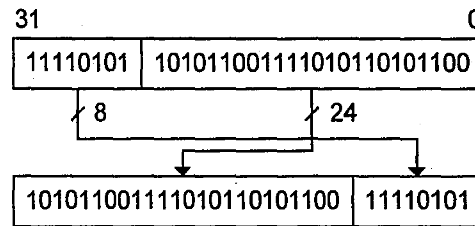

Figure 1.4: Rotate Left by 8 Example

1.1.2 Shift Right Logical

The shift right logical operation is much like a rotate right, but without the *k* low order bits being moved to a high order position. Instead, the low order bits are removed. The remaining (*n-k*) bits are shifted to the right so as to fill the void created by the loss of the low order bits. The void created in the high order region by this shift is filled with zeros. Figure 1.5 illustrates this process and Figure 1.6 is an example of a shift right logical by 8 operation. As can

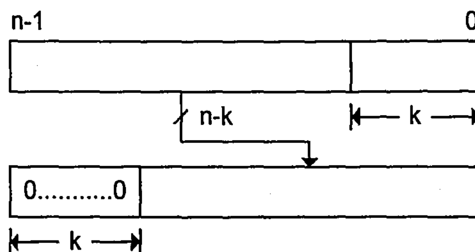

Figure 1.5: Shift Right Logical

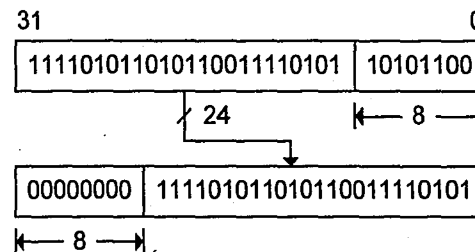

Figure 1.6: Shift Right Logical by 8 Example

be seen, the low order byte is removed from the result and the remaining bits are shifted over. The high order bits are set to zero. As such, a shift right logical operation approximates division by 2^k on unsigned data, where the result is truncated.

1.1.3 Shift Left Logical

The shift left logical operation is similar to the shift right logical operation. The difference, of course, lies in the direction of the shift, which, in this case, is to the left. As such, the *k* high order bits are removed from the high order region so that the remaining (*n-k*) bits can be shifted to the left *k* places. The void created in the low order region is then filled with zeros, as shown in Figure 1.7. Figure 1.8 demonstrates this operation. In this example, the high order byte is removed. The rest of the data is shifted to the left to fill this void. The void created in the low order region is then filled with zeros. Therefore, this command corresponds to a multiplication by 2^k in those cases where the block of data being dropped is all zeros.

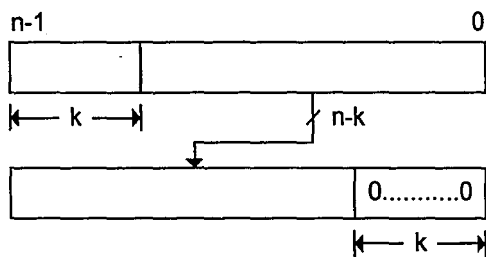

Figure 1.7: Shift Left Logical

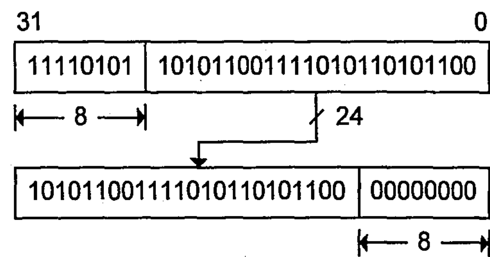

Figure 1.8: Shift Left Logical by 8 Example

1.1.4 Shift Right Arithmetic

The shift right arithmetic operation is similar to the right shift logical operation except for the fact that the bits used to fill the high order region are not necessarily zero. These bits are

instead dependent on the sign bit, the most significant bit, of the data. The sign bit is used as the fill bit, as shown in Figure 1.9. As such, a shift right arithmetic operation maintains the sign of the data. Figure 1.10 demonstrates this operation with a sign bit of one. As can be seen, once the data is arithmetically right shifted, the sign bit with a value of one is used to fill the high order region. Thus, the shift right arithmetic operation approximates division by 2^k on two's complement data, where the result is truncated.

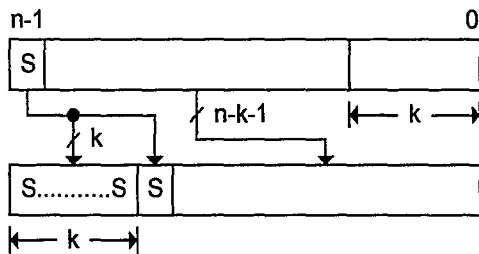

Figure 1.9: Shift Right Arithmetic

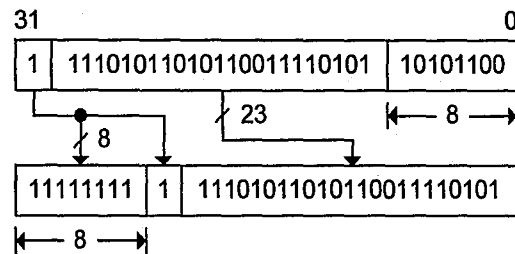

Figure 1.10: Shift Right Arithmetic by 8 Example

1.1.5 Shift Left Arithmetic

The shift left arithmetic operation is a combination of the actions taken by the shift right arithmetic and shift left logical operations. As shown in Figure 1.11, the shift left arithmetic operation maintains the sign bit by excluding it from the shift. The (n-k-1) low order bits are shifted to the left k spaces. The k high order bits, excluding the sign bit, make up the region shifted out of the data vector. The void created in the low order region is filled with zeros. An example of this operation is shown in Figure 1.12. This operation corresponds to multiplication by 2^k on two's complement data, as long as each of the k-bits shifted out of the data vector is equal to the sign bit. Due to the limited use of this operation, it is not supported in the designs analyzed in this thesis.

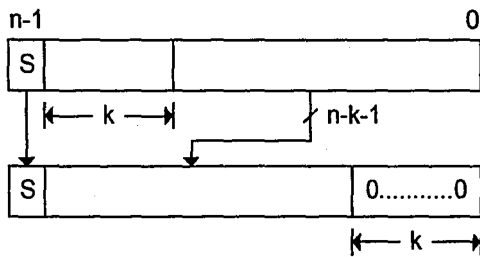

Figure 1.11: Shift Left Arithmetic

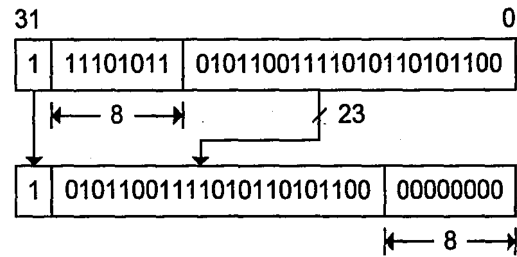

Figure 1.12: Shift Left Arithmetic by 8 Example

1.2 Flags

Barrel shifters often produce flags that indicate special conditions. The two most common are the zero and overflow flags.

1.2.1 Zero Flag

The zero flag is a simple 1-bit flag that indicates whether the result of the operation performed has a value of zero. A value of one indicates that the result is zero, while a zero indicates a non-zero result. This flag is useful when the result is used in an equality test with zero, as is done in some processors, or when the result of a shift/rotate operation sets the processor condition codes.

1.2.2 Overflow Flag

The overflow flag is also a 1-bit flag. It is used to indicate a sign bit change during a left shift operation. More specifically, if the shift were to be implemented as successive 1-bit shifts, then overflow is said to occur if at any point during the process, the bit passing over the sign bit location is different than the original sign bit. Overflow is indicated by a value of one. The overflow flag is useful when one needs to know if the true result cannot be represented using n bits.

Chapter 2 - Previous Research

This section concentrates on techniques previously applied to barrel shifter designs that may offer insight as to which design direction has the highest potential for current research. As such, many designs will be reviewed so as to gain an understanding of what each offers and what can be learned from the various approaches. In these designs, data is an n-bit vector where n is an integer power of two.

2.1 Uni-directional Shift/Rotate Mechanisms

When performing a shift in a single direction, there are multiple approaches that can be taken in terms of design. First, a shift of any magnitude can be computed as a number of successive 1-bit shifts. A shift register accomplishes this. This is, however, not entirely practical due to its inefficiencies in terms of delay and the fact that the delay is dependent on the amount of the shift. The latter creates problems for analysis and scheduling, which one would like to avoid.

Second, a shift can be performed in whole with a single shift equal to the shift amount. This would be the quickest method of performing the shift if only a single shift amount were allowed. That is not the case, however, so a design would require that all possible shift amounts be available. A selection mechanism would then be required to select the shift of the proper amount. This is impractical due to its cost inefficiencies.

Finally, a cross between the previous approaches is proposed. With this method, a shift is performed by breaking it into stages less than the whole amount and greater than one. This is done in such a way that a shift of any amount is possible through a linear combination of the shift amounts offered. In this manner, more of the overall shift is performed in every step, which reduces the number of required steps, while simultaneously allowing shifts of any value.

Fortunately, such a breakdown already exists in the structure of an unsigned binary number. Specifically, an unsigned binary number is inherently a breakdown into powers of two. Any number can therefore be represented as a sum of powers of two. If this breakdown is applied to how a shift is performed, then given an n-bit data value, only $\log_2(n)$ bits are required to represent any allowed shift. As such, the shift can be broken into $\log_2(n)$ stages [1], [2]. Even if n is very large, this value will still be relatively small.

Therefore, a shifter using this natural breakdown of the shift amount has $\log_2(n)$ rows of multiplexors. Each row performs a shift by an amount equal to an integer power of two. For example, the shift amounts for n equal to 8 are: 1,2, and 4. It does not so much matter what order these shifts are performed, but that the correct bit from the shift amount is used to control the multiplexors of its stage. All of the designs presented will use structures of this sort. This type of shifter is often called a barrel shifter.

2.1.1 Logical Right Shifter

A Logical Right Shifter using the aforementioned approach is shown in Figure 2.1. The first row corresponds to a shift of one, while the last row corresponds to a shift of four. As required, zeros fill the high order region. Hence, interconnects route zero into the high order multiplexors. The value s_amt[x] represents the bit in position x of the shift amount, and as such represents the value 2^x.

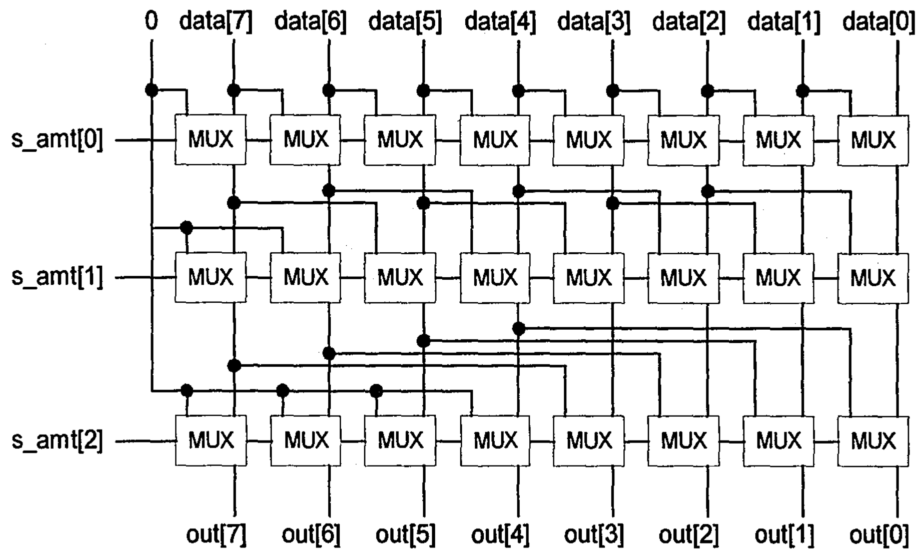

Figure 2.1: Logical Right Shifter

2.1.2 Right Rotator

A Right Rotator is very similar to a Logical Right Shifter. The difference between the two lies in the manner in which interconnect lines are placed. In particular, since all of the input bits are routed to the output, there is no longer a need for interconnect lines carrying the zero signal. Instead, interconnect lines need to be inserted to enable routing of the low order bits from each row to the high order region so that rotate can occur. Figure 2.2 shows a Right Rotator. The change to a rotator from a shifter has no impact on the theoretical cost or delay. The longer interconnect lines of the rotator, however, can increase both area and delay.

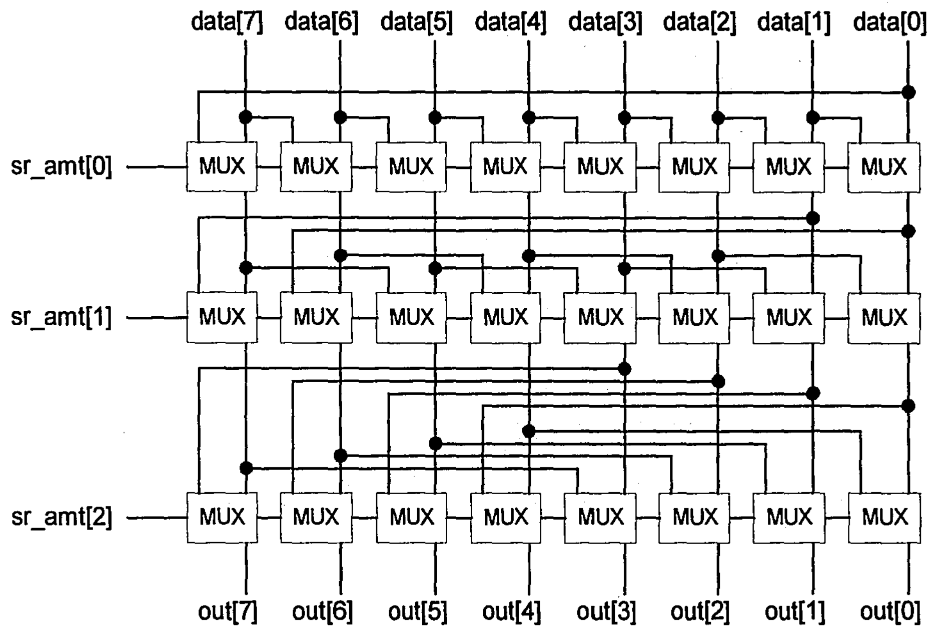

Figure 2.2: Right Rotator

2.2 Bi-directional Shift/Rotate Mechanisms

In this section, designs that perform either a shift, a rotate, or both in the right and left directions are explored. The designs offer insight into the approaches taken to design a barrel shifter with the capability to perform some or all of the operations described in Section 1.1.

2.2.1 Series Bi-directional Logical Shifter

The Series Bi-directional Logical Shifter utilizes two sequential uni-directional shifters to accommodate the ability to shift logically either to the right or left [3]. This is accomplished by having one of these shifters operate in the right direction and the other in the left direction. For any given shift, only one shifter will actually alter the data. The other will simply operate as a pass through, or perform a zero shift. As such, the design allows for bi-directional shifts to take place, albeit with slightly more than twice the hardware.

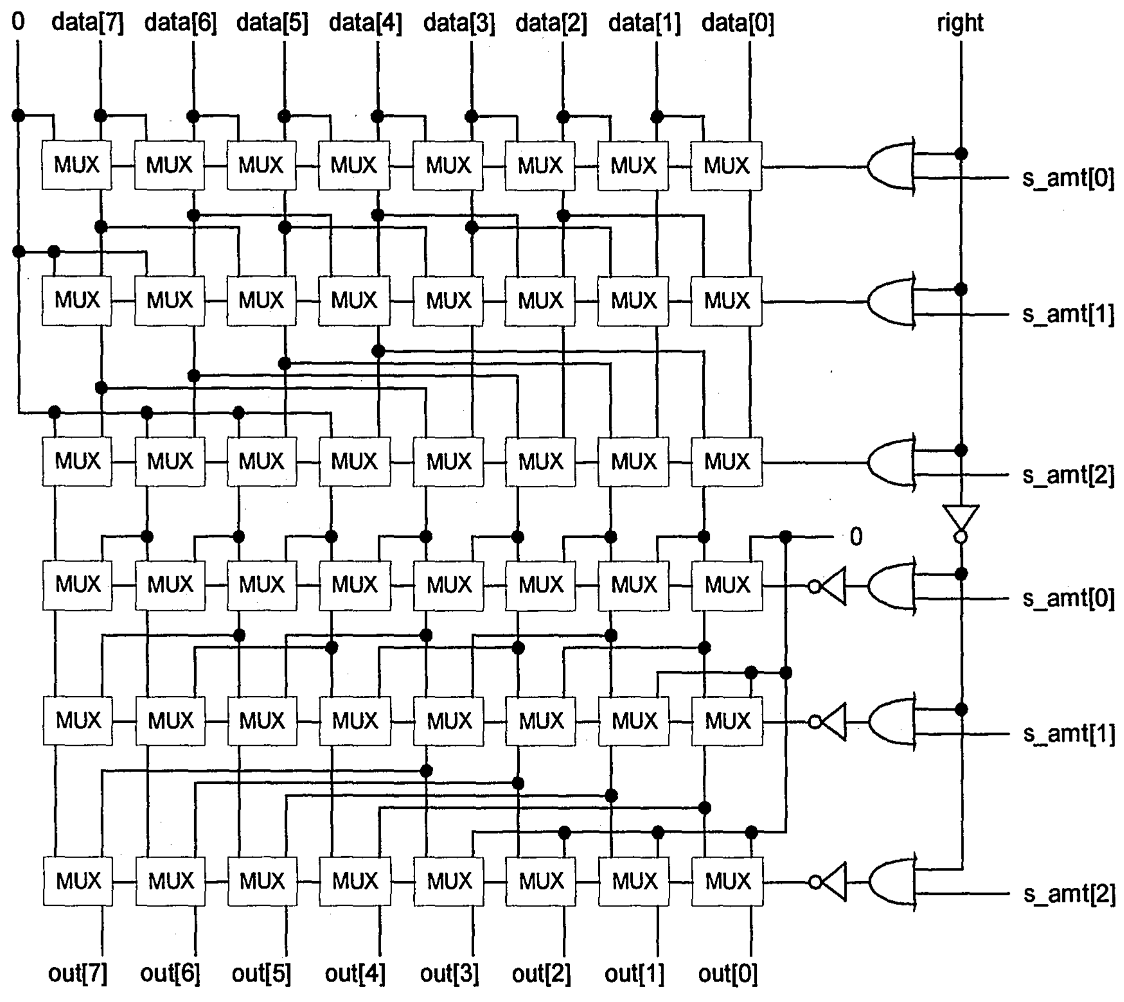

Figure 2.3: Series Bi-directional Logical Shifter

As one can see from Figure 2.3, the first shifter in the series operates in the right direction. If the desired operation is a right shift, then the logic will pass the shift amount to the multiplexors that actually accomplish the shift. Otherwise, the multiplexors receive a logic low and simply pass the input onto the second shifter. The second shifter is much like the first except for the fact that it shifts towards the left. If the operation signals a left shift, then the shift amount is passed to the multiplexors in order for the shift to occur. The result of this second shifter is the result of the entire unit.

From a cost analysis point of view, this design is fairly costly. Table 2.1 lists the cost of this design and two designs yet to be mentioned. As can be seen, this design has $2\log_2(n)$ multiplexors in the critical path. The design as a whole requires $2n\log_2(n)$ multiplexors. Therefore, it is easily seen that this design is costly, but its relative cost will be more easily seen after reviewing the next two designs.

Version	AND(2 input)	NOT	MUX(2-to-1)
		Total Components	
Series Bi-directional Logical Shifter	$2\log_2(n)$	$\log_2(n)+1$	$2n\log_2(n)$
Parallel Bi-directional Logical Shifter	0	0	$n+2n\log_2(n)$
Data Reversal Bi-directional Logical Shifter	0	0	$2n+n\log_2(n)$
		Components on the Critical Delay Path	
Series Bi-directional Logical Shifter	1	0	$2\log_2(n)$
Parallel Bi-directional Logical Shifter	0	0	$\log_2(n)+1$
Data Reversal Bi-directional Logical Shifter	0	0	$\log_2(n)+2$
		Approx. Area-Delay Product - MUX(2 to 1)	
Series Bi-directional Logical Shifter		$4n\log_2^2(n)$	
Parallel Bi-directional Logical Shifter		$n+2n\log_2^2(n)+3n\log_2(n)$	
Data Reversal Bi-directional Logical Shifter		$4n+n\log_2^2(n)+4n\log_2(n)$	

Table 2.1: Components Counts and Critical Delay Paths for Bi-directional Logical Shifters

2.2.2 Parallel Bi-directional Logical Shifter

As in the aforementioned Series Bi-directional Logical Shifter, the Parallel Bi-directional Logical Shifter also utilizes two uni-directional shifters, one for the right direction and another for the left. These units, however, are placed in parallel as opposed to in series [3]. In this manner, both a right and left shift are computed simultaneously. A multiplexor following these shifters then selects the result from the shifter operating in the desired direction.

As shown in Figure 2.4, the shifter on the left performs the right shift, while the shifter on the right performs the left shift. The output of each shifter is routed to a row of multiplexors that use the control signal right to select the properly shifted data.

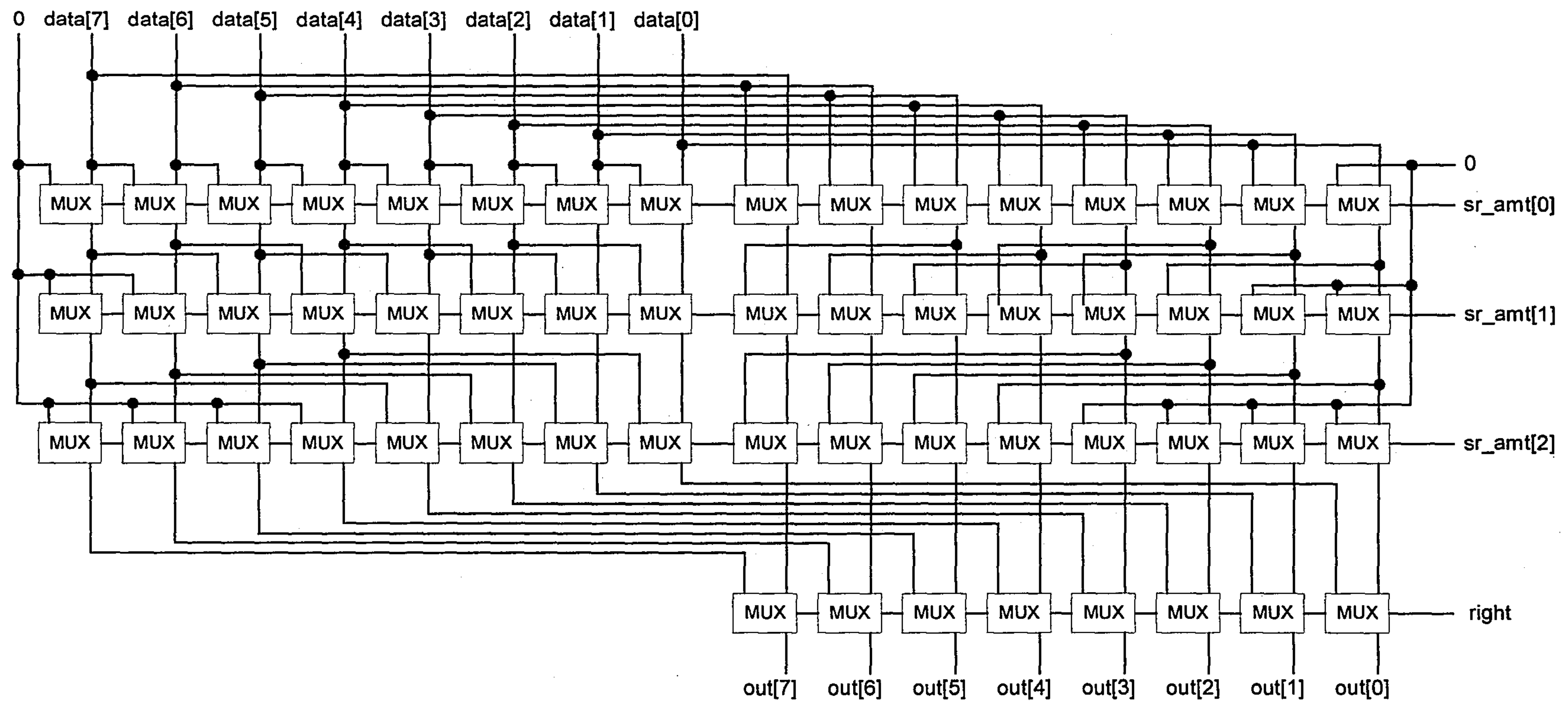

Figure 2.4: Parallel Bi-directional Logical Shifter

Referring back to Table 2.1, it can be seen that the critical path delay of this unit is reduced to $\log_2(n)+1$ multiplexors. The cost has, however, risen to $2n\log_2(n)+n$ multiplexors due to the inclusion of the multiplexors to select the proper shift result. Therefore, while the delay is reduced, the cost is increased. This ambiguity is resolved by using the area-delay product, a metric unifying the competing attributes of cost and delay, to compare the designs. Since 2-to-1 multiplexors dominate the design, they are the only component used to compute the area-delay product. In this case, the area-delay product for the Parallel Bi-directional Logical Shifter is less than that for the Series Bi-directional Logical Shifter. Therefore, the reduction in delay dominates the increase in cost.

2.2.3 Data Reversal Bi-directional Logical Shifter

The previous two designs offer solutions by using two uni-directional shifters. Understandably, they are expensive solutions due to their redundant nature. A more efficient design would use only a single shifter. Offering only a single shifter means that a shift operation can only operate in a single direction. There must then exist a mechanism to adjust the data so that the shift in the supported direction alters the data in a manner conducive to accomplishing the shift in the reverse direction. The Data Reversal Bi-directional Logical Shifter offers such a mechanism.

This design reveals that a solution is to reverse the order of the bits in the data if it is to be shifted in the direction opposite to that which the single shifter supports both before and after the shifter [1], [2], [3], [4]. In this manner, the data is manipulated so that a right shift with two bit reversals performs the desired left shift. Therefore, as seen in Figure 2.5, a bi-directional logical shifter is achieved while utilizing only a single uni-directional shifter.

As Table 2.1 reveals, the delay of this design is larger than that of the Parallel Bi-directional Logical Shifter, but less than that of the Series Bi-directional Logical Shifter. In terms of cost, this design uses fewer gates than the others. The area-delay product shows that this method is preferred to the others. In other words, the decreased cost outweighs the increased delay, which is the opposite of what was seen between the Series Bi-directional Logical Shifter and the Parallel Bi-directional Logical Shifter.

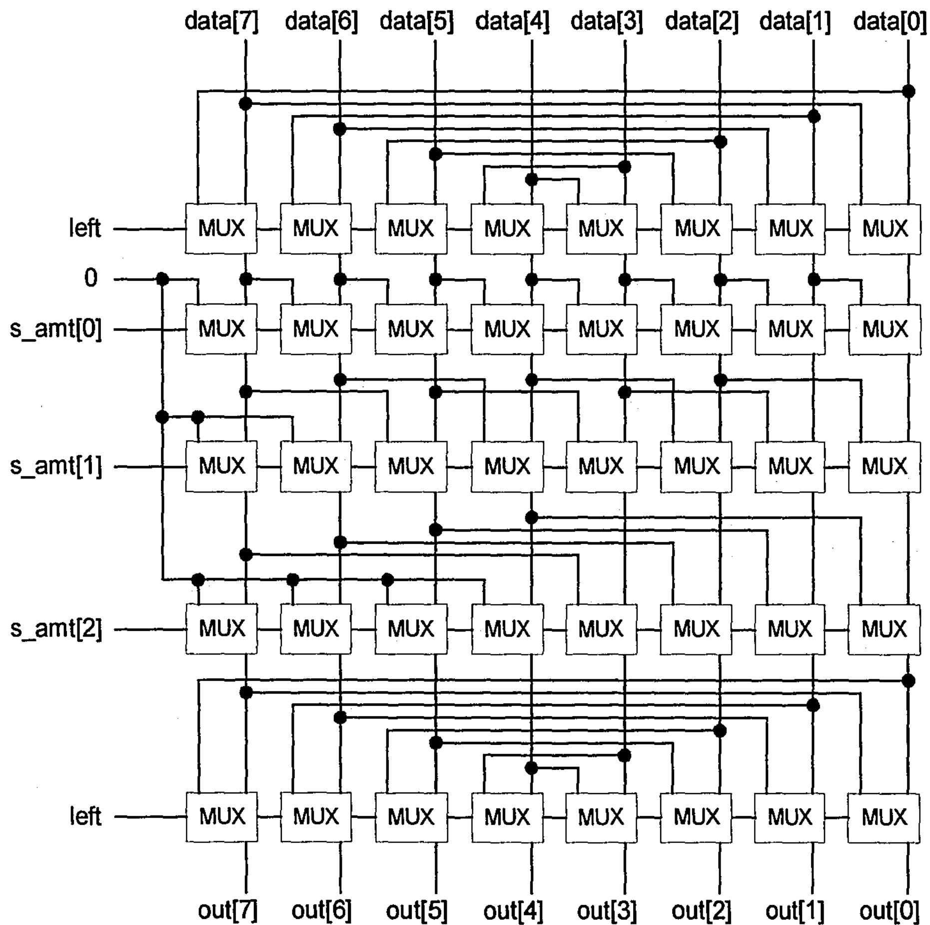

Figure 2.5: Data Reversal Bi-directional Logical Shifter

2.2.4 One's Complement Bi-directional Rotator

Just as the aforementioned Data Reversal Bi-directional Logical Shifter manipulates the data so as to require only a single shifter, the One's Complement Bi-directional Rotator manipulates the rotate amount to accomplish the same goal. This is done by supporting one rotate direction directly and handling the opposite case by calculating the amount that the data must be rotated in the supported direction to emulate a rotate in the opposite direction.

To truly realize this requires that the value (n-r_amt) be calculated, where n is the width of the input data and r_amt is the rotate amount [5]. When n is an integer power of two, this becomes a computation of the two's complement of the rotate amount. Computing the two's complement, however, is slow due to the carry propagation involved. As such, a different approach is taken. Instead of computing the two's complement, the one's complement of the rotate amount is determined [3]. This computation is more easily accomplished, requiring only a simple inversion of the rotate amount bits. Since the one's complement value is off by one, the result may need to be rotated by one for correction. To remedy this, an extra rotate-by-one stage is added to the rotator. This can be seen in Figure 2.6, which shows how an additional rotate is performed for a rotate left when the unit supports rotate right. In addition, the control for the multiplexors has been altered so that the inverse of the rotate amount is taken for a rotate left.

The One's Complement Bi-directional Rotator requires only a single additional row of multiplexors, as opposed to the two additional rows used in the Data Reversal Bi-directional Logical Shifter. As such, the cost is less since the logical XOR gates are generally no more complicated than a multiplexor. The other concern, critical path latency, also decreases. By placing the additional multiplexor stage at the beginning of the unit, the proper rotate amount can be calculated without incurring a delay penalty, as the computation should be complete by the time it is required. Since the rotator does not differ in theoretical gate count or critical path

latency, introduction of the unit into the comparison does not compromise the ability to evaluate the varying designs.

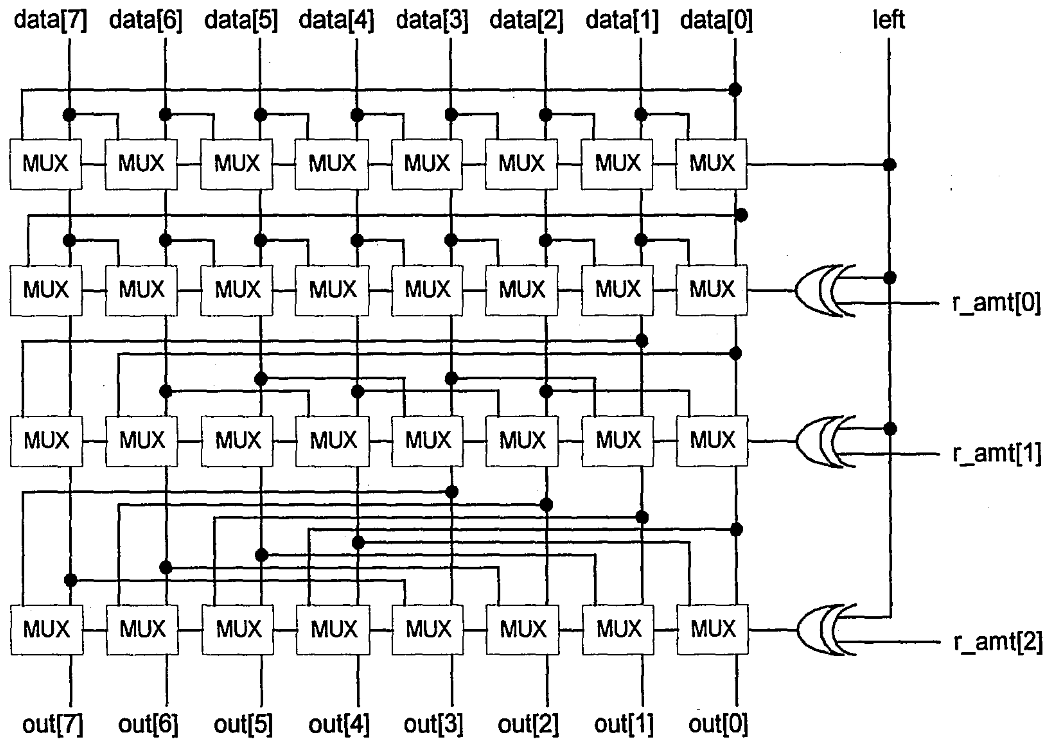

Figure 2.6: One's Complement Bi-directional Rotator

2.2.5 Masking Rotating Shifter

While the previous designs offer insight into the creation of an efficient bi-directional mechanism for shifts and rotates, they do not offer insight into how all the required operations can be implemented in a single unit. The Masking Rotating Shifter offers that. By using masks to manipulate the results of a rotate, it is possible to emulate logical and arithmetic shifts [6].

To accomplish this, the design starts by determining the rotation count, which is the amount the data must be rotated to the right, as shown in Figure 2.7. For left oriented operations, this value is the two's complement of the shift amount, otherwise, it is simply the original shift amount. This value is passed onto the mask decode stage and the rotator. The rotator is a single

right rotator. It is the rotator result that will be altered through the use of masks. The mask is computed in the mask decode stage. Setting the shift amount high order bits to one and the remaining bits to zero constructs the mask. Once the mask and the rotator result are known, they are passed into a logic stage that computes the final result through application of the mask to the rotator result. Using two control signals, the logic is able to manipulate the rotator result and the mask so as to compute the desired shift.

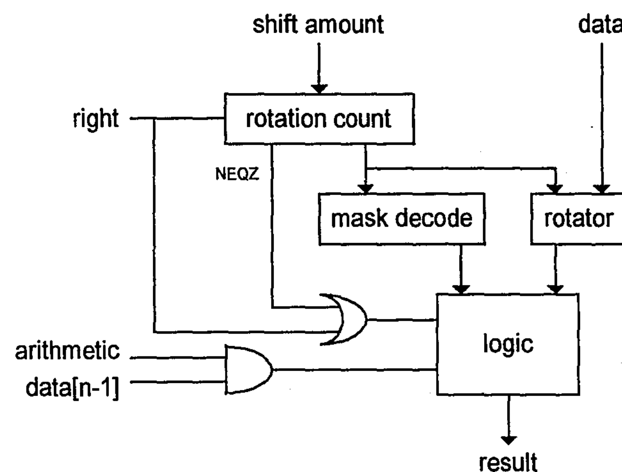

Figure 2.7: Masking Rotating Shifter

In particular, the shift right logical operation is computed by taking the bit-wise logical AND of the rotator result with the inverse of the mask. The shift left logical operation is computed by taking the bit-wise logical AND of the rotator result with the mask. The right shift arithmetic operation has similar computation details associated with it, however, the precise computation depends on the sign bit of the data. In the case where this bit is a one, the result is computed by taking the bit-wise logical OR of the rotator result with the mask. In the case where the sign bit is a zero, the result is computed by taking the bit-wise logical AND of the rotator result with the inverse of the mask, just as in the case of the shift right logical operation.

right rotator. It is the rotator result that will be altered through the use of masks. The mask is computed in the mask decode stage. Setting the shift amount high order bits to one and the remaining bits to zero constructs the mask. Once the mask and the rotator result are known, they are passed into a logic stage that computes the final result through application of the mask to the rotator result. Using two control signals, the logic is able to manipulate the rotator result and the mask so as to compute the desired shift.

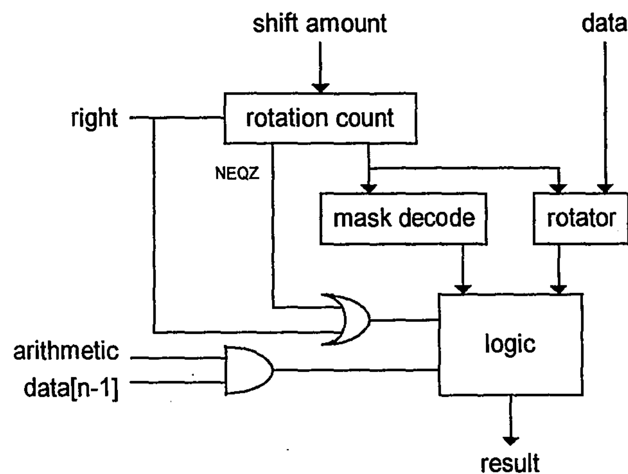

Figure 2.7: Masking Rotating Shifter

In particular, the shift right logical operation is computed by taking the bit-wise logical AND of the rotator result with the inverse of the mask. The shift left logical operation is computed by taking the bit-wise logical AND of the rotator result with the mask. The right shift arithmetic operation has similar computation details associated with it, however, the precise computation depends on the sign bit of the data. In the case where this bit is a one, the result is computed by taking the bit-wise logical OR of the rotator result with the mask. In the case where the sign bit is a zero, the result is computed by taking the bit-wise logical AND of the rotator result with the inverse of the mask, just as in the case of the shift right logical operation.

It is difficult to judge cost given so few implementation details. The approach to this design, however, is the most useful piece of information gained since it offers a method to design a complete barrel shifter, not yet explored.

2.3 Shift Operation Zero Flag

As previously discussed, the zero flag indicates when the result of an operation yields a value of zero. Typically, this calculation is performed at the end of the unit since a shift operation has the capability to do more than simply reorder the data bits. While this is certainly acceptable, it is inefficient since any delay associated with its calculation adds directly to the delay of the overall unit. Instead, the zero flag should be calculated in parallel with the result computation so that the delay penalty can be avoided. This is accomplished by first determining what bits from the input data exist in the result. Since the task of the barrel shifter is to determine both what bits will remain from the data and their position, there is considerable work to do. The computation for the zero flag, however, does not require that the position of the bits be known, only what bits will be in the result. By taking advantage of this distinction, it is possible to compute the zero flag in parallel to the result computation [7].

The process enabling such a calculation relies on masks. Essentially, a mask Z is used to suppress those bits that are known not to exist in the final result. The manner in which the mask Z is computed depends on the operation being carried out. For a shift right logical operation, the mask Z has shift/rotate amount high order bits set to zero with ones in the other bit positions. The bit-wise logical AND of the input data with the mask Z is then taken, thus forcing to zero those bits that do not exist in the result. The logical OR is taken of that result, which is then inverted to complete the zero flag computation. An output of one indicates a result of zero. This process is illustrated in Figure 2.8.

In order to adapt this method to the shift left logical operation, the mask Z is generated such that shift/rotate amount low order bits are set to zero with ones in the other bit positions. The remaining steps are identical to the right shift logical operation. The masks, as they apply to both logical shifts, are related in that each is the reverse of the other. This is due to the fact that the bits being removed are on opposite sides of the data vector.

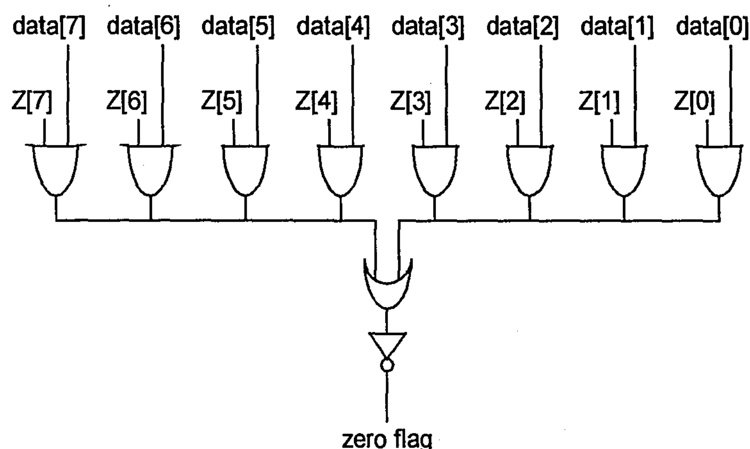

Figure 2.8: Zero Flag Calculation

The shift right arithmetic operation does not differ from the shift right logical operation. Since the maximum allowed shift amount is (n-1) bits, the sign bit of the data during a shift right arithmetic operation cannot be shifted off. Therefore, there is no need to take notice of the fact that either a zero or the sign bit is shifted in since it makes no difference to the computation of the zero flag.

2.4 Overflow Flag

The overflow flag is generally more complicated to compute than the zero flag. This is due to the fact that it does not represent the presence of a given state, but rather a transition. As

such, its calculation requires careful *attention* to the left shift process. There is a method, however, to implement this flag rather painlessly through the use of a mask.

This method, shown in Figure 2.9, selects either the input data or its inverse such that the most significant bit is a one. Knowing that the most significant bit is a one and that it is not needed later, however, implies that it does not need to be explicitly calculated, so it is not. Simultaneously, a mask F is computed. The mask is constructed by placing shift-amount zeros in the high order region while filling the low order region with ones. The bit-wise logical OR of the selected data with the mask is then taken. This process highlights overflow causing conditions. These (n-1) bits are logically AND'd together and then inverted [8]. This value represents the overflow flag. A value of one signifies that overflow has occurred. This process is illustrated in the example given in Table 2.2.

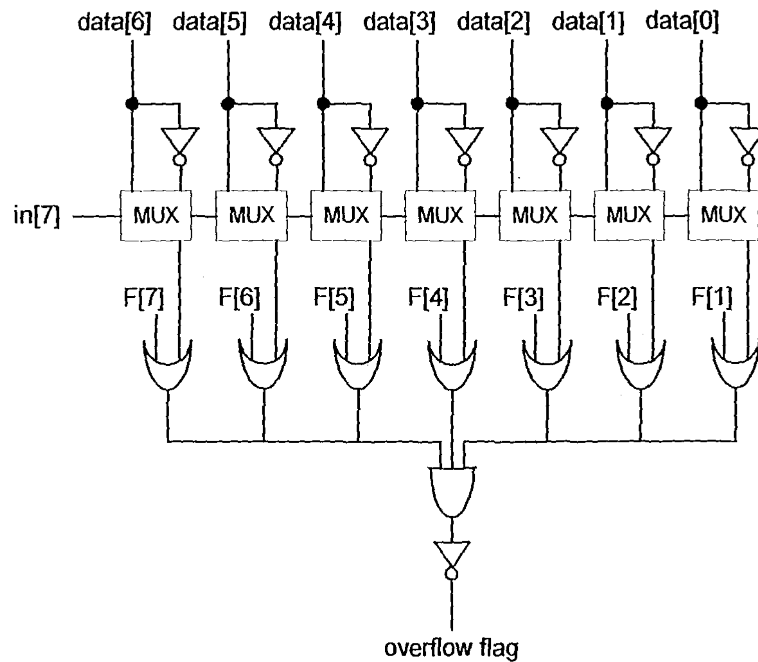

Figure 2.9: Previous Overflow Flag Calculation

It is also possible to implement this design such that the data is selected so that the most significant bit is a zero. Once again, knowing the value of the most significant bit in advance suggests that its calculation can be removed. The mask is then computed as having the high order region filled with shift-amount ones while the low order region is filled with zeros. This is the inverse of the previous mask computation. Instead of taking the bit-wise logical OR, the bit-wise logical AND of the selected data with the mask is taken. Likewise, this result is logically OR'd together. As before, an output of one indicates that overflow has occurred. An example of this process is shown in Table 2.3.

Operation	Data	Shift/Rotate Amount
Overflow Flag Calculation Method 1	11101101	2
Step	Value	
~Data	00010010	
Mux Data	1101101	
Calculate F	0011111	
(Mux Data)+F	1111111	
Calculate Overflow Flag	1111111->1->0	

Table 2.2: Overflow Flag Calculation Method 1 Example

Operation	Data	Shift/Rotate Amount
Overflow Flag Calculation Method 2	11101101	2
Step	Value	
~Data	00010010	
Mux Data	0010010	
Calculate F	1100000	
(Mux Data)*F	0000000	
Calculate Overflow Flag	0000000->0	

Table 2.3: Overflow Flag Calculation Method 2 Example

INTENTIONAL SECOND EXPOSURE

It is also possible to implement this design such that the data is selected so that the most significant bit is a zero. Once again, knowing the value of the most significant bit in advance suggests that its calculation can be removed. The mask is then computed as having the high order region filled with shift-amount ones while the low order region is filled with zeros. This is the inverse of the previous mask computation. Instead of taking the bit-wise logical OR, the bit-wise logical AND of the selected data with the mask is taken. Likewise, this result is logically OR'd together. As before, an output of one indicates that overflow has occurred. An example of this process is shown in Table 2.3.

Operation	Data	Shift/Rotate Amount
Overflow Flag Calculation Method 1	11101101	2
Step	**Value**	
~Data	00010010	
Mux Data	1101101	
Calculate F	0011111	
(Mux Data)+F	1111111	
Calculate Overflow Flag	1111111->1->0	

Table 2.2: Overflow Flag Calculation Method 1 Example

Operation	Data	Shift/Rotate Amount
Overflow Flag Calculation Method 2	11101101	2
Step	**Value**	
~Data	00010010	
Mux Data	0010010	
Calculate F	1100000	
(Mux Data)*F	0000000	
Calculate Overflow Flag	0000000->0	

Table 2.3: Overflow Flag Calculation Method 2 Example

Chapter 3 - Current Research

Current research examines four approaches to the problem of designing an optimal barrel shifter. As such, various design techniques will be applied in an attempt to understand what optimizations provide the most improvement. Many of these optimizations will have roots in previous designs, but are expanded to satisfy the imposed demands.

Each of the following four designs take three inputs: an *n*-bit data vector where *n* is an integer power of two, a $\log_2(n)$-bit shift/rotate (S/R) amount vector, and a 3-bit operation code (opcode) vector. Each bit from the opcode designates a specific aspect of the operation. In particular, the separation is right/left, rotate/shift, and arithmetic/logical. The output of each design varies somewhat, as some implement flags that others do not. All output, however, an *n*-bit result vector. The two flags that are occasionally added are the zero flag and the overflow flag, each 1 bit.

3.1 Mux-based Data Reversal

3.1.1 Design Overview

The Mux-based Data Reversal design utilizes some of the simpler design ideas reviewed. Due to this, it represents a conservative approach to barrel shifter design as it strives for cost minimization. As such, it is this model that all other designs are compared to in the forthcoming analysis. As the name reveals, this design primarily utilizes multiplexors in addition to the aforementioned data reversal mechanism to carryout its operations.

As shown in Figure 3.1, the design overview is very straightforward. This is due in part to the fact that the R calculation encompasses the majority of the desired functionality. It is this unit that actually implements the shifts and rotates. The mux data reversal unit of the design helps to simplify this computation, as does the S calculation.

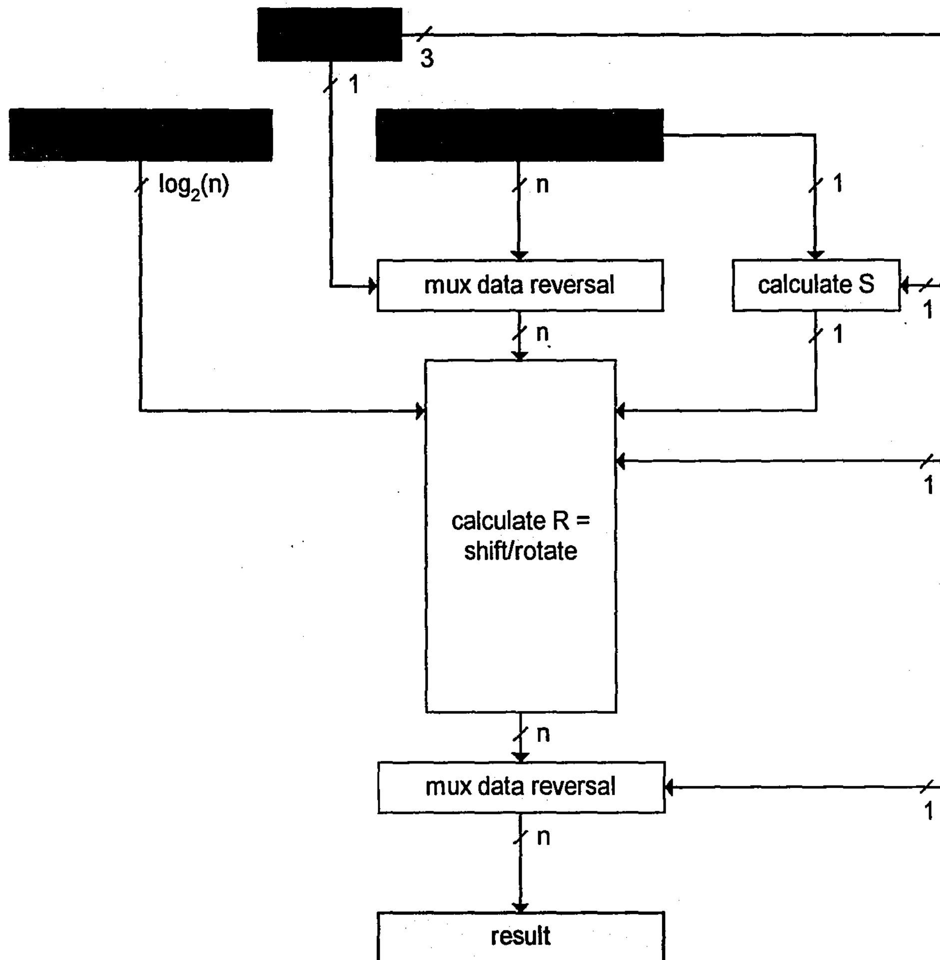

Figure 3.1: Mux-based Data Reversal

As previously mentioned, the mux data reversal units reverse the order of the input data so that a left oriented operation may be emulated through the use of right oriented hardware. Likewise, the result is reversed so as to undo the original reversal. The S calculation computes the value that will be used to fill in the vacated regions during a shift. For logical shifts this is a zero, while for the arithmetic right shift, it is dependent on the sign bit of the data. In particular, S is zero in all cases except when both the sign bit of the data is a one and the operation is an arithmetic right shift.

INTENTIONAL SECOND EXPOSURE

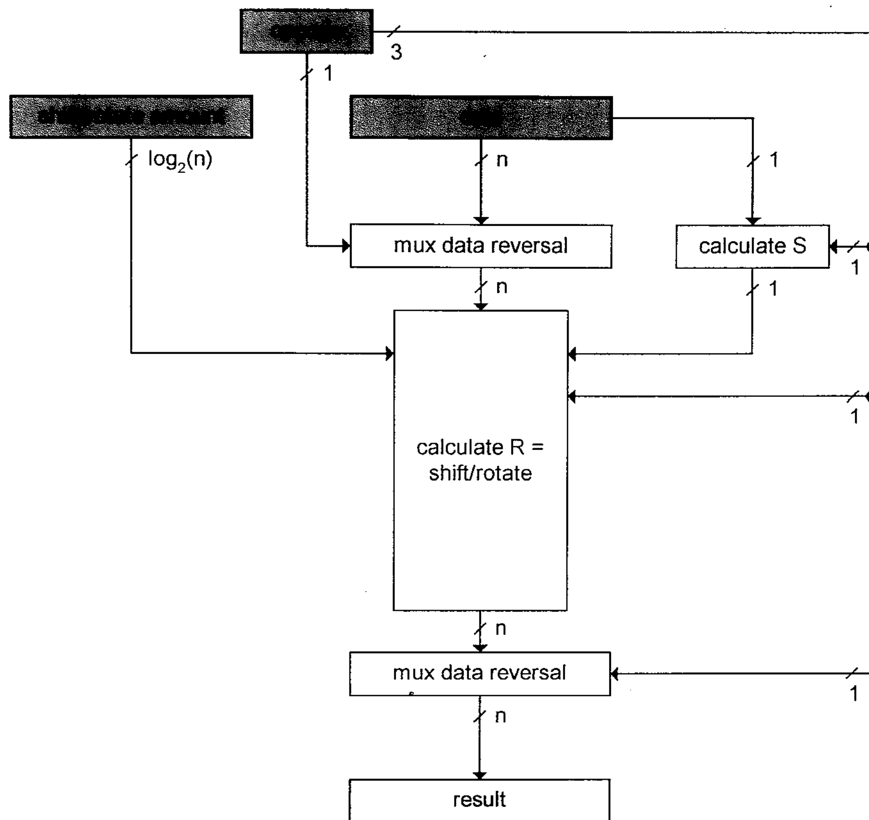

Figure 3.1: Mux-based Data Reversal

As previously mentioned, the mux data reversal units reverse the order of the input data so that a left oriented operation may be emulated through the use of right oriented hardware. Likewise, the result is reversed so as to undo the original reversal. The S calculation computes the value that will be used to fill in the vacated regions during a shift. For logical shifts this is a zero, while for the arithmetic right shift, it is dependent on the sign bit of the data. In particular, S is zero in all cases except when both the sign bit of the data is a one and the operation is an arithmetic right shift.

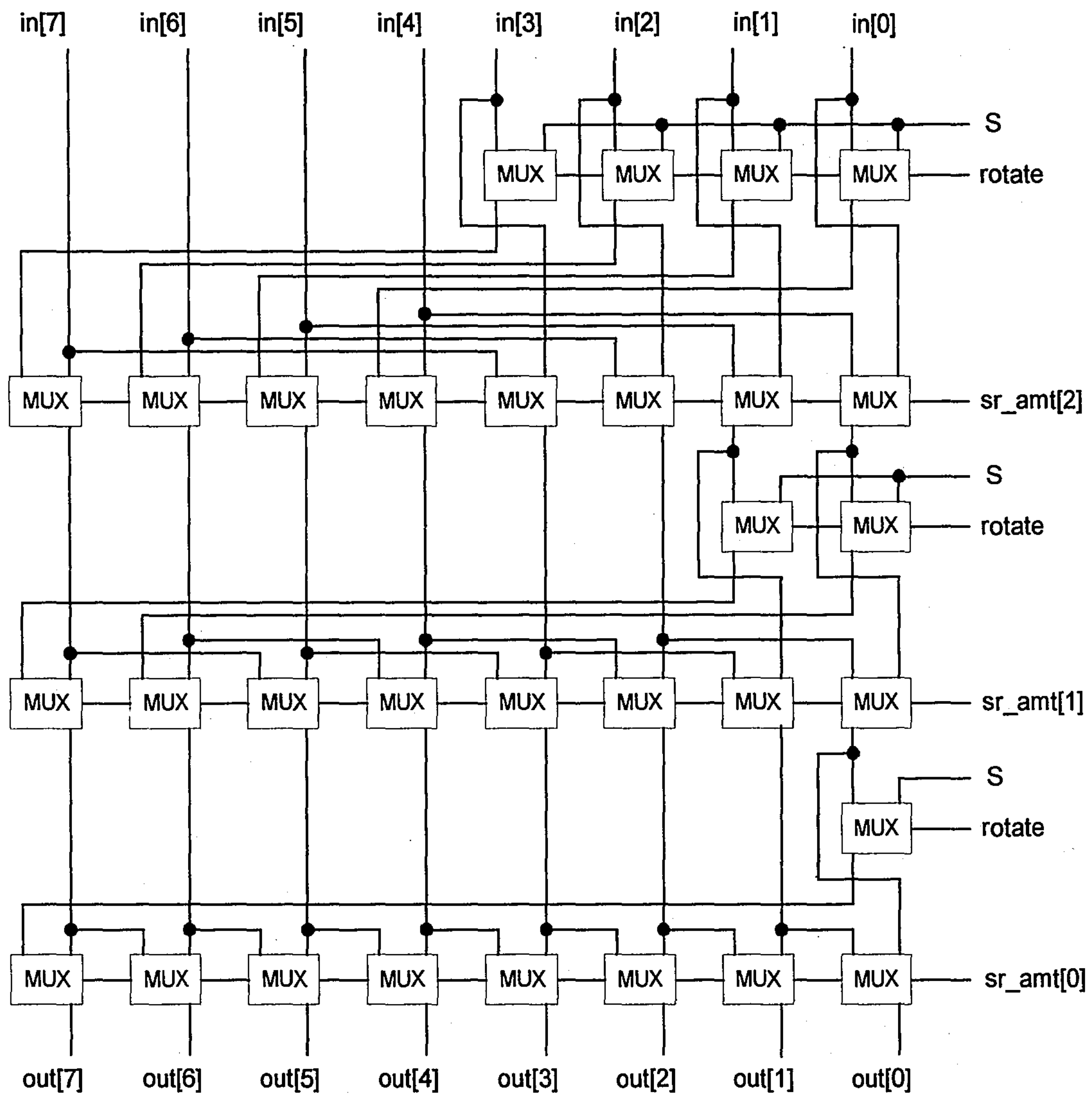

Figure 3.2: Right Shifter/Rotator

The right shifter/rotator, as shown in Figure 3.2, is a combination of the logical right shifter and the right rotator with alterations made enabling the arithmetic right shift operation. This is accomplished by adding a pad calculation before every row of multiplexors. This computation determines which bits are used to fill voids created in the high order region by the shifting of data out of the low order region. In the case of a rotate, the bits shifted out are placed in the high order region and thus constitute the pad. For a shift, the fill bit, S, makes up the pad.

The pad, once calculated, is then used during any non-zero shift/rotate on that level. In particular, the pad makes up the high order region of the multiplexed result, while the low order region is filled with those bits of the input not forced out by the shift. If the shift/rotate amount is zero, then the input is simply passed through. This process repeats itself $\log_2(n)$ times. Each time, the pad size is adjusted to match the particular shift/rotate amount of that particular row. Once this process is complete, the result is sent to a mux data reversal unit that reverses the actions of the first mux data reversal unit. In this manner, the result is put into the correct form.

3.1.2 Zero Flag

The zero flag often adds additional delay to the design since its computation is typically based on the result. In a few cases, however, including this one, it is possible to begin calculation of the zero flag before the final result is known and without costly additional logic. This is possible since the final step, the data reversal, does not alter the contents of the result, just the order. Therefore, computation of the zero flag can begin once the shifter/rotator is complete, as shown in Figure 3.3.

While the computation is allowed to begin earlier than it would otherwise have started, it does not necessitate that the zero flag is known before or at the time of completion of the mux data reversal unit. It is still very much possible, and highly probable in cases where n is of even moderate size, that the zero flag computation will increase the design's overall delay. This is due to the implementation of both the mux data reversal and zero flag units. The mux data reversal unit, as mentioned, uses a single row of multiplexors. The zero flag, however, uses a tree of logical OR gates followed by a NOT gate so that an output of one signals a zero result. As such, there exist many sizes of n that would require a logical OR tree of a depth greater than that of a multiplexor, thus increasing the critical path delay of the unit as a whole. Therefore, while the

calculation of the zero flag is optimized in terms of the delay it imposes, it is not possible to completely erase its impact.

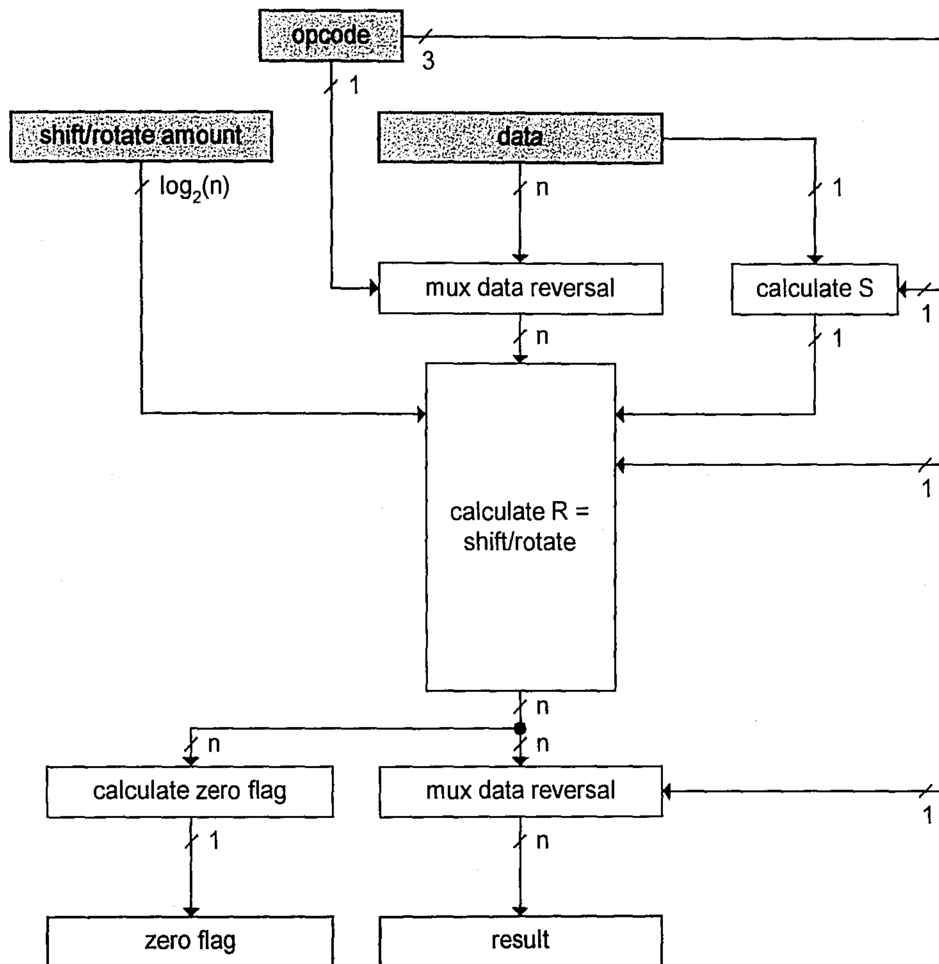

Figure 3.3: Mux-based Data Reversal with Zero Flag

3.1.3 Overflow Flag

Unlike the zero flag with its simple computation, the overflow flag is quite complicated to compute due to its measure of a transition, not a state. As such, it needs to be included in the internal workings of the shifter/rotator so that it can monitor the shift/rotate process with the least amount of additional hardware. To this end, it has been integrated into the unit as shown in Figure 3.4. The rest of the design remains the same.

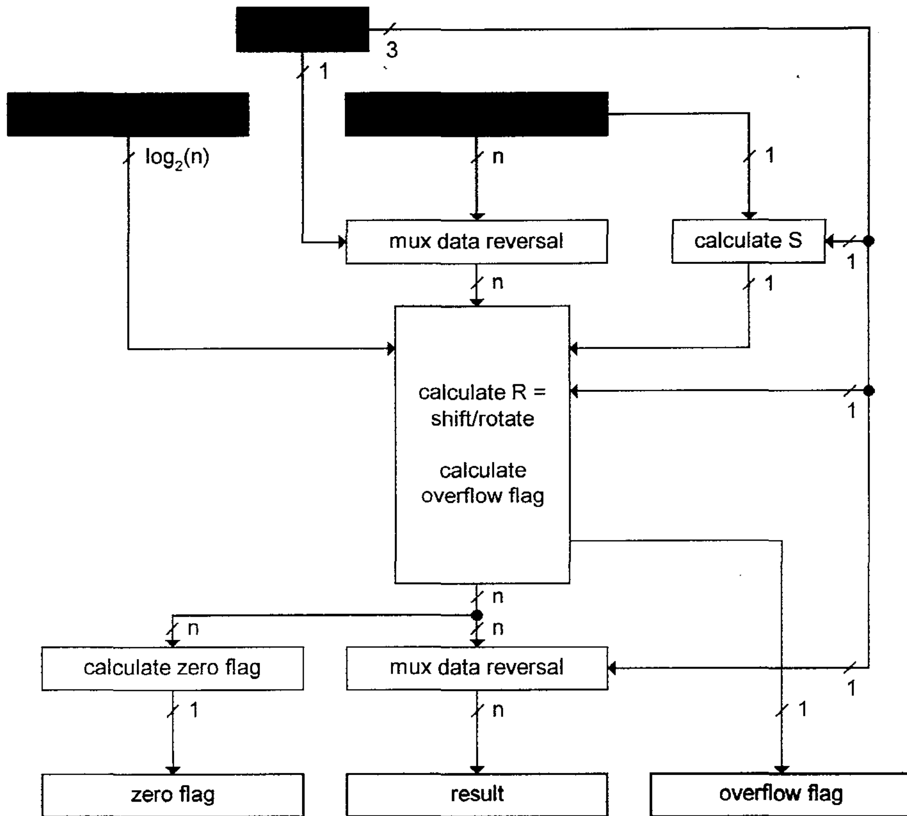

Figure 3.4: Mux-based Data Reversal with Zero and Overflow Flags

In order to add the overflow computation to the shifter/rotator without increasing the delay, it is necessary to detect overflow in parallel with the shift/rotate operation. This process is detailed in Figure 3.5. Here, it can be seen that the overflow flag is computed in a series of steps corresponding to the breakdown within the shifter/rotator. Each step uses a number of multiplexors equal to the number used in the pad computation at the same level. These multiplexors, during a shift, select those bits that pass beyond or onto the data's sign bit, which during a shift left operation inhabits the least significant bit position of the input to the right

INTENTIONAL SECOND EXPOSURE

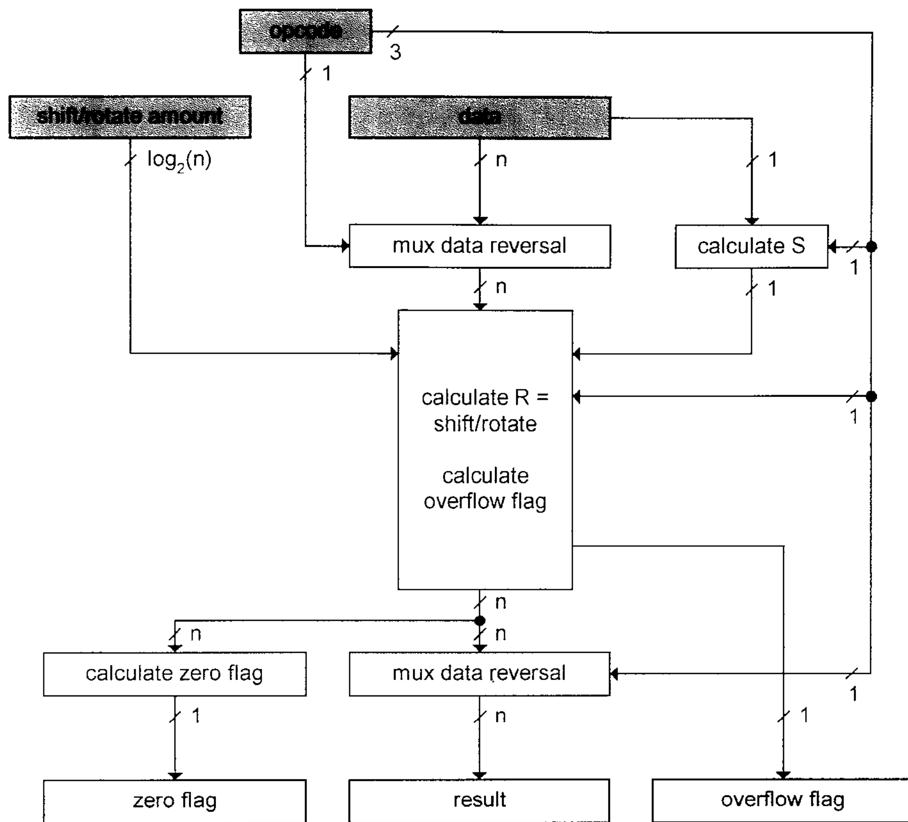

Figure 3.4: Mux-based Data Reversal with Zero and Overflow Flags

In order to add the overflow computation to the shifter/rotator without increasing the delay, it is necessary to detect overflow in parallel with the shift/rotate operation. This process is detailed in Figure 3.5. Here, it can be seen that the overflow flag is computed in a series of steps corresponding to the breakdown within the shifter/rotator. Each step uses a number of multiplexors equal to the number used in the pad computation at the same level. These multiplexors, during a shift, select those bits that pass beyond or onto the data's sign bit, which during a shift left operation inhabits the least significant bit position of the input to the right

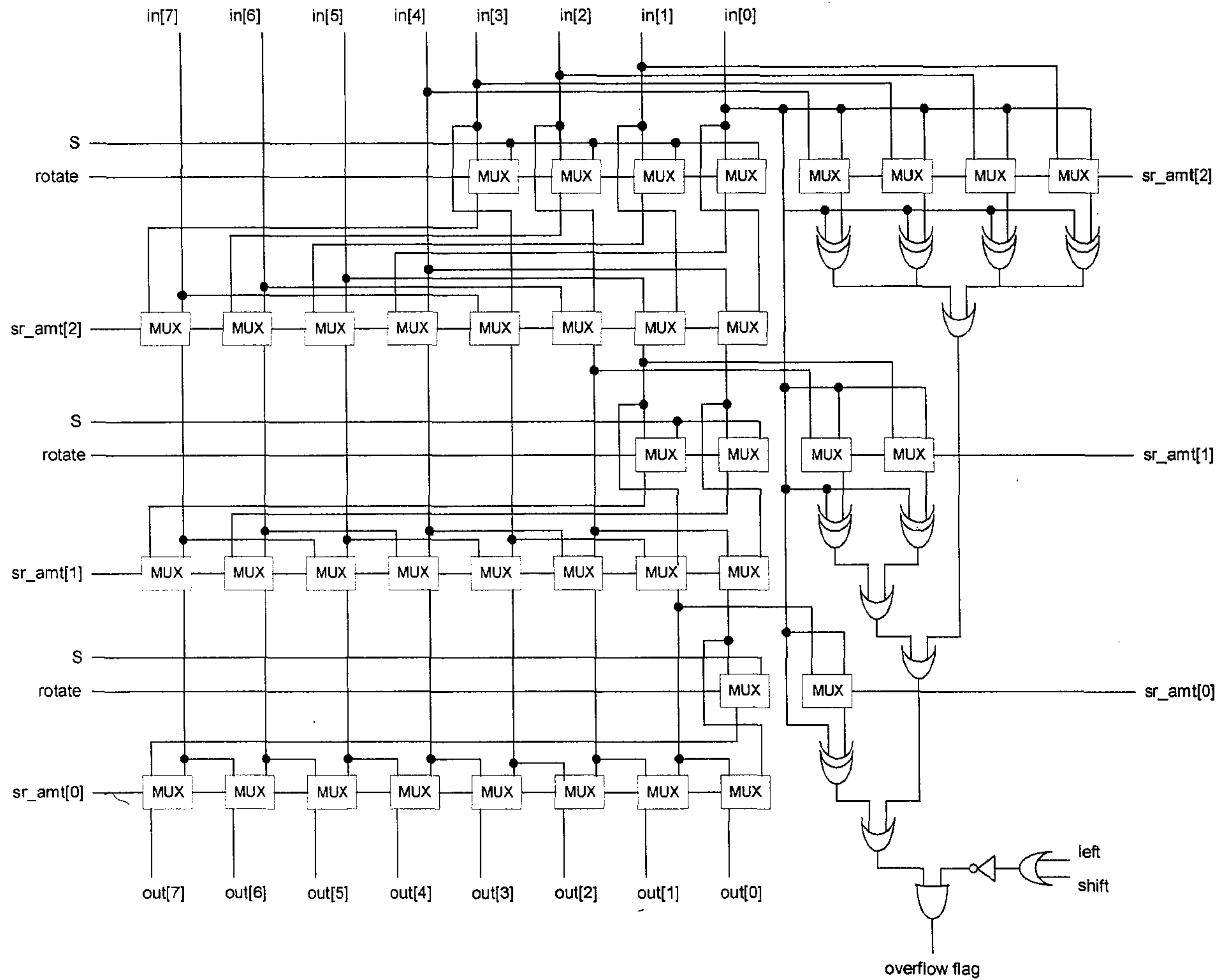

Figure 3.5: Right Shifter/Rotator with Overflow Flag

shifter/rotator. In those cases where a shift does not involve a particular level, the multiplexors select the sign bit. The selected bits are then logically XOR'd with the sign bit. In this manner, only when both a shift occurs and a bit differs from the sign bit will overflow be signaled. As such, these bits are logically OR'd together so as to detect any overflow condition existing at the particular level.

This process is repeated for each level of the shifter/rotator. Each level result is then logically OR'd with the result from the previous level and passed to the next level until all bits have essentially been logically OR'd together. A high order result signals that overflow has occurred given that the operation is a left shift.

It is not by coincidence that examination of this structure reveals that delay is minimized along the overflow computation path. This is achieved by placing the most computationally involved overflow level at the top, to be performed first. In this manner, it has ample time to complete since its result is not needed immediately. This is due to the fact that the subsequent level has its own work to do before it is ready for the result. The final level involves the least amount of work since it needs only to be logically OR'd with the result of the previous level, which by this point should be complete. To build this structure in this style requires that the shifter/rotator be designed with the overflow computation in mind. One can see that this is not a trivial design layout, but rather one that has been designed with cost and delay in mind from the start.

3.1.4 Examples

Now that the mechanisms behind the design are known, a few examples are shown to demonstrate the exact manner in which each operation is performed. Each example shows the

outputs of the subunits from the design diagrams so that the manner in which the result is computed is easily seen.

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Rotate Right	10100110	2
Step	Value		
Mux data reversal	10100110		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate R = Shift/Rotate	10101001		
Mux data reversal	10101001		
Result	10101001		
Calculate Zero Flag	0		
Calculate Overflow Flag	0		

Table 3.1: Mux-based Data Reversal Rotate Right Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Rotate Left	10100110	2
Step	Value		
Mux data reversal	01100101		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate R = Shift/Rotate	01011001		
Mux data reversal	10011010		
Result	10011010		
Calculate Zero Flag	0		
Calculate Overflow Flag	0		

Table 3.2: Mux-based Data Reversal Rotate Left Example

INTENTIONAL SECOND EXPOSURE

outputs of the subunits from the design diagrams so that the manner in which the result is computed is easily seen.

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Rotate Right	10100110	2
Step	**Value**		
Mux data reversal	10100110		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate R = Shift/Rotate	10101001		
Mux data reversal	10101001		
Result	10101001		
Calculate Zero Flag	0		
Calculate Overflow Flag	0		

Table 3.1: Mux-based Data Reversal Rotate Right Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Rotate Left	10100110	2
Step	**Value**		
Mux data reversal	01100101		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate R = Shift/Rotate	01011001		
Mux data reversal	10011010		
Result	10011010		
Calculate Zero Flag	0		
Calculate Overflow Flag	0		

Table 3.2: Mux-based Data Reversal Rotate Left Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Shift Right Logical	10100110	2

Step	Value
Mux data reversal	10100110
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate R = Shift/Rotate	00101001
Mux data reversal	00101001
Result	00101001
Calculate Zero Flag	0
Calculate Overflow Flag	0

Table 3.3: Mux-based Data Reversal Shift Right Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Shift Left Logical	10100110	2

Step	Value
Mux data reversal	01100101
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate R = Shift/Rotate	00011001
Mux data reversal	10011000
Result	10011000
Calculate Zero Flag	0
Calculate Overflow Flag	1

Table 3.4: Mux-based Data Reversal Shift Left Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Shift Right Arithmetic	10100110	2

Step	Value
Mux data reversal	10100110
Calculate S = arithmetic*data[n-1]	1*1=1
Calculate R = Shift/Rotate	11101001
Mux data reversal	11101001
Result	11101001
Calculate Zero Flag	0
Calculate Overflow Flag	0

Table 3.5: Mux-based Data Reversal Shift Right Arithmetic Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Shift Right Logical	10100110	2
Step	**Value**		
Mux data reversal	10100110		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate R = Shift/Rotate	00101001		
Mux data reversal	00101001		
Result	00101001		
Calculate Zero Flag	0		
Calculate Overflow Flag	0		

Table 3.3: Mux-based Data Reversal Shift Right Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Shift Left Logical	10100110	2
Step	**Value**		
Mux data reversal	01100101		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate R = Shift/Rotate	00011001		
Mux data reversal	10011000		
Result	10011000		
Calculate Zero Flag	0		
Calculate Overflow Flag	1		

Table 3.4: Mux-based Data Reversal Shift Left Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mux-based Data Reversal	Shift Right Arithmetic	10100110	2
Step	**Value**		
Mux data reversal	10100110		
Calculate S = arithmetic*data[n-1]	1*1=1		
Calculate R = Shift/Rotate	11101001		
Mux data reversal	11101001		
Result	11101001		
Calculate Zero Flag	0		
Calculate Overflow Flag	0		

Table 3.5: Mux-based Data Reversal Shift Right Arithmetic Example

3.2 Mask-based Data Reversal

3.2.1 Design Overview

The Mask-based Data Reversal design is a combination of the Data Reversal Bi-directional Logical Shifter and the Masking Rotating Shifter. It utilizes the data reversal mechanism to emulate left oriented operations with right oriented hardware. In addition, the masking operations are used to derive all results from some base form. The base form is that of the rotate since it maintains all bits throughout its computation and correctly orders those bits. Application of the mask manipulates the base form so as to replicate shift operations. A rotate operation requires no manipulation.

Figure 3.6 illustrates a block overview of the design. Many of the components from the previous design are carried over. As mentioned, this design utilizes a mux data reversal unit that is the same as before. In addition, the calculation of the fill bit, S, is also done in the same manner. This is, however, where the similarities end. The most clearly visible change is the creation of a computation path in parallel with the rotate. This path computes the mask P that will ultimately be applied to the rotate result. Since a multitude of factors must be considered in the mask computation, it is constructed in a step manner to highlight each contributing factor.

The first component, mask F, is an n-bit vector containing shift/rotate amount zeros left justified and the remaining bits filled with ones. This mask corresponds directly to a logical right shift since its application will void bits by forcing them to zero. To make this flag universally applicable, so as to later avoid a selection mechanism, requires that it be modified further. This modification is made in the calculation of P. The mask P is the result of taking the bit-wise logical OR of the mask F with the rotate signal, P=F+rotate. In this manner, if the operation is a rotate, then application of the mask makes no changes to the rotate result.

Finally, an intermediate result T is calculated which is the final result without the second mux data reversal unit applied to undo the first. It is within this calculation that the mask is applied to the rotate result. T is computed by first taking the bit-wise logical OR of the mask P with the rotate result R. Simultaneously, the logical AND of S with the inverse of P, $\bar{P}$, is taken. These two results are then bit-wise logically OR'd together, $T = R*P+S*\bar{P}$.

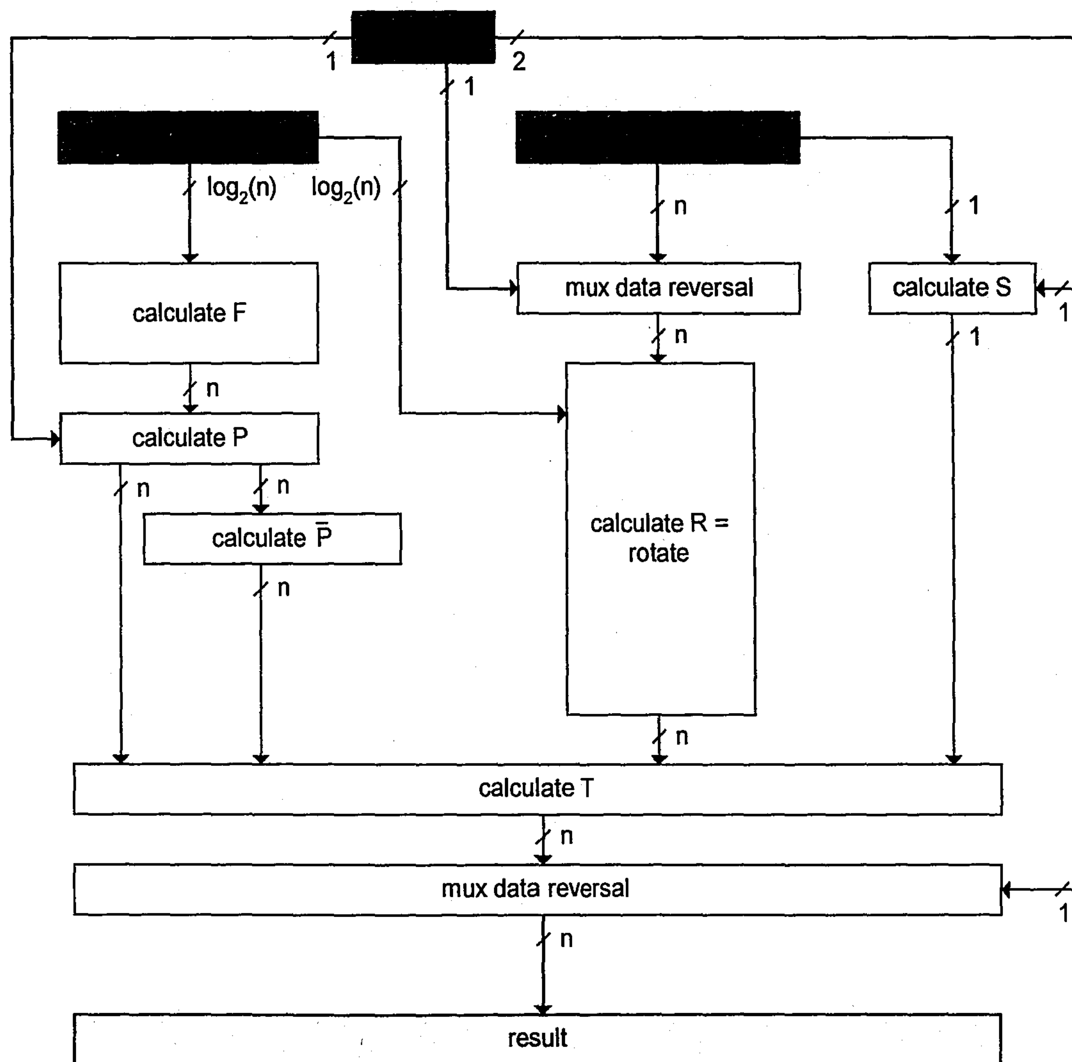

Figure 3.6: Mask-based Data Reversal

In this manner, P is used to force to zero those bits from the rotate result corresponding to shifted in zeros in the final result. Likewise, $\bar{P}$ is used to fill those bit positions voided by the

INTENTIONAL SECOND EXPOSURE

Finally, an intermediate result T is calculated which is the final result without the second mux data reversal unit applied to undo the first. It is within this calculation that the mask is applied to the rotate result. T is computed by first taking the bit-wise logical OR of the mask P with the rotate result R. Simultaneously, the logical AND of S with the inverse of P, $\overline{P}$, is taken. These two results are then bit-wise logically OR'd together, $T = R*P+S*\overline{P}$.

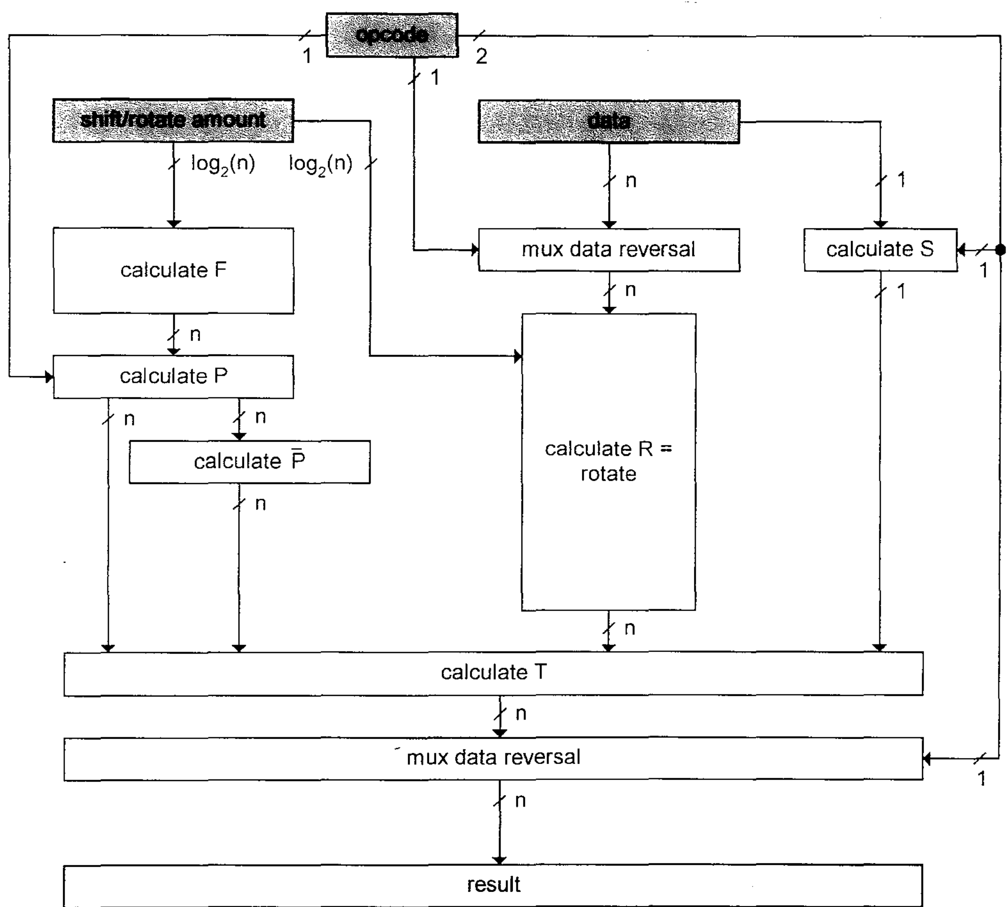

Figure 3.6: Mask-based Data Reversal

In this manner, P is used to force to zero those bits from the rotate result corresponding to shifted in zeros in the final result. Likewise, $\overline{P}$ is used to fill those bit positions voided by the

application of P with the value of S. As a reminder, S is only one when both the sign bit of the data is a one and the operation is an arithmetic shift. As such, this part of the calculation of T only alters the result if a shift right arithmetic operation is being performed. For a rotate operation, $\overline{P}$ is the zero vector since P is a vector of ones.

Therefore, all operation calculations pass through the mask application despite their differences. This is possible since the applied mask is built around the current operation. Once the calculation of the intermediate result T is known, it is passed to a mux data reversal unit to undo the previous mux data reversal unit.

3.2.2 Mask F Generator

As was previously mentioned, the mask F is composed of shift/rotate amount zeros left justified with the remaining bits filled with ones. The mechanism to create this value relies upon the shift/rotate amount to create the mask F. The generator of the mask F utilizes a recursive structure. With this approach, creating an n-bit mask F requires first that an $(n/2)$-bit mask F be generated. Figure 3.7 shows the structure used to create a 16-bit mask. It is clearly visible that the structure begins by using the inverse of the least significant bit from the shift/rotate amount. It is also clear that a one always constitutes the least significant bit of the mask F. This bit is not, however, used in the recursive structure but instead concatenated upon mask completion.

If we consider the construction of a 2-bit mask for data of the same length and take into account the restriction that the maximum shift amount is equal to $(n-1)$, then one can see how the mask is simply the inverse of the shift amount bit followed by a one. There are only two possibilities for the mask and the same number for the shift amount. In this manner, construction of a 2-bit mask F is complete and the base case for the recursive structure is set. An n-bit mask is computed using the previous mask minus the appended one. This value is used twice. First, each

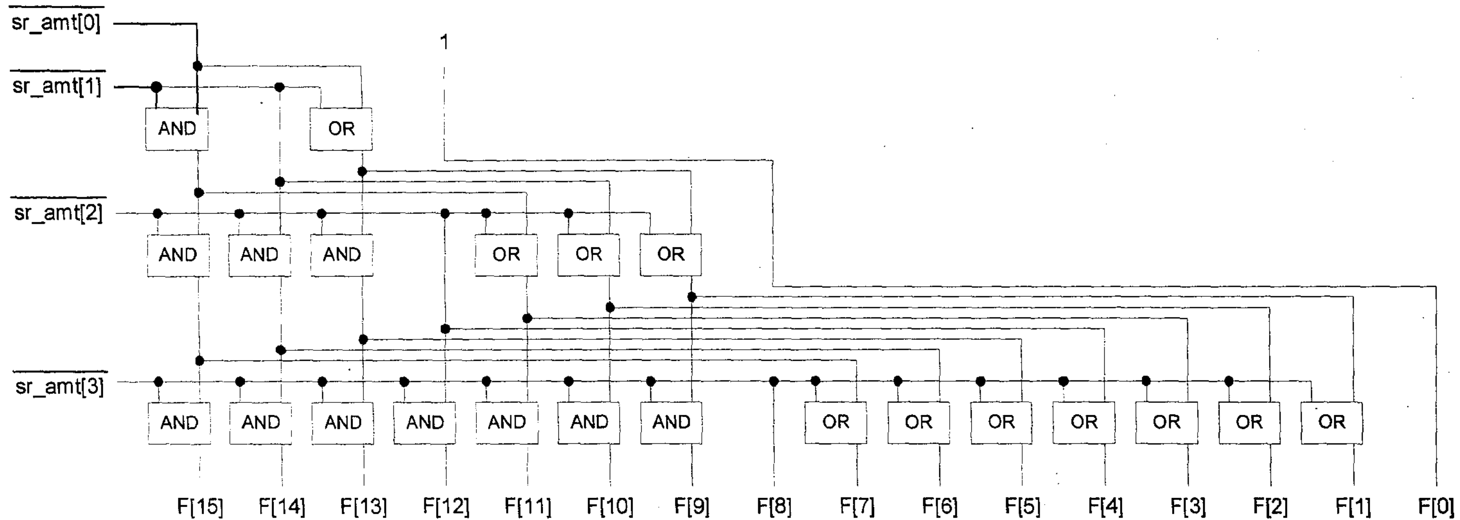

Figure 3.7: Mask F Generator

bit is logically AND'd with the inverse of the next least significant bit from the shift/rotate amount. Second, each bit is logically OR'd with the inverse of the next least significant bit from the shift/rotate amount. The second intermediate result is then concatenated to the first with the inverse of the next least significant shift/rotate amount bit occupying a position between them. Finally, concatenation of a one onto the low order side completes construction of any n-bit mask F. Now that both the base condition and the intermediate steps have been described, it is easy to see how the mask is constructed.

3.2.3 Zero Flag

The zero flag for this design could be implemented in the exact same manner as was done in the Mux-based Data Reversal design. To do so, however, would be inefficient since there exists a mechanism to perform this calculation even earlier, so as to completely avoid additional latency, without requiring a substantial increase in cost. This mechanism is a direct derivation of that used in the Shift Operation Zero Flag, described in Section 2.3. The difference between that mechanism and the one employed here is that instead of manipulating the mask so that it applies to the operation, the data the mask is applied to is manipulated. In particular, advantage is taken of the fact that the mux data reversal unit is already formatting the data so that it applies to a right oriented operation. In this manner, additional work is minimized.

Therefore, the zero flag calculation is the same as that shown in Figure 2.8, but with the input data replaced with the result of the mux data reversal unit and the mask Z computed as the reverse of the mask P. The reverse of the mask P is used since it is applied to the mux data reversal unit's output before it has been rotated. As such, the bits that will be voided are on the side opposite to which the mask P is designed for. As before, an output of one signals a zero result. Figure 3.8 shows the integration of the zero flag unit into the design.

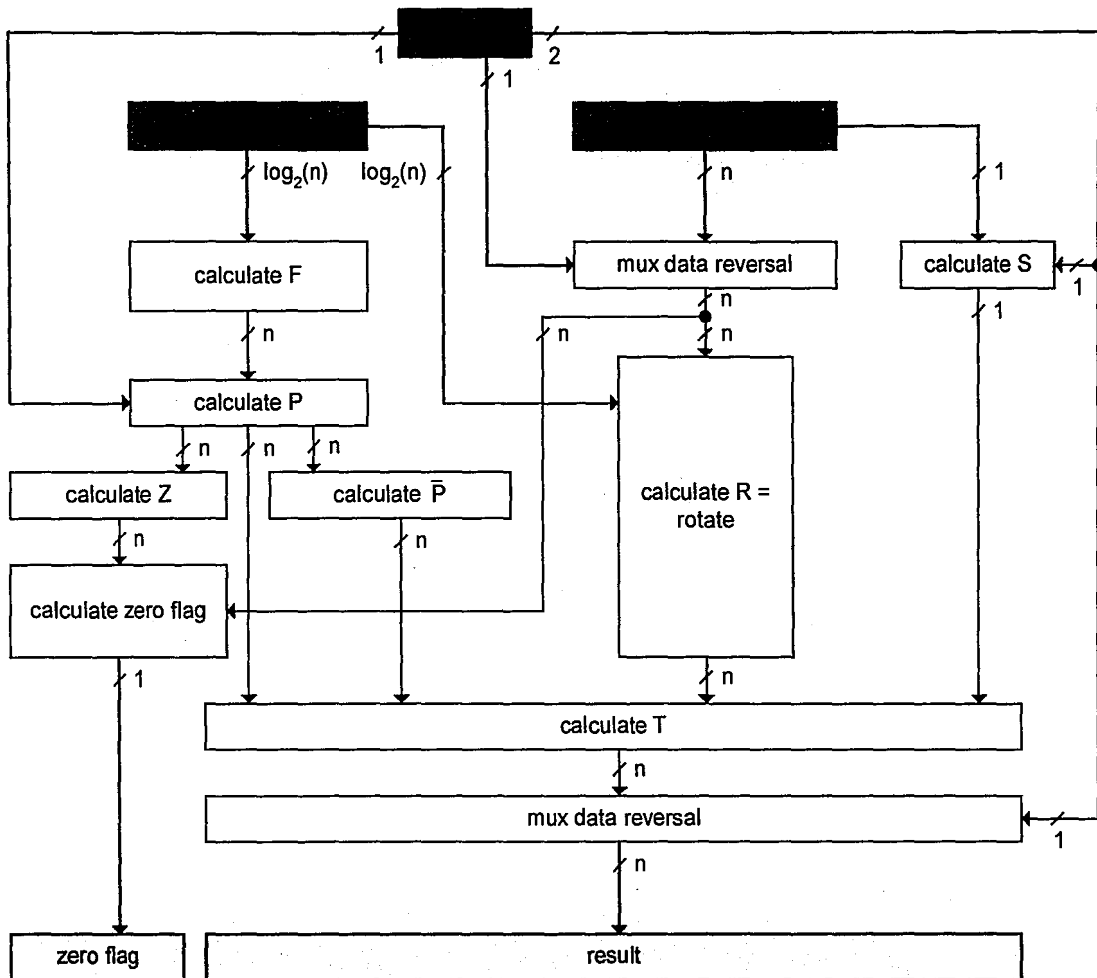

Figure 3.8: Mask-based Data Reversal with Zero Flag

3.2.4 Overflow Flag

The overflow flag, like the zero flag, can benefit from the use of masks. One such method was presented previously. An enhancement of that method, however, has been implemented instead. This overflow computation method is derived from the observation that only a few select bits are able to cause an overflow condition to arise during a left shift. As such, if these bits can be isolated, then they can be examined for signaling an overflow. To this end, the mask is used to illuminate those bits that are of concern.

INTENTIONAL SECOND EXPOSURE

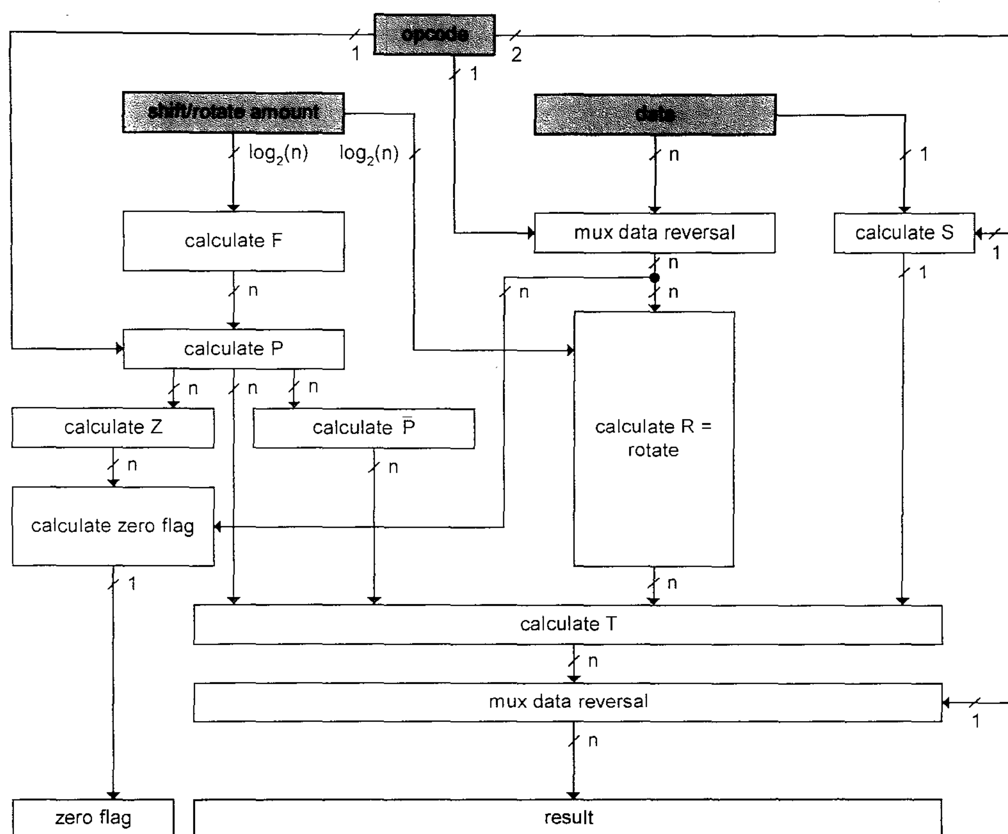

Figure 3.8: Mask-based Data Reversal with Zero Flag

3.2.4 Overflow Flag

The overflow flag, like the zero flag, can benefit from the use of masks. One such method was presented previously. An enhancement of that method, however, has been implemented instead. This overflow computation method is derived from the observation that only a few select bits are able to cause an overflow condition to arise during a left shift. As such, if these bits can be isolated, then they can be examined for signaling an overflow. To this end, the mask is used to illuminate those bits that are of concern.

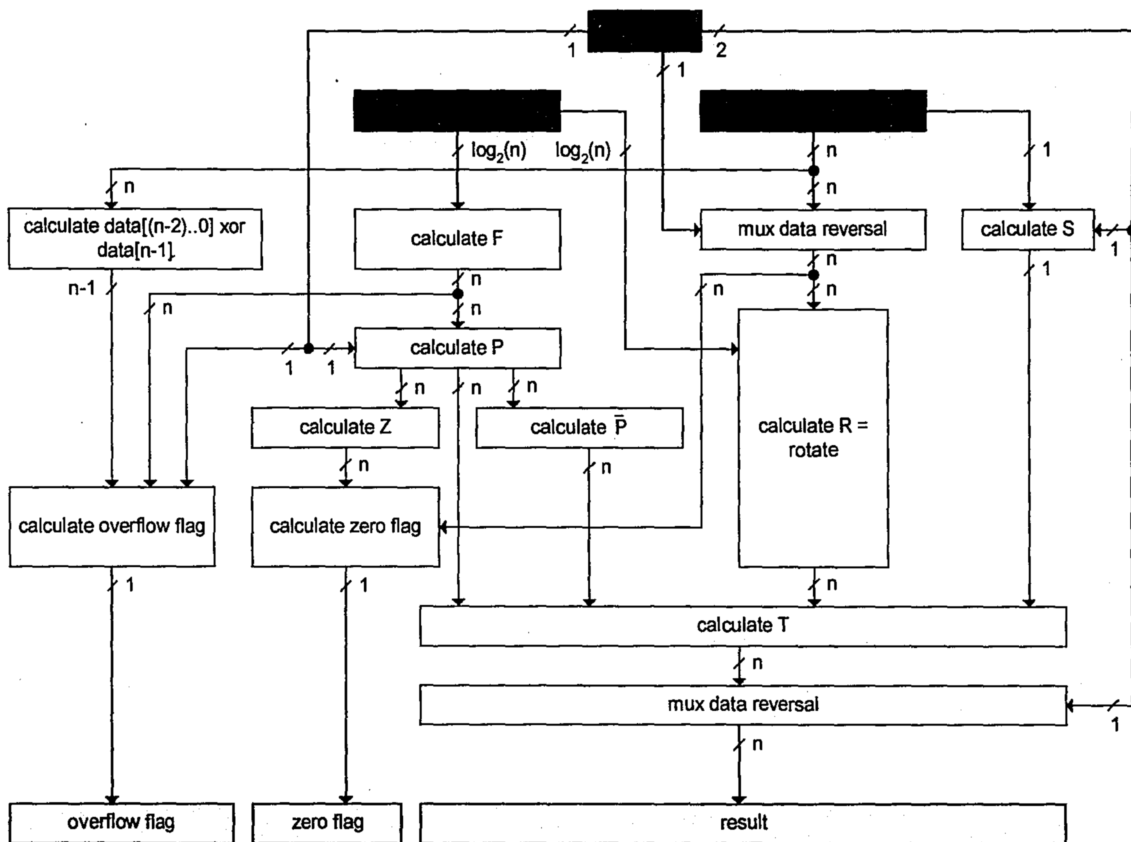

Figure 3.9: Mask-based Data Reversal with Zero and Overflow Flags

Figure 3.9 reveals the manner in which the overflow computation has been inserted into the design. It resides in a path parallel to that of the rotate and mask generation units. The computation is broken into two stages. The first step of the process is to calculate the logical XOR of the (n-1) low order data bits with the sign bit of the data. This operation determines what bits could cause an overflow if passed over the most significant bit during a logical left shift. In the second stage, this result is then logically AND'd with the inverse of the (n-1) high order bits of the mask F. The mask F has shift/rotate amount left justified zeros, which means that its inverse has shift/rotate amount left justified ones.

By shifting this value to the right by one, a value is obtained where the ones are in the same bit positions as those bits from the data that will ultimately pass onto or beyond the sign bit of the data. The logical AND operation thus highlights those positions that may cause an

INTENTIONAL SECOND EXPOSURE

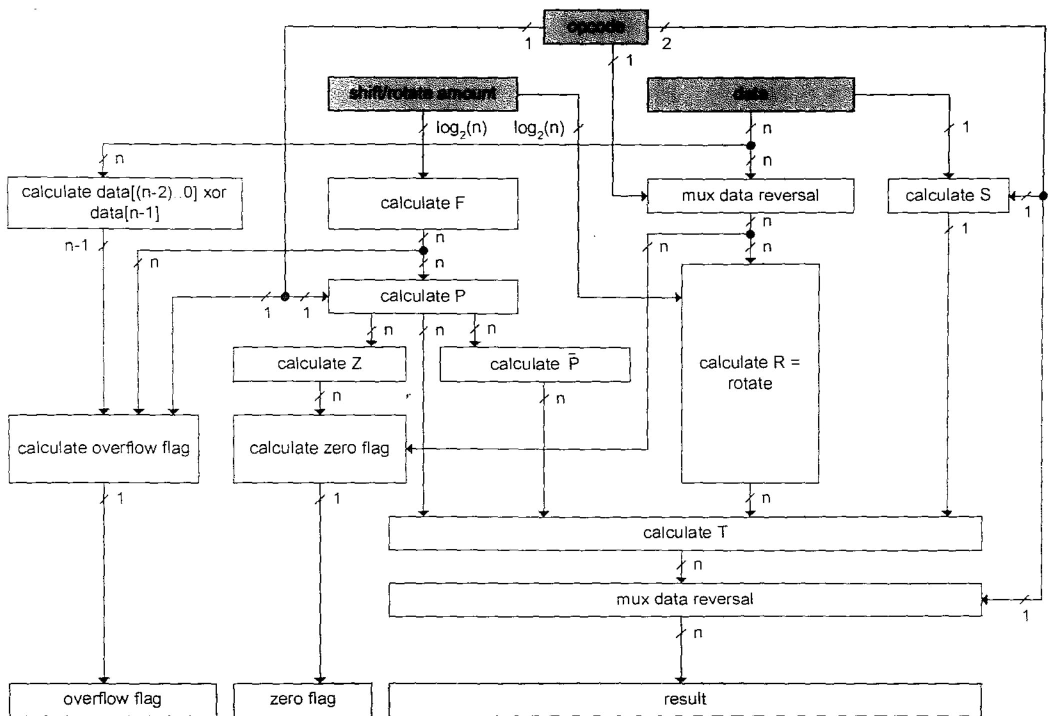

Figure 3.9: Mask-based Data Reversal with Zero and Overflow Flags

Figure 3.9 reveals the manner in which the overflow computation has been inserted into the design. It resides in a path parallel to that of the rotate and mask generation units. The computation is broken into two stages. The first step of the process is to calculate the logical XOR of the (n-1) low order data bits with the sign bit of the data. This operation determines what bits could cause an overflow if passed over the most significant bit during a logical left shift. In the second stage, this result is then logically AND'd with the inverse of the (n-1) high order bits of the mask F. The mask F has shift/rotate amount left justified zeros, which means that its inverse has shift/rotate amount left justified ones.

By shifting this value to the right by one, a value is obtained where the ones are in the same bit positions as those bits from the data that will ultimately pass onto or beyond the sign bit of the data. The logical AND operation thus highlights those positions that may cause an

overflow. The logical OR network that follows then determines if an overflow has occurred. This result must then be logically AND'd with the left shift signal. All of this can be seen in Figure 3.10. Table 3.6 reveals synthesis results for this and the previous approach described in Section 2.4. As can be seen, the current method not only has a lower cost, but a reduced latency as well.

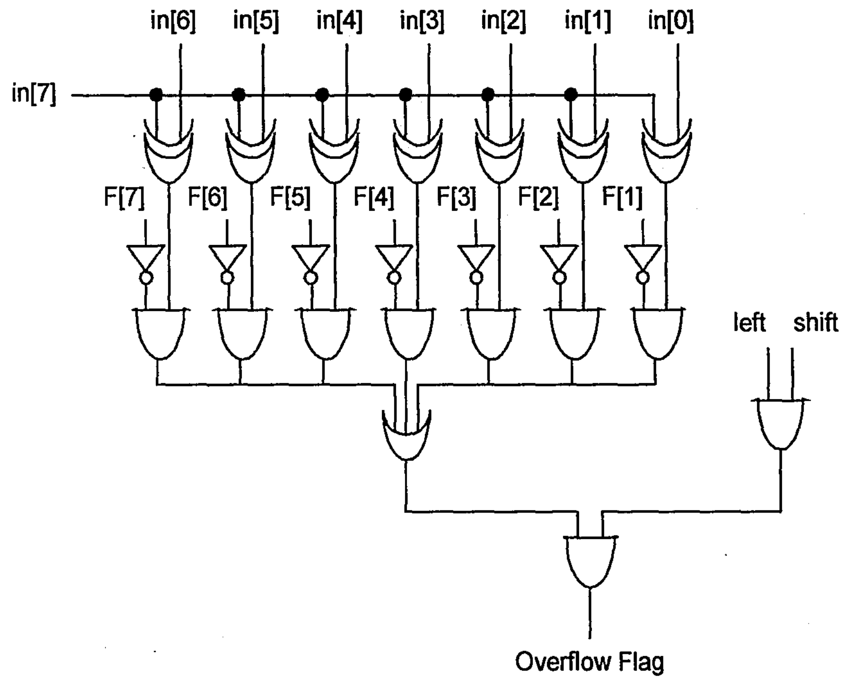

Figure 3.10: Current Overflow Flag Calculation

Version	Optimized	Report	8	128
Previous Method	Area	Area	45 gates	805 gates
		Delay	1.06 ns	1.76 ns
	Delay	Area	49 gates	855 gates
		Delay	1.28 ns	1.79 ns
Current Method	Area	Area	[illegible]	[illegible]
		Delay	[illegible]	[illegible]
	Delay	Area	48 gates	581 gates
		Delay	1.19 ns	1.97 ns

Table 3.6: Overflow Method Comparison

INTENTIONAL SECOND EXPOSURE

overflow. The logical OR network that follows then determines if an overflow has occurred. This result must then be logically AND'd with the left shift signal. All of this can be seen in Figure 3.10. Table 3.6 reveals synthesis results for this and the previous approach described in Section 2.4. As can be seen, the current method not only has a lower cost, but a reduced latency as well.

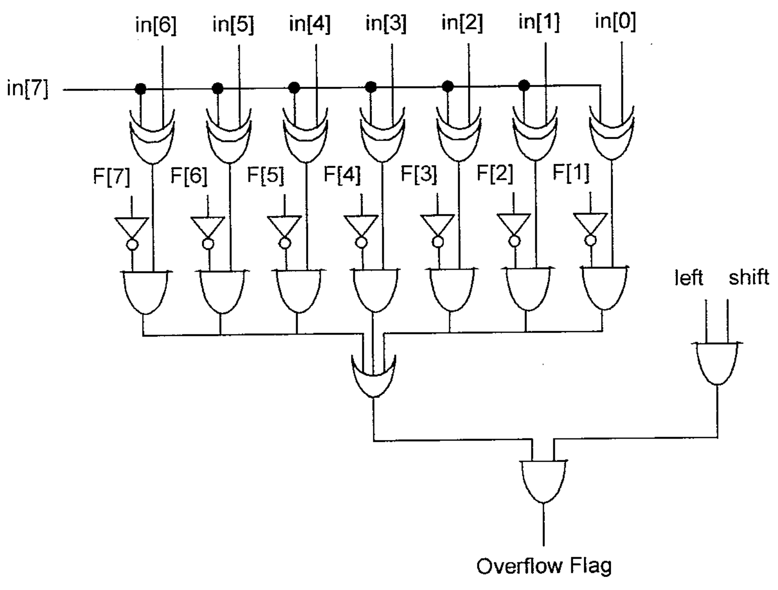

Figure 3.10: Current Overflow Flag Calculation

Version	Optimized	Report	8	128
Previous Method	Area	Area	**45 gates**	**805 gates**
		Delay	**1.06 ns**	**1.76 ns**
	Delay	Area	49 gates	855 gates
		Delay	1.28 ns	1.79 ns
Current Method	Area	Area	**28 gates**	**468 gates**
		Delay	**0.94 ns**	**1.64 ns**
	Delay	Area	48 gates	581 gates
		Delay	1.19 ns	1.97 ns

Table 3.6: Overflow Method Comparison

3.2.5 Examples

As was done with the Mux-based Data Reversal approach, examples are given to better illustrate the design at work.

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Rotate Right	10100110	2
Step	Value		
Mux data reversal	10100110		
Calculate F	00111111		
Calculate R = Rotate	10101001		
Calculate P = F + rotate	00111111+1=11111111		
Calculate $\overline{P}$	00000000		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\overline{P}$	10101001*11111111+0*00000000=101011001		
Mux left/rightMux data reversal	10101001		
Result	10101001		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	10100110*11111111=10100110->0		
Calculate data[(n-2)..0] xor data[n-1]	1011001		
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0		

Table 3.7: Mask-based Data Reversal Rotate Right Example

INTENTIONAL SECOND EXPOSURE

3.2.5 Examples

As was done with the Mux-based Data Reversal approach, examples are given to better illustrate the design at work.

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Rotate Right	10100110	2
Step	**Value**		
Mux data reversal	10100110		
Calculate F	00111111		
Calculate R = Rotate	10101001		
Calculate P = F + rotate	00111111+1=11111111		
Calculate P	00000000		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\bar{P}$	10101001*11111111+0*00000000=101011001		
Mux left/rightMux data reversal	10101001		
Result	10101001		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	10100110*11111111=10100110->0		
Calculate data[(n-2)..0] xor data[n-1]	1011001		
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0		

Table 3.7: Mask-based Data Reversal Rotate Right Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Rotate Left	10100110	2
Step	Value		
Mux data reversal	01100101		
Calculate F	00111111		
Calculate R = Rotate	01011001		
Calculate P = F + rotate	00111111+1=11111111		
Calculate $\overline{P}$	00000000		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\overline{P}$	01011001*11111111+0*00000000=01011001		
Mux data reversal	10011010		
Result	10011010		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	01100101*11111111=01100101->0		
Calculate data[(n-2)..0] xor data[n-1]	1011001		
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0		

Table 3.8: Mask-based Data Reversal Rotate Left Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Shift Right Logical	10100110	2
Step	Value		
Mux data reversal	10100110		
Calculate F	00111111		
Calculate R = Rotate	10101001		
Calculate P = F + rotate	00111111+0=00111111		
Calculate $\overline{P}$	11000000		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\overline{P}$	10101001*00111111+0*11000000=00101001		
Mux data reversal	00101001		
Result	00101001		
Calculate Z = P reversed	11111100		
Calculate Zero Flag	10100110*11111100=10100100->0		
Calculate data[(n-2)..0] xor data[n-1]	1011001		
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0		

Table 3.9: Mask-based Data Reversal Shift Right Logical Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Rotate Left	10100110	2

Step	Value
Mux data reversal	01100101
Calculate F	00111111
Calculate R = Rotate	01011001
Calculate P = F + rotate	00111111+1=11111111
Calculate P	00000000
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	01011001*11111111+0*00000000=01011001
Mux data reversal	10011010
Result	10011010
Calculate Z = P reversed	11111111
Calculate Zero Flag	01100101*11111111=01100101->0
Calculate data[(n-2)..0] xor data[n-1]	1011001
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0

Table 3.8: Mask-based Data Reversal Rotate Left Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Shift Right Logical	10100110	2

Step	Value
Mux data reversal	10100110
Calculate F	00111111
Calculate R = Rotate	10101001
Calculate P = F + rotate	00111111+0=00111111
Calculate P	11000000
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10101001*00111111+0*11000000=00101001
Mux data reversal	00101001
Result	00101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	10100110*11111100=10100100->0
Calculate data[(n-2)..0] xor data[n-1]	1011001
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0

Table 3.9: Mask-based Data Reversal Shift Right Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Shift Left Logical	10100110	2
Step	**Value**		
Mux data reversal	01100101		
Calculate F	00111111		
Calculate R = Rotate	01011001		
Calculate P = F + rotate	00111111+0=00111111		
Calculate $\overline{P}$	11000000		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\overline{P}$	01011001*00111111+0*11000000=00011001		
Mux data reversal	10011000		
Result	10011000		
Calculate Z = P reversed	11111100		
Calculate Zero Flag	11111100*01100101=01100100->0		
Calculate data[(n-2)..0] xor data[n-1]	1011001		
Calculate Overflow Flag	1011001*1100000=1000000->1*1=1		

Table 3.10: Mask-based Data Reversal Shift Left Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Shift Right Arithmetic	10100110	2
Step	**Value**		
Mux data reversal	10100110		
Calculate F	00111111		
Calculate R = Rotate	10101001		
Calculate P = F + rotate	00111111+0=00111111		
Calculate $\overline{P}$	11000000		
Calculate S = arithmetic*data[n-1]	1*1=1		
Calculate T = R*P+S* $\overline{P}$	10101001*00111111+1*11000000=11101001		
Mux data reversal	11101001		
Result	11101001		
Calculate Z = P reversed	11111100		
Calculate Zero Flag	11111100*10100110=10100100->0		
Calculate data[(n-2)..0] xor data[n-1]	1011001		
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0		

Table 3.11: Mask-based Data Reversal Shift Right Arithmetic Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Shift Left Logical	10100110	2

Step	Value
Mux data reversal	01100101
Calculate F	00111111
Calculate R = Rotate	01011001
Calculate P = F + rotate	00111111+0=00111111
Calculate $\bar{P}$	11000000
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	01011001*00111111+0*11000000=00011001
Mux data reversal	10011000
Result	10011000
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*01100101=01100100->0
Calculate data[(n-2)..0] xor data[n-1]	1011001
Calculate Overflow Flag	1011001*1100000=1000000->1*1=1

Table 3.10: Mask-based Data Reversal Shift Left Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Data Reversal	Shift Right Arithmetic	10100110	2

Step	Value
Mux data reversal	10100110
Calculate F	00111111
Calculate R = Rotate	10101001
Calculate P = F + rotate	00111111+0=00111111
Calculate $\bar{P}$	11000000
Calculate S = arithmetic*data[n-1]	1*1=1
Calculate T = R*P+S* $\bar{P}$	10101001*00111111+1*11000000=11101001
Mux data reversal	11101001
Result	11101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0
Calculate data[(n-2)..0] xor data[n-1]	1011001
Calculate Overflow Flag	1011001*1100000=1000000->1*0=0

Table 3.11: Mask-based Data Reversal Shift Right Arithmetic Example

3.3 Mask-based Two's Complement

3.3.1 Design Overview

The Mask-based Two's Complement design is very much like the previous design. A difference does exist, however, instead of manipulating the data of left oriented operations so that right oriented hardware operates on it correctly, the shift/rotate amount is manipulated. In this manner, the data is rotated to the right until it is as though it were rotated left. A mask is again used to manipulate the rotate result in order to obtain a result for a different operation.

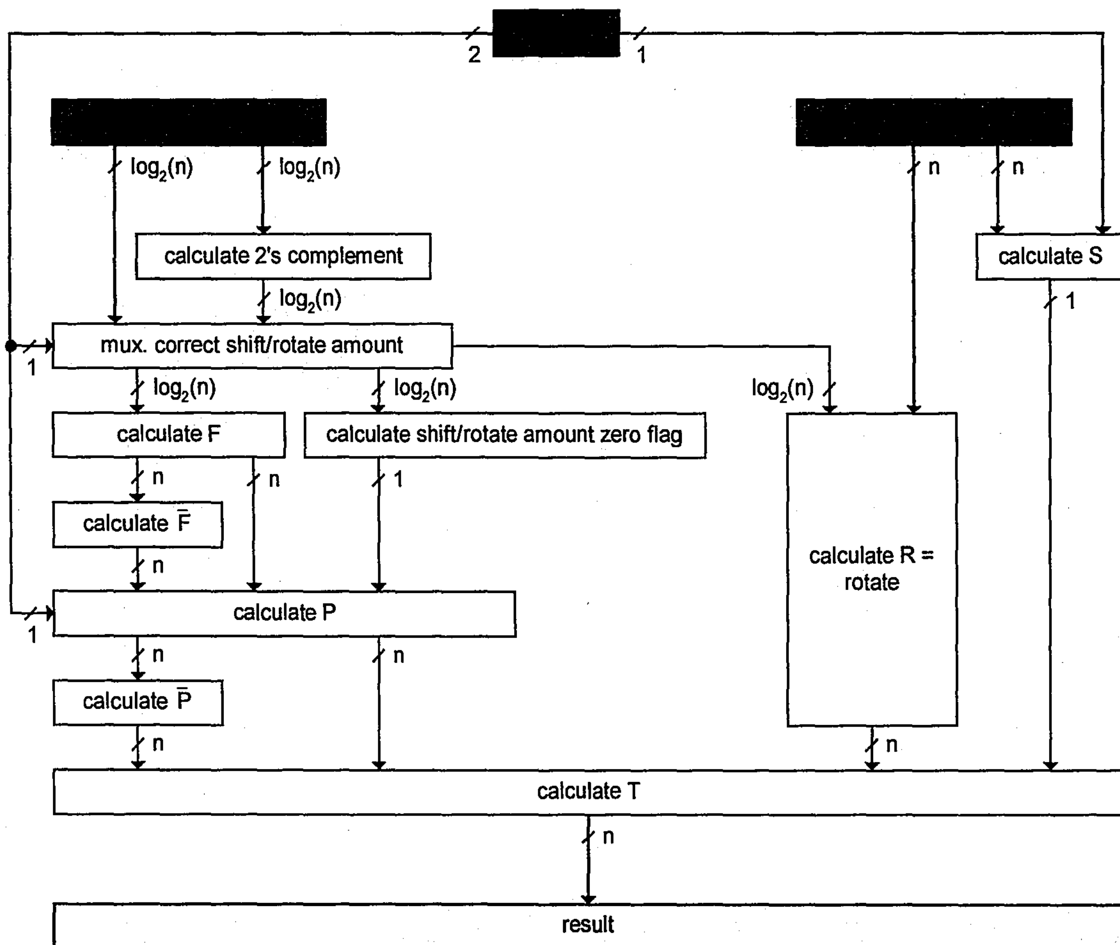

Figure 3.11: Mask-based Two's Complement

INTENTIONAL SECOND EXPOSURE

3.3 Mask-based Two's Complement

3.3.1 Design Overview

The Mask-based Two's Complement design is very much like the previous design. A difference does exist, however, instead of manipulating the data of left oriented operations so that right oriented hardware operates on it correctly, the shift/rotate amount is manipulated. In this manner, the data is rotated to the right until it is as though it were rotated left. A mask is again used to manipulate the rotate result in order to obtain a result for a different operation.

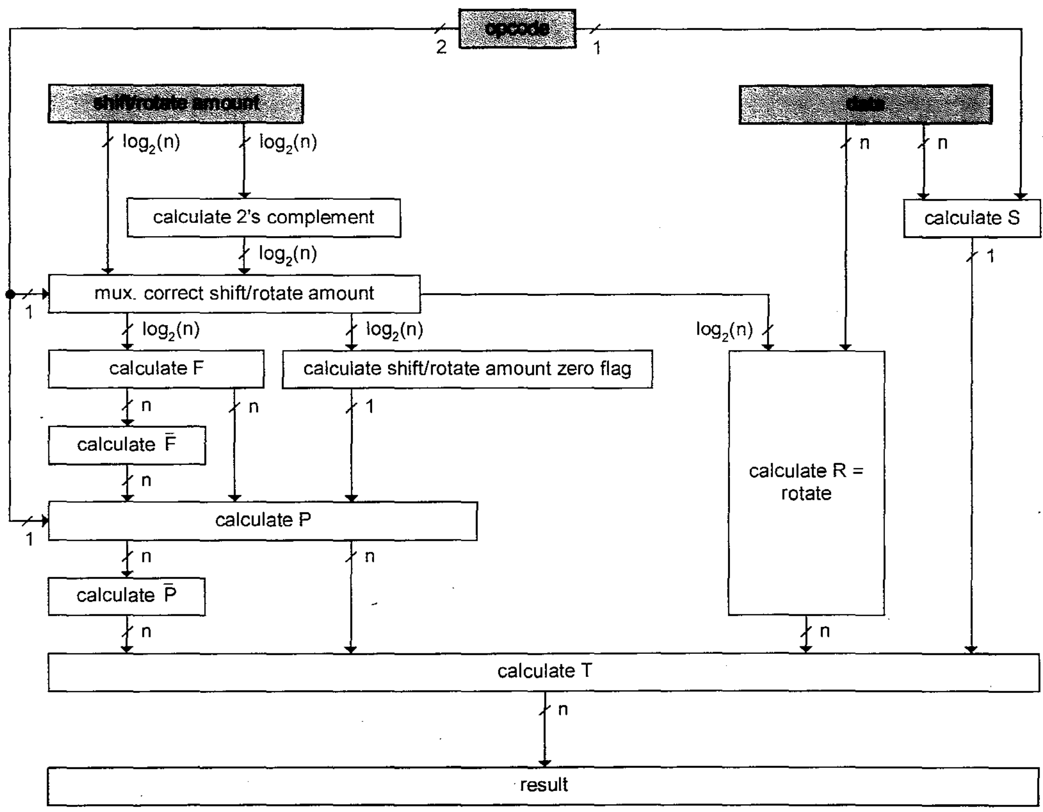

Figure 3.11: Mask-based Two's Complement

The design overview can be seen in Figure 3.11. The S calculation unit and rotator are the same as before, as is the mask F generator. That is, however, all that remains unchanged from the previous design. This design starts by calculating the two's complement of the shift/rotate amount. This value is equal to (*n*-shift/rotate amount), which is the value that left oriented operations must shift the data to the right. Once this value is known, the shift/rotate amount is selected that corresponds to the operation direction. This value is then used in the rotator and for mask generation. The mask F generator uses the same procedure as the previous method in calculating the mask.

While the mask F still applies directly to a shift right logical operation, due to the changes in the way the shift is performed, it is not directly applicable to the shift left logical operation as well. Instead, the inverse of F corresponds with that operation. Therefore, calculation of the modified mask P is much more complex and involved. In this computation, the correct orientation of F is selected for a given shift direction. In addition, two other factors are logically OR'd with this value. The first is the rotate signal which is used to alter the mask so it can be used in rotate operations as well, just as was done before. The second is a signal indicating whether or not the shift/rotate amount is zero. This is a one if the shift/rotate amount is zero and a zero otherwise. This series of operations is expressed in the equation $P = (right*F+left*\bar{F}) + (rotate + S/R\ amount\ zero)$.

This last value, the shift/rotate amount zero, is required to properly set the mask if the operation is a shift left logical operation with a shift/rotate amount of zero. This is due in part to the fact that the two's complement of zero is zero. Additionally, the mask was designed for applications actually involving a non-zero shift/rotate amount, which this does not. As such, this is a simple workaround which requires little additional hardware since it is only a logical OR tree with a trailing inverter for a bit vector of length $\log_2(n)$.

The final result is calculated as T. It is of the same form as the previous calculation. This was maintained at the expense of a more complicated mask P computation. As before, the mask P is bit-wise logically AND'd with the rotate result, R. At the same time, the value S is logically AND'd with the inverse of P. These two values are then bit-wise logically OR'd together, $T=R*P+S*\bar{P}$.

3.3.2 Two's Complement

The unit computing the two's complement is a carry ripple structure, as is evident in Figure 3.12. It operates by inverting the shift/rotate amount and then using half-adders to add a one to this value. Normally, such a unit would have such high latency that it would not even be considered. Since only the two's complement of the shift/rotate amount is needed, which is $\log_2(n)$-bits wide, the amount of the delay may be much less than one would initially assume. This unit could be replaced with one incorporating carry lookahead logic, but analysis has showed that there is no noticeable gain from doing so. This is due to the small width of the shift/rotate value. As such, the carry ripple unit is kept. This does nothing for the fact that the unit, in any form, is in the critical delay path and is slow.

The only positive aspect from using this device is that it negates the need for a data reversal unit that, while not terribly costly in terms of hardware required, is so in terms of the number and length of signal paths required in any implementation. Generally, this concern only matters when the manufacturing process is of a particularly fine type.

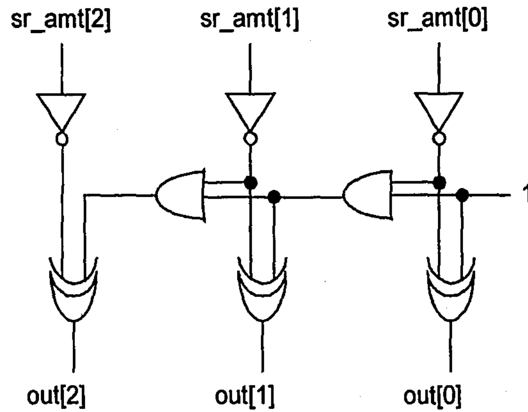

Figure 3.12: Two's Complement Unit

3.3.3 Zero Flag

This unit computes the zero flag in a manner very similar to that of the Mask-based Data Reversal design. Some modifications, however, were required to account for the altered mask construction and the fact that a mux data reversal unit is not used. Figure 3.13 shows how the zero flag computation is added to the design.

As seen, the input to the zero flag computation unit is the data directly from the input, not from a mux data reversal unit as was done before. The mask Z is computed as before by simply reversing the bit order of the mask P. The changes made to the P mask computation make it suitable for application to the zero flag computation. The remainder of the zero flag calculation remains as before.

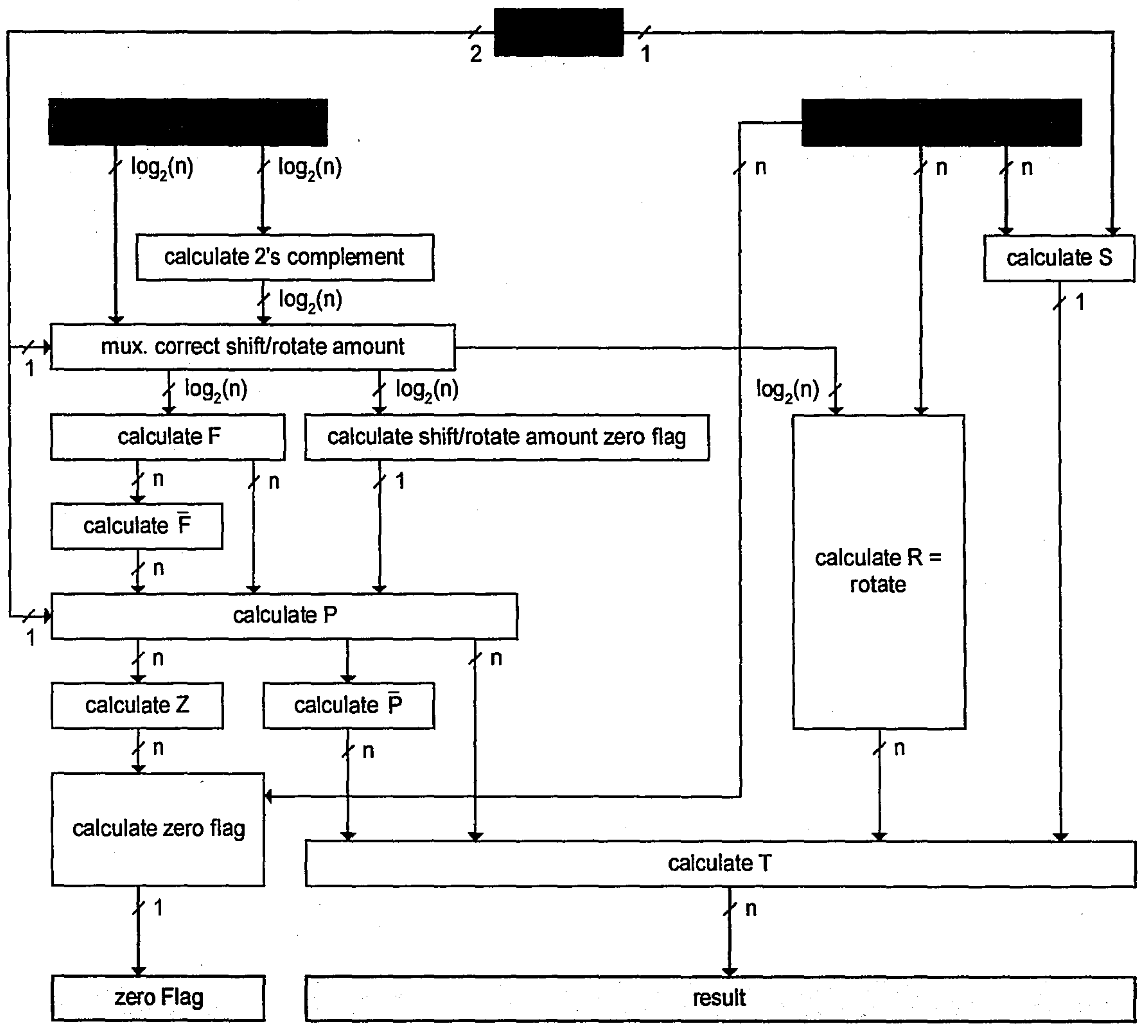

Figure 3.13: Mask-based Two's Complement with Zero Flag

3.3.4 Register Load Optimized

In some implementations, the data for a shift/rotate operation is in a register. As such, a certain amount of delay is induced into the execution of such a command for the retrieval of the register's data. Since the shift/rotate amount is often an immediate value in the instruction, it is usable as soon as the instruction is decoded. It is therefore apparent that there exists a period of time in the execution of the instruction where the shift/rotate amount is available, but the data to be shifted/rotated is not. This time is often wasted, as nothing is being done.

INTENTIONAL SECOND EXPOSURE

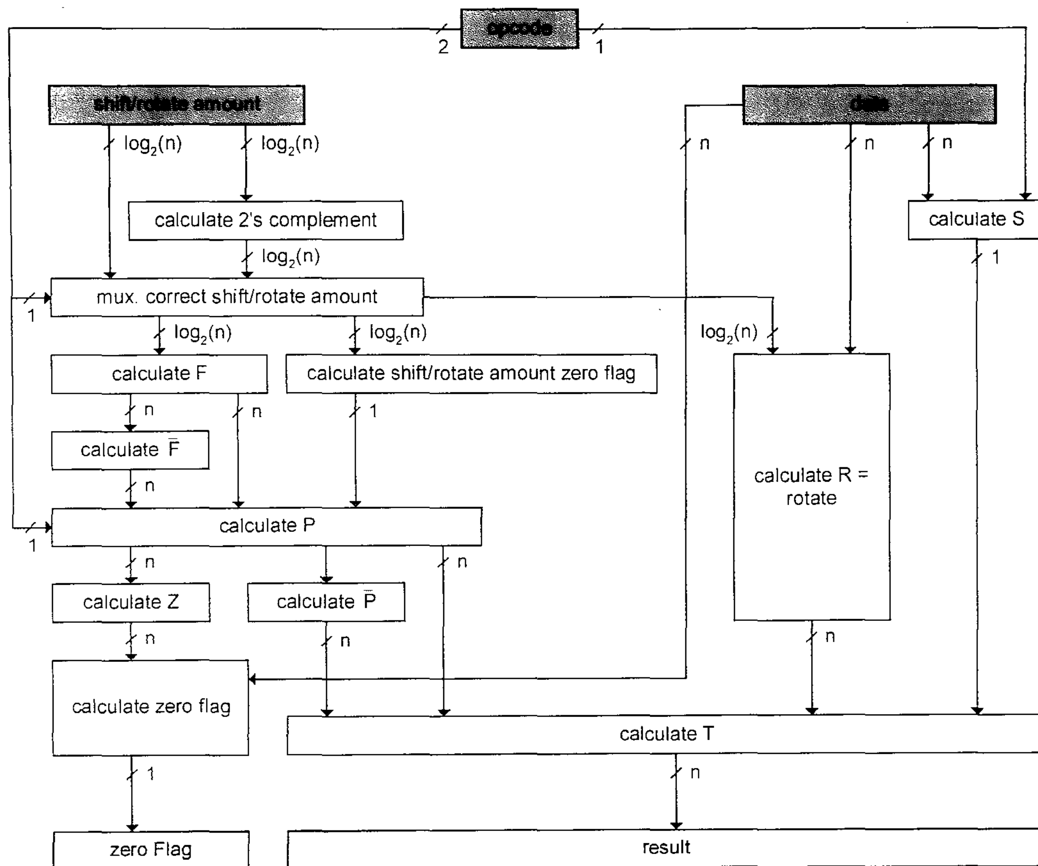

Figure 3.13: Mask-based Two's Complement with Zero Flag

3.3.4 Register Load Optimized

In some implementations, the data for a shift/rotate operation is in a register. As such, a certain amount of delay is induced into the execution of such a command for the retrieval of the register's data. Since the shift/rotate amount is often an immediate value in the instruction, it is usable as soon as the instruction is decoded. It is therefore apparent that there exists a period of time in the execution of the instruction where the shift/rotate amount is available, but the data to be shifted/rotated is not. This time is often wasted, as nothing is being done.

It is possible, however, to do some work during this period that will lower the critical path delay of the instruction. If it is assumed that any implementation will break down the design into components no longer in delay than the rotator, then there is that much time to accomplish work before the register's value is known. For the case of the Mask-based Two's Complement design, it is possible to calculate the two's complement value and perform the subsequent selection of the proper shift/rotate amount. This can be seen in Tables 3.12 and 3.13. Table 3.12 illustrates the cost of the rotator used and 3.13 illustrates the cost of the two's complement computation and the subsequent selection mechanism. It is clear that the two's complement calculation and the following selection can be done in an amount of time slightly less than that of the rotator.

As such, this calculation can be performed while the data is being retrieved. Since these two tasks are major components to the critical path delay of the design as a whole, their removal from the execution stage should reap a significant benefit.

Version	Optimized	Report	8	128
Rotator	Area	Area	[illegible]	[illegible]
		Delay	[illegible]	[illegible]
	Delay	Area	49 gates	855 gates
		Delay	1.28 ns	1.79 ns

Table 3.12: Rotator Cost

Version	Optimized	Report	8	128
2's Complement + shift/rotate mux	Area	Area	[illegible]	54 gates
		Delay	[illegible]	2.12 ns
	Delay	Area	20 gates	[illegible]
		Delay	0.85 ns	[illegible]

Table 3.13: Two's Complement and Shift/Rotate Mux Cost

Therefore, in those cases where the instruction is structured so that the data is retrieved from a register, and the shift/rotate amount is an immediate, it is beneficial to use this design. Some implementations, however, have instructions in which the shift/rotate amount is from a

INTENTIONAL SECOND EXPOSURE

It is possible, however, to do some work during this period that will lower the critical path delay of the instruction. If it is assumed that any implementation will break down the design into components no longer in delay than the rotator, then there is that much time to accomplish work before the register's value is known. For the case of the Mask-based Two's Complement design, it is possible to calculate the two's complement value and perform the subsequent selection of the proper shift/rotate amount. This can be seen in Tables 3.12 and 3.13. Table 3.12 illustrates the cost of the rotator used and 3.13 illustrates the cost of the two's complement computation and the subsequent selection mechanism. It is clear that the two's complement calculation and the following selection can be done in an amount of time slightly less than that of the rotator.

As such, this calculation can be performed while the data is being retrieved. Since these two tasks are major components to the critical path delay of the design as a whole, their removal from the execution stage should reap a significant benefit.

Version	Optimized	Report	8	128
Rotator	Area	Area	**45 gates**	**805 gates**
		Delay	**1.06 ns**	**1.76 ns**
	Delay	Area	49 gates	855 gates
		Delay	1.28 ns	1.79 ns

Table 3.12: Rotator Cost

Version	Optimized	Report	8	128
2's Complement + shift/rotate mux	Area	Area	**18 gates**	54 gates
		Delay	**0.87 ns**	2.12 ns
	Delay	Area	20 gates	**68 gates**
		Delay	0.85 ns	**1.58 ns**

Table 3.13: Two's Complement and Shift/Rotate Mux Cost

Therefore, in those cases where the instruction is structured so that the data is retrieved from a register, and the shift/rotate amount is an immediate, it is beneficial to use this design. Some implementations, however, have instructions in which the shift/rotate amount is from a

register. In this case, the proposed alterations have no benefit since both the shift/rotate amount and the data are available simultaneously.

3.3.5 Examples

As before, examples are given to better illustrate the process that this design implements. A second example of the shift left logical operation is given to highlight the special case of when the shift/rotate amount is zero.

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Rotate Right	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	010
Calculate F	00111111
Calculate $\bar{F}$	11000000
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+1+0=11111111
Calculate $\bar{P}$	00000000
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10101001*11111111+0*00000000=10101001
Result	10101001
Calculate Z = P reversed	11111111
Calculate Zero Flag	11111111*10100110=10100110->0

Table 3.14: Mask-based Two's Complement Rotate Right Example

INTENTIONAL SECOND EXPOSURE

register. In this case, the proposed alterations have no benefit since both the shift/rotate amount and the data are available simultaneously.

3.3.5 Examples

As before, examples are given to better illustrate the process that this design implements. A second example of the shift left logical operation is given to highlight the special case of when the shift/rotate amount is zero.

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Rotate Right	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	010
Calculate F	00111111
Calculate $\bar{F}$	11000000
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+1+0=11111111
Calculate $\bar{P}$	00000000
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10101001*11111111+0*00000000=10101001
Result	10101001
Calculate Z = P reversed	11111111
Calculate Zero Flag	11111111*10100110=10100110->0

Table 3.14: Mask-based Two's Complement Rotate Right Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Rotate Left	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	110
Calculate F	00000011
Calculate $\bar{F}$	11111100
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	0*00000011+1*11111100+1+0=11111111
Calculate $\bar{P}$	00000000
Calculate R = Rotate	10011010
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10011010*11111111+0*00000000=10011010
Result	10011010
Calculate Z = P reversed	11111111
Calculate Zero Flag	11111111*10100110=10100110->0

Table 3.15: Mask-based Two's Complement Rotate Left Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Right Logical	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	010
Calculate F	00111111
Calculate $\bar{F}$	11000000
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+0+0=00111111
Calculate $\bar{P}$	11000000
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10101001*00111111+0*11000000=00101001
Result	00101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0

Table 3.16: Mask-based Two's Complement Shift Right Logical Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Rotate Left	10100110	2
Step	**Value**		
Calculate 2's complement of shift/rotate amount	110		
Mux shift/rotate amount	110		
Calculate F	00000011		
Calculate F	11111100		
Calculate shift/rotate amount zero	0		
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	0*00000011+1*11111100+1+0=11111111		
Calculate P	00000000		
Calculate R = Rotate	10011010		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* P	10011010*11111111+0*00000000=10011010		
Result	10011010		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	11111111*10100110=10100110->0		

Table 3.15: Mask-based Two's Complement Rotate Left Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Right Logical	10100110	2
Step	**Value**		
Calculate 2's complement of shift/rotate amount	110		
Mux shift/rotate amount	010		
Calculate F	00111111		
Calculate $\bar{F}$	11000000		
Calculate shift/rotate amount zero	0		
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+0+0=00111111		
Calculate $\bar{P}$	11000000		
Calculate R = Rotate	10101001		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\bar{P}$	10101001*00111111+0*11000000=00101001		
Result	00101001		
Calculate Z = P reversed	11111100		
Calculate Zero Flag	11111100*10100110=10100100->0		

Table 3.16: Mask-based Two's Complement Shift Right Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Left Logical	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	110
Calculate F	00000011
Calculate $\overline{F}$	11111100
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	0*00000011+1*11111100+0+0=11111100
Calculate $\overline{P}$	00000011
Calculate R = Rotate	10011010
Calculate S = arithmetic*data[*n* -1]	0*1=0
Calculate T = R*P+S* $\overline{P}$	10011010*11111100+0*00000011=10011000
Result	10011000
Calculate Z = P reversed	00111111
Calculate Zero Flag	00111111*10100110=00100110->0

Table 3.17: Mask-based Two's Complement Shift Left Logical Example 1

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Right Arithmetic	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	010
Calculate F	00111111
Calculate $\overline{F}$	11000000
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+0+0=00111111
Calculate $\overline{P}$	11000000
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[*n* -1]	1*1=1
Calculate T = R*P+S* $\overline{P}$	10101001*00111111+1*11000000=11101001
Result	11101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0

Table 3.18: Mask-based Two's Complement Shift Right Arithmetic Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Left Logical	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	110
Calculate F	00000011
Calculate $\overline{F}$	11111100
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	0*00000011+1*11111100+0+0=11111100
Calculate $\overline{P}$	00000011
Calculate R = Rotate	10011010
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\overline{P}$	10011010*11111100+0*00000011=10011000
Result	10011000
Calculate Z = P reversed	00111111
Calculate Zero Flag	00111111*10100110=00100110->0

Table 3.17: Mask-based Two's Complement Shift Left Logical Example 1

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Right Arithmetic	10100110	2

Step	Value
Calculate 2's complement of shift/rotate amount	110
Mux shift/rotate amount	010
Calculate F	00111111
Calculate $\overline{F}$	11000000
Calculate shift/rotate amount zero	0
Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+0+0=00111111
Calculate $\overline{P}$	11000000
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	1*1=1
Calculate T = R*P+S* $\overline{P}$	10101001*00111111+1*11000000=11101001
Result	11101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0

Table 3.18: Mask-based Two's Complement Shift Right Arithmetic Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Left Logical	10100110	0
[illegible]			
Calculate 2's complement of shift/rotate amount	000		
Mux shift/rotate amount	000		
Calculate F	11111111		
Calculate $\overline{F}$	00000000		
Calculate shift/rotate amount zero	1		
Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+0+1=11111111		
Calculate $\overline{P}$	00000000		
Calculate R = Rotate	10100110		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\overline{P}$	10100110*11111111+0*00000000=10100110		
Result	10100110		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	11111111*10100110=10100110->0		

Table 3.19: Mask-based Two's Complement Shift Left Logical Example 2

3.4 Mask-based One's Complement

3.4.1 Design Overview

This version of the barrel shifter is a derivative of the Mask-based Two's Complement design. The motivation for this change is that the unit responsible for computing the two's complement of the shift/rotate amount is quite slow in its ability to perform that calculation. In particular, the unit uses a ripple carry structure. Analysis of a unit designed specifically with lookahead logic to decrease the delay of the computation, however, showed no noticeable reduction. As such, the ripple structure was kept and the delay problem persisted. Therefore, since there is no real way of optimizing the unit as it stands, the only course of action is to alter the computation performed and then correct any calculation resulting from this altered computation.

This is where the one's complement of the shift/rotate amount comes into effect. Instead of the two's complement being computed, the one's complement is computed. The one's complement is of course a simple inversion of the bits. The one's complement, however, is off

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based Two's Complement	Shift Left Logical	10100110	0

Step	Value
Calculate 2's complement of shift/rotate amount	000
Mux shift/rotate amount	000
Calculate F	11111111
Calculate $\bar{F}$	00000000
Calculate shift/rotate amount zero	1
Calculate P = (right*F+left* $\bar{F}$)+(rotate+S/R amount zero)	1*00111111+0*11000000+0+1=11111111
Calculate $\bar{P}$	00000000
Calculate R = Rotate	10100110
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10100110*11111111+0*00000000=10100110
Result	10100110
Calculate Z = P reversed	11111111
Calculate Zero Flag	11111111*10100110=10100110->0

Table 3.19: Mask-based Two's Complement Shift Left Logical Example 2

3.4 Mask-based One's Complement

3.4.1 Design Overview

This version of the barrel shifter is a derivative of the Mask-based Two's Complement design. The motivation for this change is that the unit responsible for computing the two's complement of the shift/rotate amount is quite slow in its ability to perform that calculation. In particular, the unit uses a ripple carry structure. Analysis of a unit designed specifically with lookahead logic to decrease the delay of the computation, however, showed no noticeable reduction. As such, the ripple structure was kept and the delay problem persisted. Therefore, since there is no real way of optimizing the unit as it stands, the only course of action is to alter the computation performed and then correct any calculation resulting from this altered computation.

This is where the one's complement of the shift/rotate amount comes into effect. Instead of the two's complement being computed, the one's complement is computed. The one's complement is of course a simple inversion of the bits. The one's complement, however, is off

by 1. Figure 3.14 shows the design using the one's complement computation. As can be seen, its introduction has necessitated the introduction of a few units. They are meant to correct for the inaccuracy that is inherent to the one's complement.

In addition, a unit has been added directly after the mask F calculation unit. This unit logically shifts the mask F right by one if the operation is left oriented. This sets the F mask with the proper number of leading zeros. If not adjusted, it would have one less than that which is required. The F mask generator with the additional logical shift can be seen in Figure 3.15.

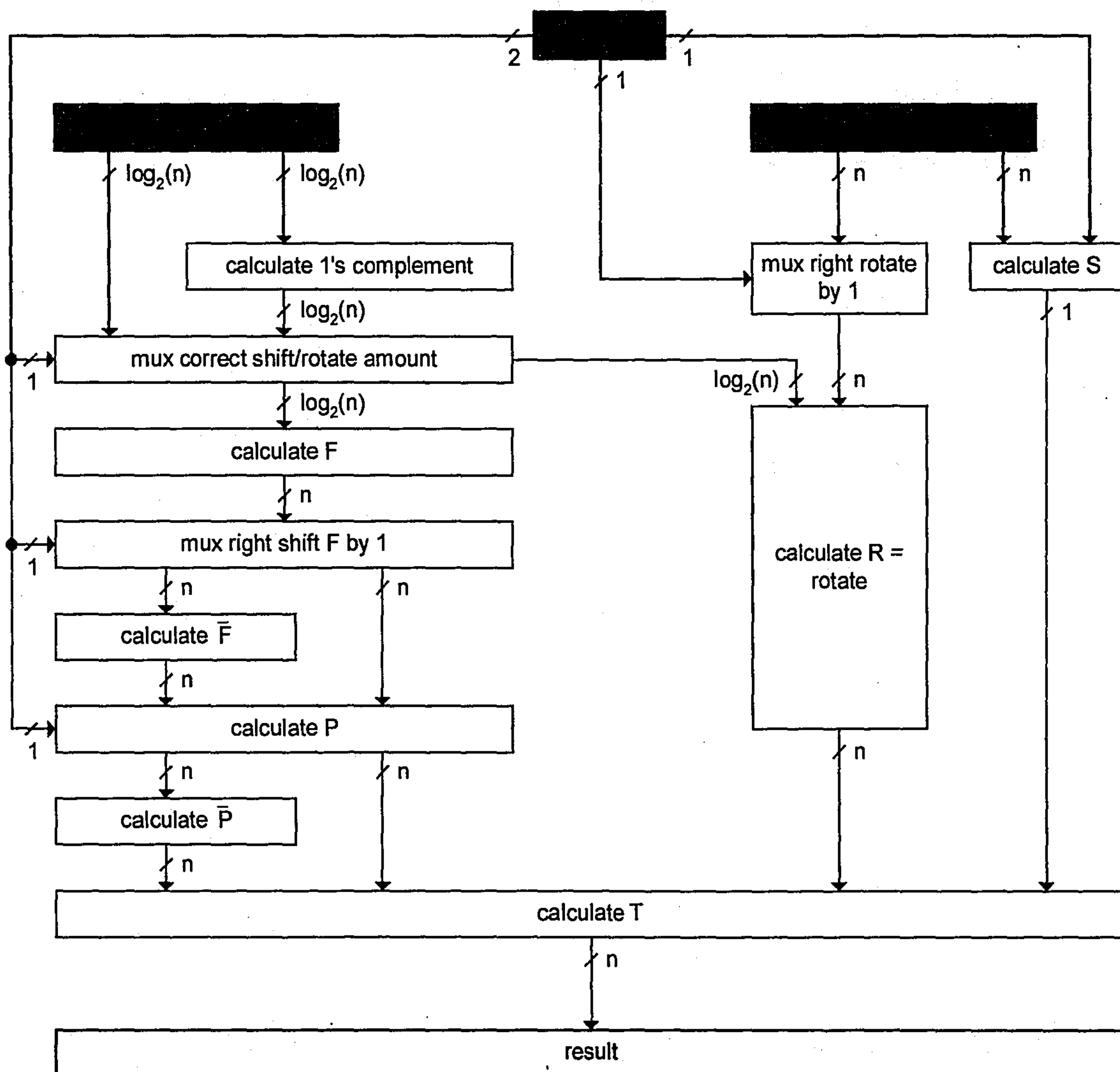

Figure 3.14: Mask-based One's Complement

INTENTIONAL SECOND EXPOSURE

by 1. Figure 3.14 shows the design using the one's complement computation. As can be seen, its introduction has necessitated the introduction of a few units. They are meant to correct for the inaccuracy that is inherent to the one's complement.

In addition, a unit has been added directly after the mask F calculation unit. This unit logically shifts the mask F right by one if the operation is left oriented. This sets the F mask with the proper number of leading zeros. If not adjusted, it would have one less than that which is required. The F mask generator with the additional logical shift can be seen in Figure 3.15.

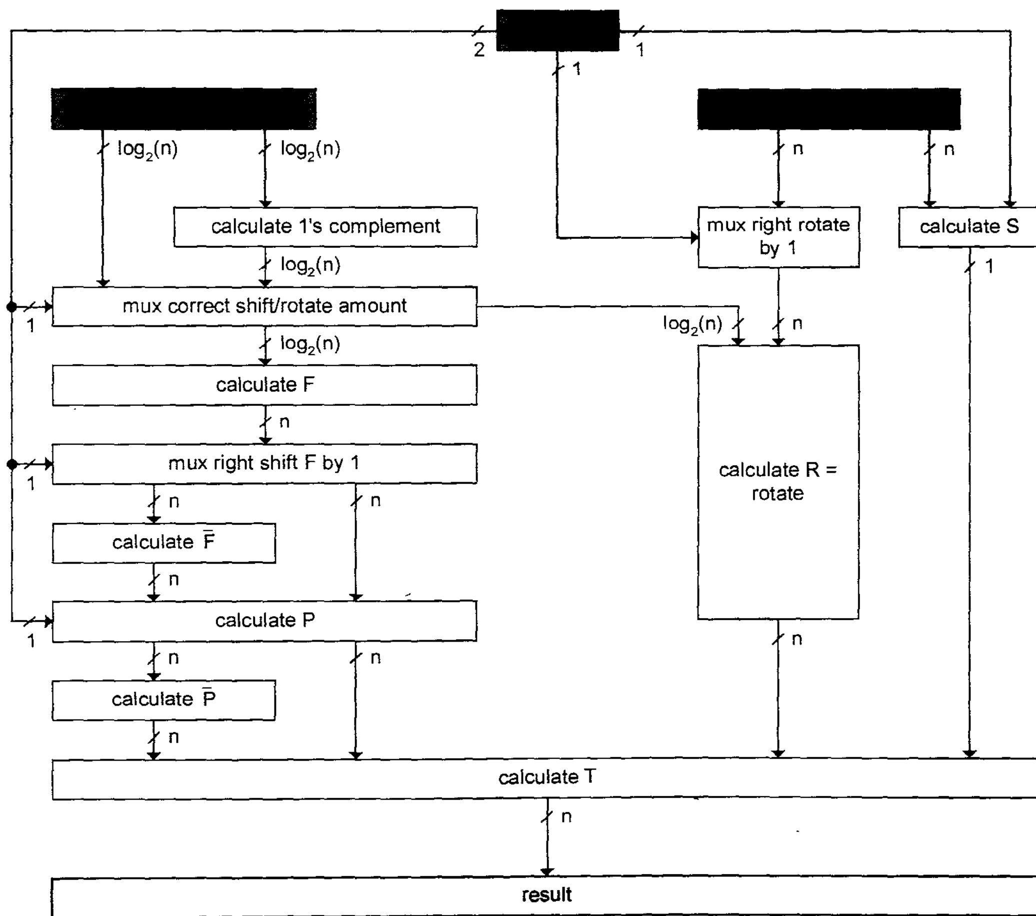

Figure 3.14: Mask-based One's Complement

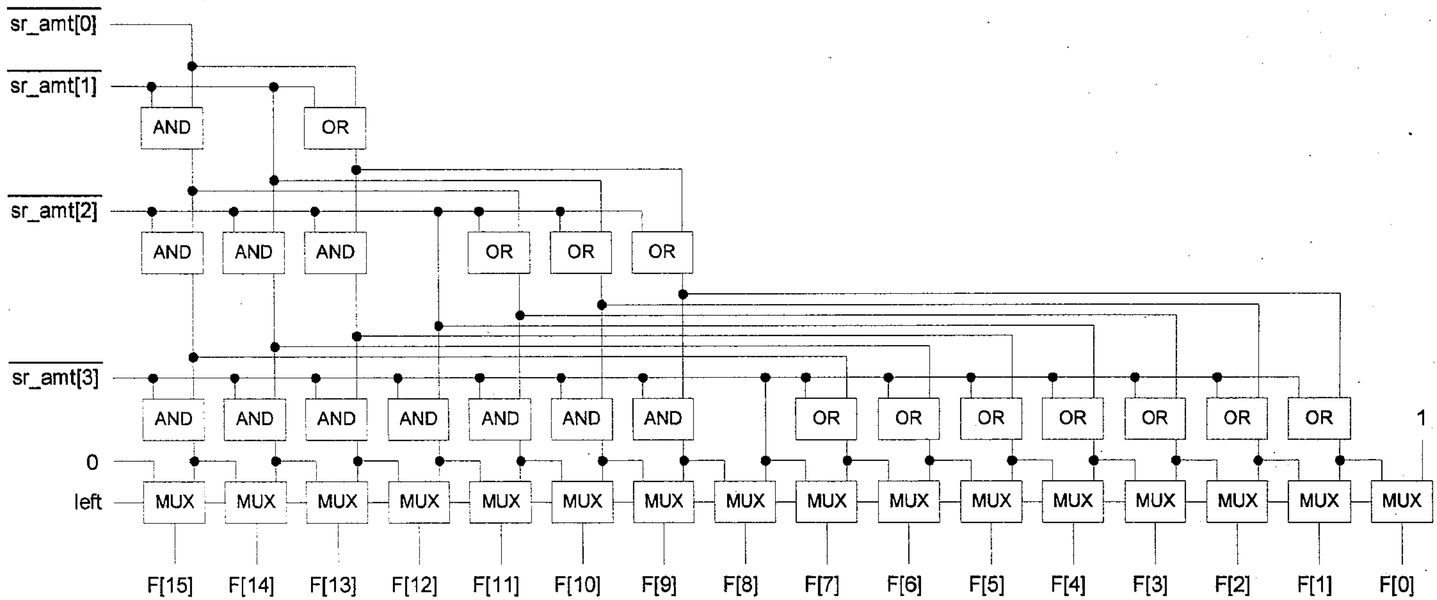

Figure 3.15: Mask F Generator for One's Complement

In addition to the change in the calculation of the mask F, there is a change in the computation of the rotator result as well. As with the mask F, an additional multiplexor row is needed. In this case, the multiplexors perform a rotate right by one if the operation is left oriented. These multiplexors have been placed at the beginning of the unit since the left signal is immediately available, while the selected shift/rotate amount is not. In this manner, it should be ready by the time the added multiplexor row is finished. This can be seen in Figure 2.6, which is similar to the rotator used here except it uses a multiplexor to select the proper shift/rotate amount, not logical XOR gates.

The last noticeable difference between this design and the two's complement version is that the shift/rotate amount zero flag is no longer required. The alteration to the mask F computation means that this value is no longer needed. As such, it is dropped from the design.

3.4.2 Zero Flag

Integration of the zero flag computation into this design does not differ in any way from the Mask-based Two's Complement design. This can be seen in Figure 3.16.

3.4.3 Examples

Now that the mechanisms behind the design are known, a few examples are shown to demonstrate the exact manner in which each operation is performed. Each example shows the outputs of the subunits from the design diagrams so that the manner in which the result is computed is easily seen.

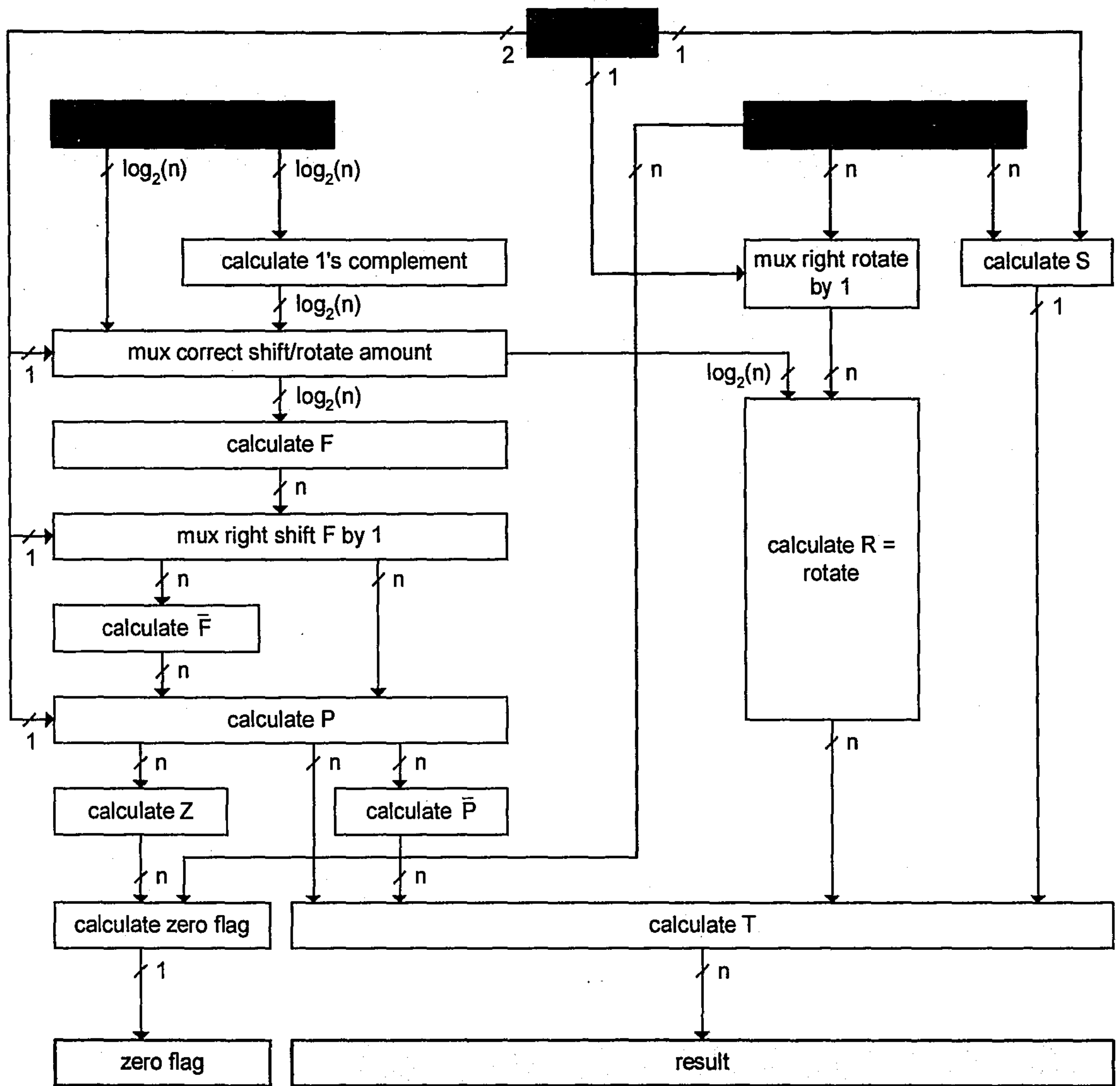

Figure 3.16: Mask-based One's Complement with Zero Flag

INTENTIONAL SECOND EXPOSURE

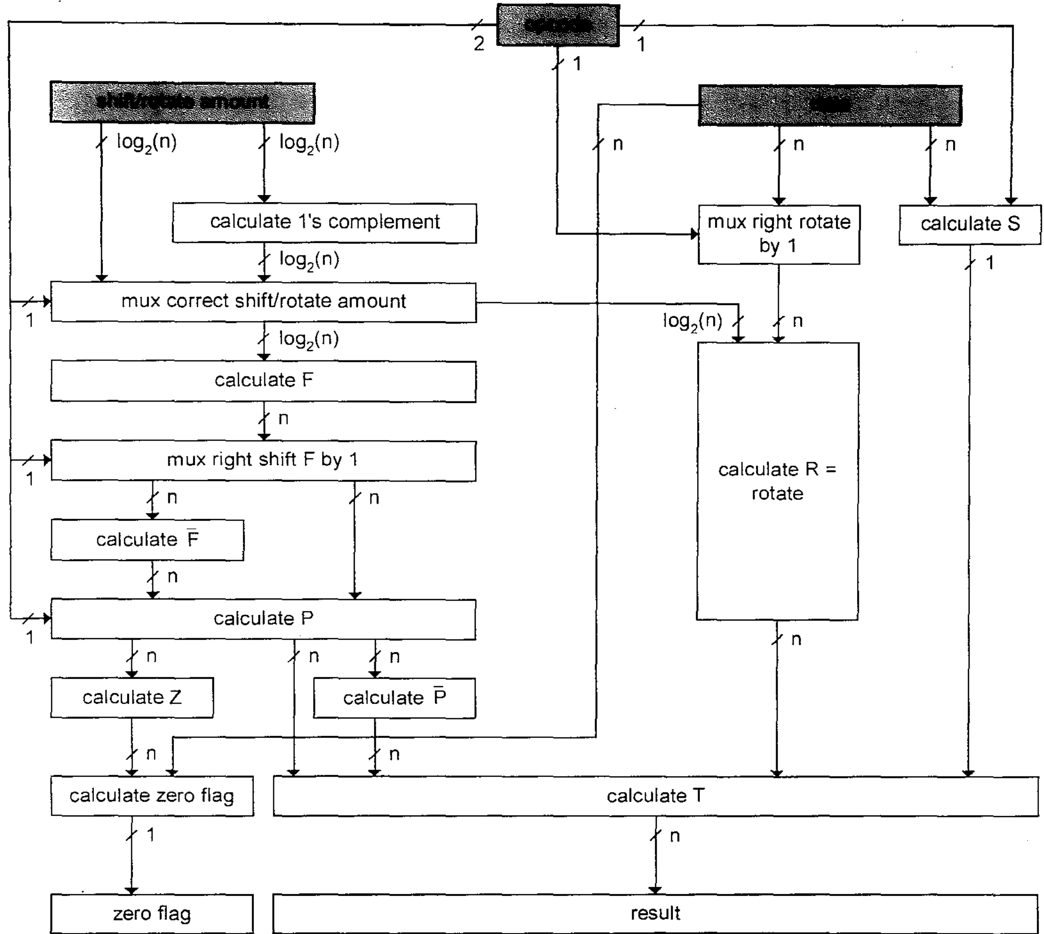

Figure 3.16: Mask-based One's Complement with Zero Flag

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Rotate Right	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	010
Calculate F	00111111
Mux right shift F by 1	00111111
Calculate $\bar{F}$	11000000
Calculate P = (right*F+left* $\bar{F}$)+rotate	1*00111111+0*11000000+1=11111111
Calculate $\bar{P}$	00000000
Mux right rotate by 1	10100110
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10101001*11111111+0*00000000=10101001
Result	10101001
Calculate Z = P reversed	11111111
Calculate Zero Flag	11111111*10100110=10100110->0

Table 3.20: Mask-based One's Complement Rotate Right Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Rotate Left	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	101
Calculate F	00000111
Mux right shift F by 1	00000011
Calculate $\bar{F}$	11111100
Calculate P = (right*F+left* $\bar{F}$)+rotate	0*00000011+1*11111100+1+11111111
Calculate $\bar{P}$	00000000
Mux right rotate by 1	01010011
Calculate R = Rotate	10011010
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\bar{P}$	10011010*11111111+0*00000000=10011010
Result	10011010
Calculate Z = P reversed	11111111
Calculate Zero Flag	11111111*10100110=10100110->0

Table 3.21: Mask-based One's Complement Rotate Left Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Rotate Right	10100110	2
Step	**Value**		
Calculate 1's complement of shift/rotate amount	101		
Mux shift/rotate amount	010		
Calculate F	00111111		
Mux right shift F by 1	00111111		
Calculate $\bar{F}$	11000000		
Calculate P = (right*F+left* F)+rotate	1*00111111+0*11000000+1=11111111		
Calculate $\bar{P}$	00000000		
Mux right rotate by 1	10100110		
Calculate R = Rotate	10101001		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* $\bar{P}$	10101001*11111111+0*00000000=10101001		
Result	10101001		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	11111111*10100110=10100110->0		

Table 3.20: Mask-based One's Complement Rotate Right Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Rotate Left	10100110	2
Step	**Value**		
Calculate 1's complement of shift/rotate amount	101		
Mux shift/rotate amount	101		
Calculate F	00000111		
Mux right shift F by 1	00000011		
Calculate F	11111100		
Calculate P = (right*F+left* $\bar{F}$)+rotate	0*00000011+1*11111100+1+11111111		
Calculate $\bar{P}$	00000000		
Mux right rotate by 1	01010011		
Calculate R = Rotate	10011010		
Calculate S = arithmetic*data[n-1]	0*1=0		
Calculate T = R*P+S* P	10011010*11111111+0*00000000=10011010		
Result	10011010		
Calculate Z = P reversed	11111111		
Calculate Zero Flag	11111111*10100110=10100110->0		

Table 3.21: Mask-based One's Complement Rotate Left Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Shift Right Logical	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	010
Calculate F	00111111
Mux right shift F by 1	00111111
Calculate $\overline{F}$	11000000
Calculate P = (right*F+left* $\overline{F}$)+rotate	1*00111111+0*11000000+0=00111111
Calculate $\overline{P}$	11000000
Mux right rotate by 1	10100110
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\overline{P}$	10101001*00111111+0*11000000=00101001
Result	00101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0

Table 3.22: Mask-based One's Complement Shift Right Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Shift Left Logical	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	101
Calculate F	00000111
Mux right shift F by 1	00000011
Calculate $\overline{F}$	11111100
Calculate P = (right*F+left* $\overline{F}$)+rotate	0*00000011+1*11111100+0=11111100
Calculate $\overline{P}$	00000011
Mux right rotate by 1	01010011
Calculate R = Rotate	10011010
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* $\overline{P}$	10011010*11111100+0*00000011=10011000
Result	10011000
Calculate Z = P reversed	00111111
Calculate Zero Flag	00111111*10100110=00100110->0

Table 3.23: Mask-based One's Complement Shift Left Logical Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Shift Right Logical	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	010
Calculate F	00111111
Mux right shift F by 1	00111111
Calculate F	11000000
Calculate P = (right*F+left* $\bar{F}$)+rotate	1*00111111+0*11000000+0=00111111
Calculate $\bar{P}$	11000000
Mux right rotate by 1	10100110
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* P	10101001*00111111+0*11000000=00101001
Result	00101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0

Table 3.22: Mask-based One's Complement Shift Right Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Shift Left Logical	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	101
Calculate F	00000111
Mux right shift F by 1	00000011
Calculate F	11111100
Calculate P = (right*F+left* $\bar{F}$)+rotate	0*00000011+1*11111100+0=11111100
Calculate $\bar{P}$	00000011
Mux right rotate by 1	01010011
Calculate R = Rotate	10011010
Calculate S = arithmetic*data[n-1]	0*1=0
Calculate T = R*P+S* P	10011010*11111100+0*00000011=10011000
Result	10011000
Calculate Z = P reversed	00111111
Calculate Zero Flag	00111111*10100110=00100110->0

Table 3.23: Mask-based One's Complement Shift Left Logical Example

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Shift Right Arithmetic	10100110	2
[illegible]	[illegible]		
Calculate 1's complement of shift/rotate amount	101		
Mux shift/rotate amount	010		
Calculate F	00111111		
Mux right shift F by 1	00111111		
Calculate $\overline{F}$	11000000		
Calculate P = (right*F+left* $\overline{F}$)+rotate	1*00111111+0*11000000+0=00111111		
Calculate $\overline{P}$	11000000		
Mux right rotate by 1	10100110		
Calculate R = Rotate	10101001		
Calculate S = arithmetic*data[n-1]	1*1=1		
Calculate T = R*P+S* $\overline{P}$	10101001*00111111+1*11000000=11101001		
Result	11101001		
Calculate Z = P reversed	11111100		
Calculate Zero Flag	11111100*10100110=10100100->0		

Table 3.24: Mask-based One's Complement Shift Right Arithmetic Example

INTENTIONAL SECOND EXPOSURE

Version	Operation	Data	Shift/Rotate Amount
Mask-based One's Complement	Shift Right Arithmetic	10100110	2

Step	Value
Calculate 1's complement of shift/rotate amount	101
Mux shift/rotate amount	010
Calculate F	00111111
Mux right shift F by 1	00111111
Calculate F	11000000
Calculate P = (right*F+left* $\overline{F}$)+rotate	1*00111111+0*11000000+0=00111111
Calculate $\overline{P}$	11000000
Mux right rotate by 1	10100110
Calculate R = Rotate	10101001
Calculate S = arithmetic*data[n-1]	1*1=1
Calculate T = R*P+S* P	10101001*00111111+1*11000000=11101001
Result	11101001
Calculate Z = P reversed	11111100
Calculate Zero Flag	11111100*10100110=10100100->0

Table 3.24: Mask-based One's Complement Shift Right Arithmetic Example

Chapter 4 – Results

4.1 Estimates for Component Count and Number of Components on Critical Path

Theoretical component count and number of components on the critical path for each design are estimated and given in Tables 4.1 through 4.4. The designs have been decomposed into the following basic components: AND, OR, NOT, Multiplexor, and XOR. In addition, each design is divided into subunits. Each block of the respective design's overview diagram constitutes a subunit in the table. In some cases the number of components on the critical path could not be computed since the relative weighting of components was unknown. In these cases, no estimate is given. The number of components on the critical path of every subunit, however, is given, and therefore, one has the information to judge what stages are the most costly.

4.2 Synthesis Results for Area and Delay

Synthesis was performed on structural-level VHDL models. These models were generated with Java programs, one per design, which took as an argument the data width. The data width had to be a power of 2.

Each VHDL model was synthesized at standard effort for both the lca300k ASIC, a 0.6μm gate array library, and the Xilinx Spartan FPGA. In addition, each design was optimized twice, once for area and once for delay. These results are presented in Tables 4.5 through 4.8 for data input widths of: 8, 16, 32, 64, and 128-bits. For each table, a blue highlight indicates which design and optimization has the smallest area-delay product for a particular data width. A purple highlight indicates for a particular design the optimization that provides the smallest area-delay product.

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2 to 1)	XOR
		Component Counts					
	Base	Mux data reversal	0	0	0	n	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate R = Shift/Rotate	0	0	0	$n\lg(n)+n-1$	0
		Mux data reversal	0	0	0	n	0
		Total	1	0	0	$3n+n\lg(n)-1$	0
	Zero Flag	Calculate Zero Flag	0	$n-1$	1	0	0
		Total	1	$n-1$	1	$3n+n\lg(n)-1$	0
	Overflow Flag	Calculate Overflow Flag	1	n	1	$n-1$	$n-1$
		Total	2	$2n-1$	2	$4n+n\lg(n)-2$	$n-1$
		Number of Components on Critical Path					
	Base	Mux data reversal	0	0	0	1	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate R = Shift/Rotate	0	0	0	$2\lg(n)$	0
		Mux data reversal	0	0	0	1	0
		Total	0	0	0	$2\lg(n)+2$	0
	Zero Flag	Calculate Zero Flag	0	$\lg(n)$	1	0	0
		Total	-	-	-	-	-
	Overflow Flag	Calculate Overflow Flag	-	-	-	-	-
		Total	-	-	-	-	-

Table 4.1: Mux-based Data Reversal Theoretical Gate Count and Delay Measure

INTENTIONAL SECOND EXPOSURE

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2 to 1)	XOR
Mux-based Data Reversal		Component Counts					
	Base	Mux data reversal	0	0	0	n	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate R = Shift/Rotate	0	0	0	$n\lg(n)+n-1$	0
		Mux data reversal	0	0	0	n	0
		Total	1	0	0	$3n+n\lg(n)-1$	0
	Zero Flag	Calculate Zero Flag	0	$n-1$	1	0	0
		Total	1	$n-1$	1	$3n+n\lg(n)-1$	0
	Overflow Flag	Calculate Overflow Flag	1	n	1	$n-1$	$n-1$
		Total	2	$2n-1$	2	$4n+n\lg(n)-2$	$n-1$
		Number of Components on Critical Path					
	Base	Mux data reversal	0	0	0	1	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate R = Shift/Rotate	0	0	0	$2\lg(n)$	0
		Mux data reversal	0	0	0	1	0
		Total	0	0	0	$2\lg(n)+2$	0
	Zero Flag	Calculate Zero Flag	0	$\lg(n)$	1	0	0
		Total	-	-	-	-	-
	Overflow Flag	Calculate Overflow Flag	-	-	-	-	-
		Total	-	-	-	-	-

Table 4.1: Mux-based Data Reversal Theoretical Gate Count and Delay Measure

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2-to-1)	XOR
				Component Counts			
	Base	Mux data reversal	0	0	0	n	0
		Calculate F	n-lg(n)-1	n-lg(n)-1	lg(n)	0	0
		Calculate R = Rotate	0	0	0	$n$lg(n)	0
		Calculate P = F + rotate	0	n	0	0	0
		Calculate $\bar{P}$	0	0	n	0	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\bar{P}$	2n	n	0	0	0
		Mux data reversal	0	0	0	n	0
		Total	3n-lg(n)	3n-lg(n)-1	n+lg(n)	2n+$n$lg(n)	0
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	n	n-1	1	0	0
		Total	4n-lg(n)	4n-lg(n)-2	n+lg(n)+1	2n+$n$lg(n)	0
	Overflow Flag	Calculate data[(n-2)..0] xor data[n-1]	0	0	0	0	n-1
		Calculate Overflow Flag	n-1	n-2	n-1	0	0
		Total	5n-lg(n)-1	5n-lg(n)-4	2n+lg(n)	2n+$n$lg(n)	n-1
				Number of Components on Critical Path			
	Base	Mux data reversal	0	0	0	1	0
		Calculate F	lg(n)-1	0	1	0	0
		Calculate R = Rotate	0	0	0	lg(n)	0
		Calculate P = F + rotate	0	1	0	0	0
		Calculate $\bar{P}$	0	0	1	0	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\bar{P}$	1	1	0	0	0
		Mux data reversal	0	0	0	1	0
		Total	-	-	-	-	-
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	1	lg(n)	1	0	0
		Total	-	-	-	-	-
	Overflow Flag	Calculate data[(n-2)..0] xor data[n-1]	0	0	0	0	1
		Calculate Overflow Flag	1	lg(n)	0	0	0
		Total	-	-	-	-	-

Table 4.2: Mask-based Data Reversal Theoretical Gate Count and Delay Measure

INTENTIONAL SECOND EXPOSURE

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2-to-1)	XOR
Mask-based Data Reversal			Component Counts				
	Base	Mux data reversal	0	0	0	n	0
		Calculate F	n-lg(n)-1	n-lg(n)-1	lg(n)	0	0
		Calculate R = Rotate	0	0	0	n lg(n)	0
		Calculate P = F + rotate	0	n	0	0	0
		Calculate $\bar{P}$	0	0	n	0	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\bar{P}$	2n	n	0	0	0
		Mux data reversal	0	0	0	n	0
		Total	3n-lg(n)	3n-lg(n)-1	n+lg(n)	2n+n lg(n)	0
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	n	n-1	1	0	0
		Total	4n-lg(n)	4n-lg(n)-2	n+lg(n)+1	2n+n lg(n)	0
	Overflow Flag	Calculate data[(n-2)..0] xor data[n-1]	0	0	0	0	n-1
		Calculate Overflow Flag	n-1	n-2	n-1	0	0
		Total	5n-lg(n)-1	5n-lg(n)-4	2n+lg(n)	2n+n lg(n)	n-1
			Number of Components on Critical Path				
	Base	Mux data reversal	0	0	0	1	0
		Calculate F	lg(n)-1	0	1	0	0
		Calculate R = Rotate	0	0	0	lg(n)	0
		Calculate P = F + rotate	0	1	0	0	0
		Calculate $\bar{P}$	0	0	1	0	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\bar{P}$	1	1	0	0	0
		Mux data reversal	0	0	0	1	0
		Total	-	-	-	-	-
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	1	lg(n)	1	0	0
		Total	-	-	-	-	-
	Overflow Flag	Calculate data[(n-2)..0] xor data[n-1]	0	0	0	0	1
		Calculate Overflow Flag	1	lg(n)	0	0	0
		Total	-	-	-	-	-

Table 4.2: Mask-based Data Reversal Theoretical Gate Count and Delay Measure

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2-to-1)	XOR
		Component Counts					
	Base	Calculate 2's complement of shift/rotate amount	$\lg(n)$-1	0	$\lg(n)$	0	$\lg(n)$
		Mux shift/rotate amount	0	0	0	$\lg(n)$	0
		Calculate F	n-$\lg(n)$-1	n-$\lg(n)$-1	$\lg(n)$	0	0
		Calculate $\overline{F}$	0	0	n	0	0
		Calculate shift/rotate amount zero	0	$\lg(n)$-1	1	0	0
		Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	0	n+1	0	n	0
		Calculate $\overline{P}$	0	0	n	0	0
		Calculate R = Rotate	0	0	0	$n\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	$2n$	n	0	0	0
		Total	$3n$-1	$3n$-1	$2n+2\lg(n)+1$	$n+(n+1)\lg(n)$	$\lg(n)$
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	n	n-1	1	0	0
		Total	$4n$-1	$4n$-2	$2n+2\lg(n)+2$	$n+(n+1)\lg(n)$	$\lg(n)$
		Number of Components on Critical Path					
	Base	Calculate 2's complement of shift/rotate amount	$\lg(n)$-1	0	1	0	1
		Mux shift/rotate amount	0	0	0	1	0
		Calculate F	$\lg(n)$-1	0	1	0	0
		Calculate $\overline{F}$	0	0	1	0	0
		Calculate shift/rotate amount zero	0	$\lceil \lg(\lg(n)) \rceil$	1	0	0
		Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	0	1	0	1	0
		Calculate $\overline{P}$	0	0	1	0	0
		Calculate R = Rotate	0	0	0	$\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	1	1	0	0	0
		Total	-	-	-	-	-
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	1	$\lg(n)$	1	0	0
		Total	-	-	-	-	-

Table 4.3: Mask-based Two's Complement Theoretical Gate Count and Delay Measure

INTENTIONAL SECOND EXPOSURE

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2-to-1)	XOR
Mask-based 2's Complement			Component Counts				
	Base	Calculate 2's complement of shift/rotate amount	$\lg(n)$-1	0	$\lg(n)$	0	$\lg(n)$
		Mux shift/rotate amount	0	0	0	$\lg(n)$	0
		Calculate F	n-$\lg(n)$-1	n-$\lg(n)$-1	$\lg(n)$	0	0
		Calculate $\overline{F}$	0	0	n	0	0
		Calculate shift/rotate amount zero	0	$\lg(n)$-1	1	0	0
		Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	0	n+1	0	n	0
		Calculate $\overline{P}$	0	0	n	0	0
		Calculate R = Rotate	0	0	0	$n\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	2n	n	0	0	0
		Total	3n-1	3n-1	2n+2$\lg(n)$+1	$n+(n+1)\lg(n)$	$\lg(n)$
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	n	n-1	1	0	0
		Total	4n-1	4n-2	2n+2$\lg(n)$+2	$n+(n+1)\lg(n)$	$\lg(n)$
			Number of Components on Critical Path				
	Base	Calculate 2's complement of shift/rotate amount	$\lg(n)$-1	0	1	0	1
		Mux shift/rotate amount	0	0	0	1	0
		Calculate F	$\lg(n)$-1	0	1	0	0
		Calculate $\overline{F}$	0	0	1	0	0
		Calculate shift/rotate amount zero	0	$\lceil\lg(\lg(n))\rceil$	1	0	0
		Calculate P = (right*F+left* $\overline{F}$)+(rotate+S/R amount zero)	0	1	0	1	0
		Calculate $\overline{P}$	0	0	1	0	0
		Calculate R = Rotate	0	0	0	$\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	1	1	0	0	0
		Total	-	-	-	-	-
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	1	$\lg(n)$	1	0	0
		Total	-	-	-	-	-

Table 4.3: Mask-based Two's Complement Theoretical Gate Count and Delay Measure

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2-to-1)	XOR
				Component Counts			
	Base	Calculate 1's complement of shift/rotate amount	0	0	$\lg(n)$	0	0
		Mux shift/rotate amount	0	0	0	$\lg(n)$	0
		Calculate F	$n-\lg(n)-1$	$n-\lg(n)-1$	$\lg(n)$	0	0
		Mux right shift F by 1	0	0	0	n	0
		Calculate $\overline{F}$	0	0	n	0	0
		Calculate P = (right*F+left* $\overline{F}$)+rotate	0	n	0	n	0
		Calculate $\overline{P}$	0	0	n	0	0
		Mux right rotate by 1	0	0	0	n	0
		Calculate R = Rotate	0	0	0	$n\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	$2n$	n	0	0	0
		Total	$3n-\lg(n)$	$3n-\lg(n)-1$	$2n+2\lg(n)$	$3n+(n+1)\lg(n)$	0
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	n	$n-1$	1	0	0
		Total	$4n-\lg(n)$	$4n-\lg(n)-2$	$2n+2\lg(n)+1$	$3n+(n+1)\lg(n)$	0
				Number of Components on Critical Path			
	Base	Calculate 1's complement of shift/rotate amount	0	0	1	0	0
		Mux shift/rotate amount	0	0	0	1	0
		Calculate F	lg(n)-1	0	1	0	0
		Mux right shift F by 1	0	0	0	1	0
		Calculate $\overline{F}$	0	0	1	0	0
		Calculate P = (right*F+left* $\overline{F}$)+rotate	0	1	0	1	0
		Calculate $\overline{P}$	0	0	1	0	0
		Mux right rotate by 1	0	0	0	1	0
		Calculate R = Rotate	0	0	0	$\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	1	1	0	0	0
		Total	-	-	-	-	-
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	1	$\lg(n)$	1	0	0
		Total	-	-	-	-	-

Table 4.4: Mask-based One's Complement Theoretical Gate Count and Delay Measure

INTENTIONAL SECOND EXPOSURE

Version	Type	Stage	AND(2-input)	OR(2-input)	NOT	MUX(2-to-1)	XOR
Mask-based 1's Complement			Component Counts				
	Base	Calculate 1's complement of shift/rotate amount	0	0	$\lg(n)$	0	0
		Mux shift/rotate amount	0	0	0	$\lg(n)$	0
		Calculate F	$n-\lg(n)-1$	$n-\lg(n)-1$	$\lg(n)$	0	0
		Mux right shift F by 1	0	0	0	n	0
		Calculate $\overline{F}$	0	0	n	0	0
		Calculate P = (right*F+left* $\overline{F}$)+rotate	0	n	0	n	0
		Calculate $\overline{P}$	0	0	n	0	0
		Mux right rotate by 1	0	0	0	n	0
		Calculate R = Rotate	0	0	0	$n\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	$2n$	n	0	0	0
		Total	$3n-\lg(n)$	$3n-\lg(n)-1$	$2n+2\lg(n)$	$3n+(n+1)\lg(n)$	0
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	n	$n-1$	1	0	0
		Total	$4n-\lg(n)$	$4n-\lg(n)-2$	$2n+2\lg(n)+1$	$3n+(n+1)\lg(n)$	0
			Number of Components on Critical Path				
	Base	Calculate 1's complement of shift/rotate amount	0	0	1	0	0
		Mux shift/rotate amount	0	0	0	1	0
		Calculate F	lg(n)-1	0	1	0	0
		Mux right shift F by 1	0	0	0	1	0
		Calculate $\overline{F}$	0	0	1	0	0
		Calculate P = (right*F+left* $\overline{F}$)+rotate	0	1	0	1	0
		Calculate $\overline{P}$	0	0	1	0	0
		Mux right rotate by 1	0	0	0	1	0
		Calculate R = Rotate	0	0	0	$\lg(n)$	0
		Calculate S = arithmetic*data[n-1]	1	0	0	0	0
		Calculate T = R*P+S* $\overline{P}$	1	1	0	0	0
		Total	-	-	-	-	-
	Zero Flag	Calculate Z = P reversed	0	0	0	0	0
		Calculate Zero Flag	1	$\lg(n)$	1	0	0
		Total	-	-	-	-	-

Table 4.4: Mask-based One's Complement Theoretical Gate Count and Delay Measure

4.2.1 Designs Without Flags

Analysis of the synthesis results for the designs without any flags, shown in Table 4.5, reveals that with the exception of the 8-bit case, the Mask-based Data Reversal design has a smaller delay than the other three designs. The Mux-based Data Reversal design, however, has the smallest area in all cases. Therefore, in four cases, it is difficult to say which design is preferred. In order to resolve this ambiguity, the area-delay product is analyzed. It shows that the optimal design depends on the bit-width of the input data. In particular, the Mux-based Data Reversal design is preferred in those cases where the data width is either 8 or 16-bits. The Mask-based Data Reversal design is preferred for all other data widths. As such, the preferred design is dependent on the input data width.

4.2.2 Designs with Zero Flag

When considering those designs only equipped with the zero flag, the Mask-based Data Reversal design has the smallest delay for all data widths with the exception of the 8-bit case, as shown in Table 4.6. Again, the Mux-based Data Reversal design has the smallest area for all data widths. Analysis of the area-delay product reveals that the Mux-based Data Reversal design is preferred for data widths less than or equal to 32-bits. Likewise, the Mask-based Data Reversal design is preferred for data widths greater than 32-bits. The Mask-based Two's Complement and Mask-based One's Complement designs have large delays. This factor dominates the area-delay product making them extremely uncompetitive.

4.2.3 Designs with Zero and Overflow Flags

Comparison of those designs with both flags reveals that, with the exception of the 8-bit case, the Mask-based Data Reversal design has a smaller delay than does the Mux-based Data

Reversal design. Again, the Mux-based Data Reversal design has a smaller area than does the Mask-based Data Reversal design for all data widths. The fact that the Mux-based Data Reversal design has a smaller area-delay product in four out of five instances indicates that in general the increase in the number of gates in the Mask-based Data Reversal design does not produce a proportional savings in delay so as to produce a design with an equal area-delay product. Therefore, the Mux-based Data Reversal design is preferred for all data widths except 64-bits. This is shown in Table 4.7.

4.2.4 Register Load Optimized Mask-based Two's Complement

The Register Load Optimized Mask-based Two's Complement design has a delay less than all other designs without flags. This, of course, assumes that every other design does no work until all operands are available. If that condition is not held, then it may be that one of the other designs fares better. Table 4.8 shows the synthesis results for this design.

Version	Synthesize	Optimize	Report	8	16	32	64	128
Mux-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	190 gates	446 gates	1022 gates	2302 gates	5118 gates
			Delay	2.49 ns	3.16 ns	4.15 ns	5.58 ns	8.24 ns
		Delay	Area			[illegible]		
			Delay			[illegible]		
	Xilinx Spartan FPGA	Area	Area	48 gates; 48 CLBs; 0 H	112 gates; 112 CLBs; 0 H	256 gates; 256 CLBs; 0 H	576 gates; 576 CLBs; 0 H	1280 gates; 1280 CLBs; 0 H
			Delay	22.75 ns	25.35 ns	27.95 ns	30.55 ns	33.15 ns
		Delay	Area	48 gates; 48 CLBs; 0 H	112 gates; 112 CLBs; 0 H	256 gates; 256 CLBs; 0 H	576 gates; 576 CLBs; 0 H	1280 gates; 1280 CLBs; 0 H
			Delay	22.75 ns	25.35 ns	27.95 ns	30.55 ns	33.15 ns
Mask-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	[illegible] gates	[illegible] gates			
			Delay	2.52 ns	2.96 ns			
		Delay	Area	251 gates	576 gates	1283 gates	2855 gates	6087 gates
			Delay	2.56 ns	2.92 ns	3.45 ns	3.96 ns	3.88 ns
	Xilinx Spartan FPGA	Area	Area	48 gates; 47 CLBs; 6 H	114 gates; 111 CLBs; 14 H	262 gates; 256 CLBs; 30 H	612 gates; 577 CLBs; 44 H	1361 gates; 1279 CLBs; 87 H
			Delay	25.54 ns	28.14 ns	30.74 ns	36.44 ns	38.85 ns
		Delay	Area	48 gates; 47 CLBs; 6 H	122 gates; 114 CLBs; 6 H	290 gates; 273 CLBs; 18 H	688 gates; 614 CLBs; 69 H	1382 gates; 1346 CLBs; 109 H
			Delay	25.54 ns	28.14 ns	30.74 ns	42.24 ns	39.04 ns
Mask-based 2s Complement	LSI Logic lca300k ASIC	Area	Area	209 gates	541 gates	[illegible] gates	[illegible]	[illegible]
			Delay	3.42 ns	5.03 ns	6.13 ns	[illegible]	[illegible]
		Delay	Area	263 gates	658 gates	1574 gates	3234 gates	5737 gates
			Delay	4.17 ns	5.83 ns	8.70 ns	15.98 ns	25.29 ns
	Xilinx Spartan FPGA	Area	Area	59 gates; 48 CLBs; 0 H	133 gates; 109 CLBs; 1 H	298 gates; 248 CLBs; 1 H	643 gates; 560 CLBs; 20 H	1383 gates; 1257 CLBs; 61 H
			Delay	31.95 ns	36.24 ns	40.55 ns	47.54 ns	53.44 ns
		Delay	Area	66 gates; 66 CLBs; 16 H	164 gates; 137 CLBs; 32 H	394 gates; 337 CLBs; 65 H	937 gates; 783 CLBs; 162 H	1445 gates; 1451 CLBs; 233 H
			Delay	32.14 ns	33.64 ns	37.53 ns	38.44 ns	45.94 ns
Mask-based 1s Complement	LSI Logic lca300k ASIC	Area	Area	272 gates	602 gates	[illegible] gates	[illegible] gates	[illegible]
			Delay	2.70 ns	3.46 ns	5.08 ns	6.72 ns	[illegible]
		Delay	Area	279 gates	676 gates	1476 gates	3203 gates	6936 gates
			Delay	3.02 ns	4.85 ns	7.08 ns	11.86 ns	19.20 ns
	Xilinx Spartan FPGA	Area	Area	66 gates; 59 CLBs; 0 H	147 gates; 132 CLBs; 0 H	324 gates; 295 CLBs; 4 H	700 gates; 652 CLBs; 16 H	1528 gates; 1423 CLBS; 26 H
			Delay	32.55 ns	32.55 ns	38.45 ns	41.54 ns	46.34 ns
		Delay	Area	66 gates; 59 CLBs; 0 H	147 gates; 132 CLBs; 0 H	324 gates; 315 CLBs; 16 H	708 gates; 681 CLBs; 37 H	1553 gates; 1522 CLBs; 74 H
			Delay	32.55 ns	32.55 ns	35.34 ns	38.64 ns	40.94 ns

Table 4.5: Designs without Flags Area and Delay Synthesis Results

INTENTIONAL SECOND EXPOSURE

Version	Synthesize	Optimize	Report	8	16	32	64	128
Mux-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	190 gates	446 gates	1022 gates	2302 gates	5118 gates
			Delay	2.49 ns	3.16 ns	4.15 ns	5.58 ns	8.24 ns
		Delay	Area	**190 gates**	**446 gates**	**1022 gates**	**2303 gates**	**5119 gates**
			Delay	**2.27 ns**	**2.99 ns**	**4.11 ns**	**4.49 ns**	**6.25 ns**
	Xilinx Spartan FPGA	Area	Area	48 gates; 48 CLBs; 0 H	112 gates; 112 CLBs; 0 H	256 gates; 256 CLBs; 0 H	576 gates; 576 CLBs; 0 H	1280 gates; 1280 CLBs; 0 H
			Delay	22.75 ns	25.35 ns	27.95 ns	30.55 ns	33.15 ns
		Delay	Area	48 gates; 48 CLBs; 0 H	112 gates; 112 CLBs; 0 H	256 gates; 256 CLBs; 0 H	576 gates; 576 CLBs; 0 H	1280 gates; 1280 CLBs; 0 H
			Delay	22.75 ns	25.35 ns	27.95 ns	30.55 ns	33.15 ns
Mask-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	**207 gates**	**482 gates**	**1097 gates**	**2487 gates**	**5480 gates**
			Delay	**2.52 ns**	**2.96 ns**	**3.40 ns**	**3.84 ns**	**4.29 ns**
		Delay	Area	251 gates	576 gates	1283 gates	2855 gates	6087 gates
			Delay	2.56 ns	2.92 ns	3.45 ns	3.96 ns	3.88 ns
	Xilinx Spartan FPGA	Area	Area	48 gates; 47 CLBs; 6 H	114 gates; 111 CLBs; 14 H	262 gates; 256 CLBs; 30 H	612 gates; 577 CLBs; 44 H	1361 gates; 1279 CLBs; 87 H
			Delay	25.54 ns	28.14 ns	30.74 ns	36.44 ns	38.85 ns
		Delay	Area	48 gates; 47 CLBs; 6 H	122 gates; 114 CLBs; 6 H	290 gates; 273 CLBs; 18 H	688 gates; 614 CLBs; 69 H	1382 gates; 1346 CLBs; 109 H
			Delay	25.54 ns	28.14 ns	30.74 ns	42.24 ns	39.04 ns
Mask-based 2s Complement	LSI Logic lca300k ASIC	Area	Area	**209 gates**	**541 gates**	**1195 gates**	**2606 gates**	**5677 gates**
			Delay	**3.42 ns**	**5.03 ns**	**6.13 ns**	**8.42 ns**	**11.58 ns**
		Delay	Area	263 gates	658 gates	1574 gates	3234 gates	5737 gates
			Delay	4.17 ns	5.83 ns	8.70 ns	15.98 ns	25.29 ns
	Xilinx Spartan FPGA	Area	Area	59 gates; 48 CLBs; 0 H	133 gates; 109 CLBs; 1 H	298 gates; 248 CLBs; 1 H	643 gates; 560 CLBs; 20 H	1383 gates; 1257 CLBs; 61 H
			Delay	31.95 ns	36.24 ns	40.55 ns	47.54 ns	53.44 ns
		Delay	Area	66 gates; 66 CLBs; 16 H	164 gates; 137 CLBs; 32 H	394 gates; 337 CLBs; 65 H	937 gates; 783 CLBs; 162 H	1445 gates; 1451 CLBs; 233 H
			Delay	32.14 ns	33.64 ns	37.53 ns	38.44 ns	45.94 ns
Mask-based 1s Complement	LSI Logic lca300k ASIC	Area	Area	**272 gates**	**602 gates**	**1439 gates**	**3109 gates**	**6698 gates**
			Delay	**2.70 ns**	**3.46 ns**	**5.08 ns**	**6.72 ns**	**9.75 ns**
		Delay	Area	279 gates	676 gates	1476 gates	3203 gates	6936 gates
			Delay	3.02 ns	4.85 ns	7.08 ns	11.86 ns	19.20 ns
	Xilinx Spartan FPGA	Area	Area	66 gates; 59 CLBs; 0 H	147 gates; 132 CLBs; 0 H	324 gates; 295 CLBs; 4 H	700 gates; 652 CLBs; 16 H	1528 gates; 1423 CLBS; 26 H
			Delay	32.55 ns	32.55 ns	38.45 ns	41.54 ns	46.34 ns
		Delay	Area	66 gates; 59 CLBs; 0 H	147 gates; 132 CLBs; 0 H	324 gates; 315 CLBs; 16 H	708 gates; 681 CLBs; 37 H	1553 gates; 1522 CLBs; 74 H
			Delay	32.55 ns	32.55 ns	35.34 ns	38.64 ns	40.94 ns

Table 4.5: Designs without Flags Area and Delay Synthesis Results

Version	Synthesize	Optimize	Report	8	16	32	64	128
Mux-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	196 gates	457 gates		2344 gates	5204 gates
			Delay	2.82 ns	3.64 ns		6.21 ns	9.01 ns
		Delay	Area			1049 gates		
			Delay			4.66 ns		
	Xilinx Spartan FPGA	Area	Area	50 gates; 49 CLBs; 1 H	116 gates; 114 CLBs; 2 H	265 gates; 261 CLBs; 3 H	593 gates; 585 CLBs; 8 H	1314 gates; 1297 CLBs; 17 H
			Delay	27.94 ns	33.34 ns	38.54 ns	41.14 ns	45.23 ns
		Delay	Area	50 gates; 49 CLBs; 1 H	117 gates; 115 CLBs; 0 H	265 gates; 261 CLBs; 4 H	594 gates; 586 CLBs; 9 H	1317 gates; 1299 CLBs; 16 H
			Delay	27.94 ns	31.65 ns	35.74 ns	39.84 ns	43.54 ns
Mask-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	[illegible]	[illegible]	[illegible]		
			Delay	[illegible]	[illegible]	[illegible]		
		Delay	Area	308 gates	678 gates	1450 gates	3200 gates	6960 gates
			Delay	2.89 ns	3.35 ns	3.86 ns	4.26 ns	4.65 ns
	Xilinx Spartan FPGA	Area	Area	60 gates; 51 CLBs; 6 H	123 gates; 117 CLBs; 18 H	303 gates; 271 CLBs; 27 H	634 gates; 600CLBs; 71 H	1382 gates; 1331 CLBs; 156 H
			Delay	30.83 ns	34.33 ns	38.04 ns	46.43 ns	47.72 ns
		Delay	Area	64 gates; 57 CLBs; 6 H	141 gates; 126 CLBs; 17 H	335 gates; 288 CLBs; 49 H	691 gates; 655 CLBs; 82 H	1542 gates; 1363 CLBs; 154 H
			Delay	25.54 ns	30.74 ns	38.54 ns	45.93 ns	37.24 ns
Mask-based 2s Complement	LSI Logic lca300k ASIC	Area	Area	[illegible]	[illegible]	[illegible]	[illegible]	
			Delay	[illegible]	[illegible]	[illegible]	[illegible]	
		Delay	Area	265 gates	755 gates	1559 gates	3222 gates	7578 gates
			Delay	4.44 ns	6.10 ns	8.67 ns	15.67 ns	21.51 ns
	Xilinx Spartan FPGA	Area	Area	66 gates; 52 CLBs; 5 H	150 gates; 118 CLBs; 10 H	334 gates; 266 CLBs; 18 H	688 gates; 605 CLBs; 83 H	1564 gates; 1300 CLBs; 37 H
			Delay	49.33 ns	47.23 ns	50.73 ns	52.03 ns	53.53 ns
		Delay	Area	86 gates; 71 CLBs; 15 H	157 gates; 139 CLBs; 20 H	371 gates; 329 CLBs; 79 H	804 gates; 690 CLBs; 181 H	1741 gates; 1531 CLBs; 181 H
			Delay	42.54 ns	48.33 ns	52.03 ns	58.53 ns	61.13 ns
Mask-based 1s Complement	LSI Logic lca300k ASIC	Area	Area	[illegible]	[illegible]	[illegible]	[illegible]	
			Delay	[illegible]	[illegible]	[illegible]	[illegible]	
		Delay	Area	321 gates	758 gates	1665 gates	3784 gates	7769 gates
			Delay	3.64 ns	5.30 ns	7.96 ns	13.40 ns	22.64 ns
	Xilinx Spartan FPGA	Area	Area	74 gates; 63 CLBs; 2 H	167 gates; 143 CLBs; 7 H	353 gates; 310 CLBs; 15 H	784 gates; 691 CLBs; 51 H	1688 gates; 1503 CLBs; 101 H
			Delay	39.94 gates	51.64 ns	52.43 ns	68.12 ns	79.10 ns
		Delay	Area	82 gates; 69 CLBs; 6 H	177 gates; 155 CLBs; 13 H	364 gates; 331 CLBs; 32 H	804 gates; 739 CLBs; 69 H	1766 gates; 1615 CLBs; 139 H
			Delay	43.44 ns	52.03 ns	48.72 ns	57.82 ns	67.61 ns

Table 4.6: Designs with Zero Flag Area and Delay Synthesis Results

INTENTIONAL SECOND EXPOSURE

Version	Synthesize	Optimize	Report	8	16	32	64	128
Mux-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	196 gates	457 gates	**1043 gates**	2344 gates	5204 gates
			Delay	2.82 ns	3.64 ns	**4.62 ns**	6.21 ns	9.01 ns
		Delay	Area	**196 gates**	**460 gates**	1049 gates	**2421 gates**	**5229 gates**
			Delay	**2.64 ns**	**3.53 ns**	4.66 ns	**5.31 ns**	**6.90 ns**
	Xilinx Spartan FPGA	Area	Area	50 gates; 49 CLBs; 1 H	116 gates; 114 CLBs; 2 H	265 gates; 261 CLBs; 3 H	593 gates; 585 CLBs; 8 H	1314 gates; 1297 CLBs; 17 H
			Delay	27.94 ns	33.34 ns	38.54 ns	41.14 ns	45.23 ns
		Delay	Area	50 gates; 49 CLBs; 1 H	117 gates; 115 CLBs; 0 H	265 gates; 261 CLBs; 4 H	594 gates; 586 CLBs; 9 H	1317 gates; 1299 CLBs; 16 H
			Delay	27.94 ns	31.65 ns	35.74 ns	39.84 ns	43.54 ns
Mask-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	**227 gates**	**512 gates**	**1262 gates**	**2741 gates**	**6022 gates**
			Delay	**2.85 ns**	**3.29 ns**	**4.10 ns**	**4.54 ns**	**4.98 ns**
		Delay	Area	308 gates	678 gates	1450 gates	3200 gates	6960 gates
			Delay	2.89 ns	3.35 ns	3.86 ns	4.26 ns	4.65 ns
	Xilinx Spartan FPGA	Area	Area	60 gates; 51 CLBs; 6 H	123 gates; 117 CLBs; 18 H	303 gates; 271 CLBs; 27 H	634 gates; 600CLBs; 71 H	1382 gates; 1331 CLBs; 156 H
			Delay	30.83 ns	34.33 ns	38.04 ns	46.43 ns	47.72 ns
		Delay	Area	64 gates; 57 CLBs; 6 H	141 gates; 126 CLBs; 17 H	335 gates; 288 CLBs; 49 H	691 gates; 655 CLBs; 82 H	1542 gates; 1363 CLBs; 154 H
			Delay	25.54 ns	30.74 ns	38.54 ns	45.93 ns	37.24 ns
Mask-based 2s Complement	LSI Logic lca300k ASIC	Area	Area	**233 gates**	**645 gates**	**1296 gates**	**2839 gates**	**6093 gates**
			Delay	**4.10 ns**	**5.84 ns**	**6.75 ns**	**10.66 ns**	**14.17 ns**
		Delay	Area	265 gates	755 gates	1559 gates	3222 gates	7578 gates
			Delay	4.44 ns	6.10 ns	8.67 ns	15.67 ns	21.51 ns
	Xilinx Spartan FPGA	Area	Area	66 gates; 52 CLBs; 5 H	150 gates; 118 CLBs; 10 H	334 gates; 266 CLBs; 18 H	688 gates; 605 CLBs; 83 H	1564 gates; 1300 CLBs; 37 H
			Delay	49.33 ns	47.23 ns	50.73 ns	52.03 ns	53.53 ns
		Delay	Area	86 gates; 71 CLBs; 15 H	157 gates; 139 CLBs; 20 H	371 gates; 329 CLBs; 79 H	804 gates; 690 CLBs; 181 H	1741 gates; 1531 CLBs; 181 H
			Delay	42.54 ns	48.33 ns	52.03 ns	58.53 ns	61.13 ns
Mask-based 1s Complement	LSI Logic lca300k ASIC	Area	Area	**289 gates**	**641 gates**	**1400 gates**	**3339 gates**	**7111 gates**
			Delay	**3.15 ns**	**4.49 ns**	**6.09 ns**	**7.94 ns**	**12.87 ns**
		Delay	Area	321 gates	758 gates	1665 gates	3784 gates	7769 gates
			Delay	3.64 ns	5.30 ns	7.96 ns	13.40 ns	22.64 ns
	Xilinx Spartan FPGA	Area	Area	74 gates; 63 CLBs; 2 H	167 gates; 143 CLBs; 7 H	353 gates; 310 CLBs; 15 H	784 gates; 691 CLBs; 51 H	1688 gates; 1503 CLBs; 101 H
			Delay	39.94 gates	51.64 ns	52.43 ns	68.12 ns	79.10 ns
		Delay	Area	82 gates; 69 CLBs; 6 H	177 gates; 155 CLBs; 13 H	364 gates; 331 CLBs; 32 H	804 gates; 739 CLBs; 69 H	1766 gates; 1615 CLBs; 139 H
			Delay	43.44 ns	52.03 ns	48.72 ns	57.82 ns	67.61 ns

Table 4.6: Designs with Zero Flag Area and Delay Synthesis Results

Version	Synthesize	Optimize	Report	8	16	32	64	128
[illegible]	LSI Logic lca300k ASIC	Area	Area	239 gates		1215 gates	2686 gates	5888 gates
			Delay	2.88 ns		4.98 ns	7.74 ns	13.12 ns
		Delay	Area		565 gates			
			Delay		3.53 ns			
	Xilinx Spartan FPGA	Area	Area	60 gates; 58 CLBs; 3 H	137 gates; 132 CLBs; 5 H	313 gates; 300 CLBs; 12 H	691 gates; 665 CLBs; 23 H	1510 gates; 1459 CLBs; 50 H
			Delay	28.34 ns	35.03 ns	42.23 ns	42.03 ns	46.13 ns
		Delay	Area	61 gates; 58 CLBs; 2 H	140 gates; 135 CLBs; 4 H	315 gates; 302 CLBs; 13 H	695 gates; 668 CLBs; 27 H	1519 gates; 1463 CLBs; 49 H
			Delay	27.94 ns	31.65 ns	36.64 ns	39.84 ns	44.44 ns
[illegible]	LSI Logic lca300k ASIC	Area	Area	259 gates	627 gates	1632 gates	3523 gates	[illegible]
			Delay	2.85 ns	3.39 ns	4.10 ns	4.54 ns	[illegible]
		Delay	Area	365 gates	815 gates	1619 gates		8745 gates
			Delay	2.95 ns	3.41 ns	3.86 ns		5.92 ns
	Xilinx Spartan FPGA	Area	Area	65 gates; 55 CLBs; 9 H	156 gates; 128 CLBs; 23 H	325 gates; 287 CLBs; 48 H	696 gates; 637 CLBs; 111 H	1738 gates; 1469 CLBs; 187 H
			Delay	34.34 ns	43.12 ns	43.54 ns	51.92 ns	67.79 ns
		Delay	Area	68 gates; 58 CLBs; 15 H	168 gates; 137 CLBs; 27 H	362 gates; 321 CLBs; 60 H	859 gates; 702 CLBs; 134 H	1965 gates; 1667 CLBs; 349 H
			Delay	29.94 ns	33.43 ns	37.03 ns	41.82 ns	52.12 ns

Table 4.7: Designs with Zero and Overflow Flags Area and Delay Synthesis Results

Version	Synthesize	Optimize	Report	8	16	32	64	128
	LSI Logic lca300k ASIC	Area	Area	189 gates	509 gates	1144 gates	2550 gates	[illegible]
			Delay	2.21 ns	3.11 ns	3.14 ns	3.59 ns	[illegible]
		Delay	Area	223 gates	629 gates	1531 gates	3180 gates	5669 gates
			Delay	2.34 ns	2.05 ns	2.37 ns	2.73 ns	4.05 ns
	Xilinx Spartan FPGA	Area	Area	54 gates; 44 CLBs; 0 H	126 gates; 104 CLBs; 1 H	288 gates; 241 CLBs; 1 H	645 gates; 547 CLBs; 0 H	1382 gates; 1233 CLBs; 38 H
			Delay	25.05 ns	28.84 ns	32.55 ns	32.55 ns	36.15 ns
		Delay	Area	61 gates; 62 CLBs; 16 H	157 gates; 135 CLBs; 32 H	385 gates; 327 CLBs; 64 H	883 gates; 767 CLBs; 150 H	1408 gates; 1426 CLBs; 212 H
			Delay	25.44 ns	28.84 ns	29.94 ns	32.54 ns	35.54 ns

Table 4.8: Register Load Optimized Mask-based Two's Complement Area and Delay Synthesis Result

INTENTIONAL SECOND EXPOSURE

Version	Synthesize	Optimize	Report	8	16	32	64	128
Mux-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	239 gates	**543 gates**	1215 gates	2686 gates	5888 gates
			Delay	2.88 ns	**3.60 ns**	4.98 ns	7.74 ns	13.12 ns
		Delay	Area	**253 gates**	565 gates	**1242 gates**	**2792 gates**	**5969 gates**
			Delay	**2.72 ns**	3.53 ns	**4.66 ns**	**5.31 ns**	**6.90 ns**
	Xilinx Spartan FPGA	Area	Area	60 gates; 58 CLBs; 3 H	137 gates; 132 CLBs; 5 H	313 gates; 300 CLBs; 12 H	691 gates; 665 CLBs; 23 H	1510 gates; 1459 CLBs; 50 H
			Delay	28.34 ns	35.03 ns	42.23 ns	42.03 ns	46.13 ns
		Delay	Area	61 gates; 58 CLBs; 2 H	140 gates; 135 CLBs; 4 H	315 gates; 302 CLBs; 13 H	695 gates; 668 CLBs; 27 H	1519 gates; 1463 CLBs; 49 H
			Delay	27.94 ns	31.65 ns	36.64 ns	39.84 ns	44.44 ns
Mask-based Data Reversal	LSI Logic lca300k ASIC	Area	Area	**259 gates**	**627 gates**	1632 gates	3523 gates	**7164 gates**
			Delay	**2.85 ns**	**3.39 ns**	4.10 ns	4.54 ns	**6.62 ns**
		Delay	Area	365 gates	815 gates	**1619 gates**	**3302 gates**	8745 gates
			Delay	2.95 ns	3.41 ns	**3.86 ns**	**4.28 ns**	5.92 ns
	Xilinx Spartan FPGA	Area	Area	65 gates; 55 CLBs; 9 H	156 gates; 128 CLBs; 23 H	325 gates; 287 CLBs; 48 H	696 gates; 637 CLBs; 111 H	1738 gates; 1469 CLBs; 187 H
			Delay	34.34 ns	43.12 ns	43.54 ns	51.92 ns	67.79 ns
		Delay	Area	68 gates; 58 CLBs; 15 H	168 gates; 137 CLBs; 27 H	362 gates; 321 CLBs; 60 H	859 gates; 702 CLBs; 134 H	1965 gates; 1667 CLBs; 349 H
			Delay	29.94 ns	33.43 ns	37.03 ns	41.82 ns	52.12 ns

Table 4.7: Designs with Zero and Overflow Flags Area and Delay Synthesis Results

Version	Synthesize	Optimize	Report	8	16	32	64	128
Mask-based 2s Complement	LSI Logic lca300k ASIC	Area	Area	**189 gates**	509 gates	**1144 gates**	2550 gates	**5614 gates**
			Delay	**2.21 ns**	3.11 ns	**3.14 ns**	3.59 ns	**4.08 ns**
		Delay	Area	223 gates	**629 gates**	1531 gates	**3180 gates**	5669 gates
			Delay	2.34 ns	**2.05 ns**	2.37 ns	**2.73 ns**	4.05 ns
	Xilinx Spartan FPGA	Area	Area	54 gates; 44 CLBs; 0 H	126 gates; 104 CLBs; 1 H	288 gates; 241 CLBs; 1 H	645 gates; 547 CLBs; 0 H	1382 gates; 1233 CLBs; 38 H
			Delay	25.05 ns	28.84 ns	32.55 ns	32.55 ns	36.15 ns
		Delay	Area	61 gates; 62 CLBs; 16 H	157 gates; 135 CLBs; 32 H	385 gates; 327 CLBs; 64 H	883 gates; 767 CLBs; 150 H	1408 gates; 1426 CLBs; 212 H
			Delay	25.44 ns	28.84 ns	29.94 ns	32.54 ns	35.54 ns

Table 4.8: Register Load Optimized Mask-based Two's Complement Area and Delay Synthesis Result

Chapter 5

Conclusions and Future Research

5.1 Conclusions

This thesis has presented four approaches to barrel shifter design. Each design is a combination of techniques that affect two key factors. The first is the mechanism used to enable uni-directional operation support. The techniques used here are the mux-based and the mask-based approaches. The second mechanism is that which allows bi-directional operation support via the first's uni-directional hardware. The schemes used here are; data reversal, two's complement, and one's complement. These techniques test the differences between adjusting the data and manipulating the shift/rotate amount so that application leads to proper result computation.

The goal of analyzing these approaches is to determine those techniques that minimize gate count and critical path latency. After the designs were verified via simulation, they were synthesized to both the lca300k ASIC and the Xilinx Spartan FPGA. Results indicate that the Mux-based Data Reversal and the Mask-based Data Reversal approaches offer the best combination of both goals. There is, however, no clearly superior design since each has a comparable area-delay product and the Mux-based Data Reversal design has a lower gate count than the Mask-based Data Reversal design, while having a larger critical path latency. As such, use is dependent on individual demands of the hardware using the barrel shifter more than any other absolute consideration. The Mask-based Two's Complement and the Mask-based One's Complement designs proved to be too costly in both area and delay and are therefore of limited use.

5.2 Future Research

While this thesis focused on design optimizations to decrease both gate count and critical path latency, there exist many additional avenues of exploration regarding this subject. They range from design variations to implementation alterations. From a design perspective, one of the more interesting avenues is the use of a signed, as opposed to unsigned, shift/rotate amount [9], [10]. In this case, an additional direction signal is embedded in the shift/rotate amount as the sign bit. This may either allow for the removal of the direction indicator from the opcode or create a more versatile barrel shifter in that both direction signals are combined in order to determine the true direction of the operation.

Another design alteration that may be considered is the support of data widths unequal to an integer power of two. These data widths introduce complications since the methods used here to emulate bi-directional operation with uni-directional hardware are not easily altered for these data widths. It would, however, be useful to determine a method that supports these data widths since there exist applications where data is not an integer power of two.

At a somewhat lower level, exploration of a physical level layout of the approaches may make available some optimizations that were not accessible at the structural level. In particular, multiplexor modeling optimizations not available before can be performed here. The gains, if any, may provide insight into the designs not readily apparent. Of course, the importance of the alterations at the structural level are just as important since no design should rely entirely upon optimizations of this sort, as they are limited in their scope.

At an even lower level, one may wish to investigate the consequences of varying implementation technologies. This thesis limited its scope to a 0.6μm ASIC implementation. Technology has since improved, however, making available much smaller manufacturing

processes. These processes increase the relative cost of interconnects as line width begins to dominate. It would be interesting to see if at these processes, whether or not the Mask-based Two's Complement and Mask-based One's Complement become competitive to the other designs since they avoid, where possible, long interconnects.

Finally, as more and more hardware is integrated into mobile devices, power becomes a motivating factor. No power analysis was done for the designs, but it would be interesting to see how they compare. In addition, if this were to be combined with an analysis of multiple implementation technologies, the end result may give insight as to how design and implementation technology affect power consumption.

Bibliography

[1] M. Seckora, "Barrel shifter or multiply/divide IC structure," *U.S. Patent 5,465,222*, November 1995.

[2] J. Muwafi, G. Fettweis, and H. Neff, "Circuit for Rotating, Left Shifting, or Right Shifting Bits," *U.S. Patent 5.978,822*, December 1995.

[3] A. Yamaguchi, "Bidirectional Shifter," *U.S. Patent 5,262,971*, November 1993.

[4] A. Ito, "Barrel Shifter," *U.S. Patent 4,829,460*, May 1989.

[5] T. Thomson and H. Tam, "Barrel Shifter," *U.S. Patent 5,652,718*, July 1997.

[6] F. Worrell, "Microprocessor Shifter using Rotation and Masking Operations," *U.S. Patent 5,729,482*, March 1998.

[7] G.F. Burns, "Method for Generating Barrel Shifter Result Flags Directly from Input Data," *U.S. Patent 6,009,451*, December 1999.

[8] H.S. Lau and L. Tieu Ly, "Left Shift Overflow Detection," *U.S. Patent 5,777,906*, July 1998.

[9] M. Diamondstein and H. Srinivas, "Fast Conversion Two's Complement Encoded Shift Value for a Barrel Shifter," *U.S. Patent 5,948,050*, September 1999.

[10] K. Dang and D. Anderson, "High-Speed Barrel Shifter," *U.S. Patent 5,416,731*, May 1995.

Vita

Matthew Rudolf Pillmeier was born in Morristown, New Jersey on the 24th of May, 1979. He is the oldest son of Rudolf Pillmeier and Denise Pillmeier. After graduating from West Morris Mendham High School in 1997, he started his undergraduate study in the Department of Computer Engineering of Lehigh University, Bethlehem, Pennsylvania and obtained his B.S. degree in 2001. He graduated with highest honors and is currently a member of Tau Beta Pi, Omicron Delta Epsilon, and Phi Eta Sigma national honor societies.

END OF
TITLE