1992

Implementation of a high performance floating point unit multiplier.

Biniam. Mesfin
University of Windsor

IMPLEMENTATION OF A HIGH PERFORMANCE FLOATING POINT UNIT MULTIPLIER

by

Biniam Mesfin

A Thesis

Submitted to the Faculty of Graduate Studies through the

Department of Electrical Engineering in Partial Fulfilment

of the Requirements for the Degree of

Master of Applied Science at the

University of Windsor

Windsor, Ontario, Canada

May, 1992

ISBN 0-315-78864-X

Canada

ABSTRACT

This work presents a new fast and efficient algorithm for a floating point multiplier that adheres to the IEEE 754 standard and also investigates its VLSI implemetation. As a verification tool, VHDL is used to simulate the hardware model of the new floating point multiplier algorithm. In addition this work describes and compares several parallel multiplier architectures including a new parallel multiplier architecture which is both time optimal and regular in structure. This new multiplier architecture will be used as part for the new floating point multiplier algortihm. Finally the BICMOS implementation of the new multiplier architecture is discussed.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to Dr. G. A. Jullien for his tremendous support and guidance throughout my undergraduate and post-graduate program. My thanks are to Dr. M. Ahmadi for his encouragement and indispensable advice. I thank Dr. N. Wigley for kindly consenting to be the external examiner.

I would also like to express my special thanks to Dr. Wang for his opinions and ideas. In addition, I would also like to thank Mr. B. Erickson, Mr. M. Jovanovic, Mr. S. Bizzan, Mrs. A. Sarkar and Mr. Chan Wa Lai for their inexpressible support.

Thanks must also go to my parents for their love and care, my sisters and brothers for their patience, understanding and encouragement throughout the progress of my thesis.

Finally, I would like to thank all those people not mentioned here in particular the VLSI research group members, who aided me to complete my post-graduate program.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Introduction

The growing market for fast floating-point co-processors, digital signal processing chips, and graphics processors has created a demand for high speed area efficient multipliers. Current architectures range from small, low-performance shift and add multipliers, to large, high performance array and tree multipliers. Conventional linear array multipliers achieve high performance in regular structure, but require large amounts of silicon. Tree structures achieve even higher performance than linear arrays but the tree interconnection is more complex and less regular making them even larger than linear arrays. Ideally, one would want the speed benefits of a tree structure, the regularity of an array multiplier, and the small size of a shift and add multiplier.

This thesis considers an implementation of a new tree multiplier architecture which is faster than linear arrays, and more regular than traditional multiplier trees. In addition, since the need for high speed and high precision computation has been increasing in applications for

image processing, computer graphics, model simulation and so on this thesis will investigate a hardware implementation of a floating point multiplier that adheres to the IEEE 754 standard. This standard will be explained briefly in the next section starting with its origin.

1.2 Historical Overview Of Floating Point Systems

Almost all of the early computers provided only fixed point arithmetic operations, the only exceptions being the Model V Relay Computer designed by George R. Stibitz of the Bell telephone Laboratories [1,2] and the Harvard Mark II Computer [3] designed by Horward Aiken. To facilitate scientific and engineering calculations, the early machines often used long word lengths for number representations. Forty bit numbers were used in machines patterned after an Institute of Advanced Study Machine, and forty five bit numbers were used in the SEAC (Bureau of Standards Eastern Automatic Computer) family. However, merely providing greater precision for number representation did not solve the need for greater range in the size of the numbers. To scale large numbers into the range afforded by the machine involved a great deal of programming effort, as well as a rather thorough analysis of the problem being solved so as to determine the appropriate scaling factors in advance. The technique of automatic scaling or floating point arithmetic, came into wide spread use in the mid 1950's, first as a software option and as a hardware feature. Nowadays all computers for scientific and engineering use have built in floating point features.

Presently there are more than twenty different floating point formats in use by various computer manufacturers. Table 1.1 shows the formats of three computer which are popular for scientific computing. From table 1.1 it can be simply seen that there is hardly any similarity between the various formats. This situation which prohibits data portability

produced by numerical software, was the main motivation for setting up in 1978 an IEEE (Computer Society) 754 committee to standardize floating point arithmetic. The main goal of the standardization effort was to establish a standard which will allow communication between systems at the data level without the need for conversion.

System /Bits_Layout	IBM/370	DEC PDP-11	CDC Cyber 70
	S = Short L = Long	S = Short L = Long	
Word length	S: 32 bits L: 64 bits	S: 32 bits L: 64 bits	60 bits
Exponent	7 bits	8 bits	11 bits
Significand	S: 6 digits L: 14 digits	S: (1) + 23 bits L: (1) + 55 bits	48 bits
Bias of Exponent	64	128	1024
Radix	16	2	2
Hidden '1'	No	Yes	No

Table 1.1: Comparison of floating point specification of three popular computers

In addition to the respectable goal of the " the same format for all computers ," the committee wanted to ensure that it would be the best possible standard for given number of bits. Specifically, the concern was to ensure correct results, that is the same as those given by the corresponding infinite precision with an error of 1/2 LSB. Furthermore, to ensure portability of all numerical data. A more detail explanation of the IEEE 754 standard will be given in the next chapter.

1.3 Thesis Organization

The next chapter provides background information on the IEEE 754 standard. In particular the three major aspects of the IEEE 754 floating point standard are dexcribed, that is the format of the data types, the arithmetic and the exception handling. In addition, it covers in detail the IEEE 754 rounding modes.

Chapter 3 focuses on the VLSI implementation of a floating point multiplier which adheres to the IEEE 754 standard presented in Chapter 2. This chapter also introduces a high performance hardware model for a floating point multiplier which incorporates several high performance rounding algorithm proposed by Mark Santario. In addition, in this chapter a new fast and efficient method for computing the exponent and the sticky bit are presented.

Chapter 4 covers the background information on the basics of binary multiplication. The advantages and disadvantages of various hardware multiplier architectures, including linear arrays, trees; two new multiplier architectures are also discussed. In addition, a comparison of several adder structures and BICMOS implementation of the two new multiplier architectures are presented.

Finally Chapter 5 presents a summary of the contributions of this thesis and describes directions for future investigations.

CHAPTER 2

IEEE Floating Point Standard

2.1 Introduction

This chapter provides a tour of the IEEE 754 floating point standard. Each subsection discusses one aspect of the standard and why it was included. It is not the purpose of this thesis to argue that the IEEE 754 standard is the best floating point standard but rather to accept the standard as given. This chapter will start by first explaining the three major aspects in the IEEE 754 floating point standard [4] : the format of the data types, the arithmetic, and the exception handling.

2.2 Data Formats

The floating point data format is made up of three parts (from left to right): sign bit, biased exponent (characteristics), and significand (mantissa).

where:

S = Sign bit (indicates sign of the significand)

CE = Biased Exponent

F = Significand

then

e = true exponent = CE - bias

f = true significand = 1.F

A normalized non-zero number X, has the following interpretation

$$X = (-1)^S * 2^{CE-bias} * (1.F)$$

Floating point numbers are usually represented in normalized form. For example, both $0.01x10^1$ and 1.00×10^{-1} represent 0.1. If the leading digit is non-zero the representation is said to be normalized. The floating point number 1.00×10^{-1} is normalized, whereas $0.01x10^1$ is not. Requiring that a floating point representation be normalized makes the representation unique. Unfortunately this restriction makes it impossible to represent zero. A natural way to represent zero is with $1.0 \times 2^{(emin\ -1)}$ (emin is the smallest allowable exponent), since this preserves the fact that the numerical ordering of non-negative real numbers corresponds to the lexicographical ordering of their floating point representation. When the exponent is stored in a k bit field, this implies that only $2^k - 1$ values are available for use as exponents, since one must be reserved to represent zero.

2.3 Precision

The IEEE 754 standard defines four different precisions: Single, Double, Single_Extended and Double_Extended.

2.3.1 Single Precision

Single precision occupies a single 32 bit word. A 32-bit format for binary floating point number X is divided as shown in Figure 2.1. The component fields of X are the 1-bit sign S, the 8-bit biased exponent CE, and the 23-bit fraction f. The value V of X is as follows:

(a) If CE = 255 and f# 0, then V = NAN

(b) If CE = 255 and f=0, then $V = (-1)^S$ x infinity

(c) If $0 < CE < 255$, then $V = (-1)^S 2^{CE-127} (1.f)$

(d) If CE = 0 and f # 0, then $V = (-1)^S 2^{-126}(0.f)$

(e) If CE = 0 and f = 0, then $V = (-1)^S 0$, (Zero)

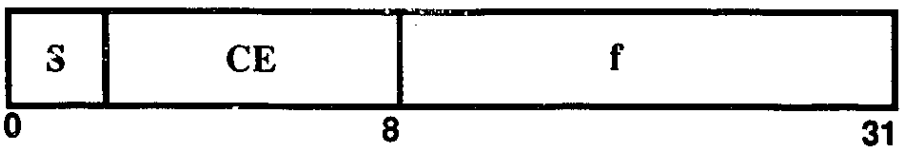


Figure 2.1: Single Precision Format

2.3.2 Double Precision

Double precision occupies a 64-bit word. The 64-bits for a binary floating point number X is divided as shown in Figure 2.2. The component fields of X are the 1-bit sign S, the 11-bit biased exponent CE, and the 52-bit fraction f. The value of X is as follows:

(a) If CE = 2047 and f# 0, then V = NAN

(b) If CE = 2047 and f=0, then V = $(-1)^S$ x infinity

(c) If 0 < CE < 2047 , then V = $(-1)^S 2^{CE-1023}$ (1.f)

(d) If CE = 0 and f # 0, then V = $(-1)^S 2^{-1022}$(0.f)

(e) If CE = 0 and f = 0, then V = $(-1)^S$0, (Zero)

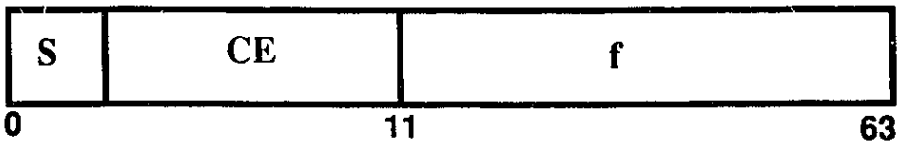


Figure 2.2: Double Precision Format

2.3.3 Single-Extended

Extended is an implementation dependent format. An extended binary floating point number X has four components: a 1 bit sign S, an exponent CE of specified range combined with an implementation dependent bias, a 1-bit integer j, and a fraction f with at least 31 bits. The exponent shall range between a minimum value m < -1023 and a maximum value M >1024. The value of V of X is as follows:

(a) If CE = M and f# 0, then V = NAN

(b) If CE = M and f=0, then V = $(-1)^S$ x infinity

(c) If m < CE < M , then V = $(-1)^S 2^{CE}$ (j.f)

(d) If CE = m and j = f = 0, then V = $(-1)^S$ 0, (normal zero)

2.3.4 Double-Extended

The double-extended format is the same as single extended described in 2.3.3, except that the exponent can range between m < -16383 and M > 16384, and a fraction shall have at least 63 bits.

2.4 Exponent

Since the exponent can be positive or negative, some methods must be chosen to represent its sign. Two common methods of representing signed numbers are sign/magnitude and two's complement. Sign/magnitude is the system used for the sign of the significand in the IEEE formats: 1 bit used to hold the sign; the rest of the bits represent the magnitude of the number. The two's complement representation is often used in integer arithmetic.

The IEEE binary standard does not use either of these methods to represent the exponent but instead uses a biased representation. In the case of single precision, where the exponent is stored in 8 bits, the bias is 127 (for double precision it is 1023). What this means is that if CE is the value of the exponent bits interpreted as an unsigned integer, then the exponent of the floating point number is CE-127. This is often called the biased exponent to distinguish it from the unbiased exponent E. An advantage of biased representation is that non-negative floating point numbers can be treated as integers for comparison purposes.

2.5 Arithmetic Operations

All conforming implementations of the IEEE standard provide the following operations:

1. Numerical Operations

    i) Add

    ii) Subtract

    iii) Multiply

    iv) Divide

    v) Square Root

    vi) Reminder

2. Conversion Operations

    i) Floating point ⟷ Integer

    ii) Binary(integer) ⟷ Decimal (integer)

    iii) Binary (floating) ⟷ Decimal (floating)

3. Miscellaneous Operations

    i) Move from one format width to another

    ii) Compare and set condition code

    iii) Find Integer part

2.6 Exception Handling

On some floating point hardware every bit pattern represents a valid floating point number. The IBM system/370 is an example of this. On the other hand, the VAX reserves some bit patterns to represent special numbers called *reserved operands* . The idea goes back to the CDC 6600, which had bit patterns for the special quantities Indefinite and Infinity.

The IEEE 754 standard continues in this tradition and has NANs (not a number) and infinities. Without special quantities, there is no good way to handle exceptional situations, such as taking the square root of a negative number, other than aborting the computation.

Under IBM System/370 FORTRAN, the default action in response to computing the square root of a negative number results in the printing of an error message. In IEEE arithmetic a NAN is returned in this situation. The IEEE standard specifies the special quantities shown in table 2.1.

Exponent	Fraction	Represents
$e = e_{min} - 1$	$f = 0$	± 0
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$
$e_{min} \leq e \leq e_{max}$	-	$1.f \times 2^{e}$
$e = e_{max} + 1$	$f = 0$	$\pm$ infinity
$e = e_{max} + 1$	$f \neq 0$	NAN

Table 2.1: IEEE 754 Special Values

2.6.1 NANs

Traditionally, the computation of 0/0 or the square root of -1 has been treated as an unrecoverable error that causes computations to halt. There are however, examples for which it makes sense for a computation to continue in such a situation. Consider a subroutine that finds the zeros of a function f, say zero(f). Traditionally, zero finders require the user to input an interval [a,b] on which the function is defined and over which the zero finder will search. That is, the subroutine is called as zero(f,a,b). A more useful zero finder would not require the user to input this extra information. This more general zero finder is especially appropriate for calculators, where it is a natural to key in a function and awkward to then have to specify the domain. It is easy, however, to see why most

zero finders require a domain. The zero finder does its work by probing the function f at various values. If it is probed for a value outside the domain of f the code for f might well compute 0/0 or the square root of -1, and the computation will halt, unnecessarily aborting the zero finding process.

This problem can be avoided by introducing a special value called NAN and specifying that the computation of expressions like 0/0 and the square root of -1 produce NAN rather than halting. A list of the situations that can cause a NAN is given in table 2.2. Then, when zero(f) probes outside the domain of f, the code for f will return NAN and the zero finder can continue. That is, zero(f) is not "punished" for making an incorrect guess. With this example in mind, it is easy to see what the result of combining NAN with an ordinary floating point number should be. Suppose that the final statement of f is return(-b + sqrt(d))/(2a). If d<0, sqrt(d) is a NAN and -b + sqrt(d) will be a NAN. Similarly, if one operand of a division operation is a NAN, the quotient should be a NAN. In general, whenever a NAN participates in a floating point operation, the result is another NAN.

In the IEEE 754 standard, NANs are represented as floating point numbers with the exponent e_{max} +1 and non-zero significands.

2.6.2 Infinity

Just as NANs provide a way to continue a computation when expressions like 0/0 or the square root of -1 are encountered, infinities provide a way to continue when an overflow occurs. This is much safer than simply returning the largest representable number. As an example, consider computing $(x^2 + y^2)^{1/2}$, when base(β)=10, precision(p) =3 and e_{max} = 98. If $x = 3x10^{70}$ and $y = 4x10^{70}$, then x^2 will overflow and be replaced by 9.99 x 10^{98}. Similarly y^2 and $x^2 + y^2$ will each overflow in turn and be replaced by 9.99 x 10^{98}. So the final result will be $(9.99 \times 10^{98})^{1/2} = 3.16 \times 10^{49}$, which is drastically in

error. The correct answer is $5x10^{70}$. In IEEE 754 standard arithmetic, the result of x^2 is infinity, as is y^2, x^2+y^2, and sqrt($x^2 + y^2$). So the final result is infinity, which is safer than returning an ordinary floating point number that is considerably different from the correct answer.

Operation	NAN produced by
+	infinity + (- infinity)
-	0 x infinity
/	0/0, infinity/infinity
REM	x REM 0, infinity REM y
square root	sqrt(x)(when x < 0)

Table 2.2: Operations that produce a NAN

2.6.3 Signed Zero

Zero is represented by the exponent e_{min}-1 and a zero significand. Since the sign bit can take on two different values there are two zeros, +0 and -0. If a distinction were made when comparing +0 and -0, simple tests, such as *if (x=0)*, would have unpredictable behavior, depending on the sign of x. Thus, the IEEE standard defines comparisons so that +0 equals -0 rather than -0 < +0. Although it would be possible to always ignore the sign of zero, the IEEE standard does not do so. When a multiplication or division involves a signed zero, the usual sign rules apply in computing the sign of the answer. Thus, 3(+0) = +0 and +0/-3 = -0. If zero did not have a sign, the relation 1/(1/x) = x would fail to hold

when x = + or - infinity. The reason is 1/(- infinity) and 1/(+ infinity) both result in zero and 1/0 results in + infinity, the sign information has been lost . One way to restore the identity 1/(1/x) = x is to have only one kind of infinity, however that would result in the disastrous consequence of losing the sign of an overflowed quantity.

Another example of the use of signed zero concerns underflow and functions that have a discontinuity at zero such as log. In IEEE arithmetic, it is natural to define log(0) = negative infinity and log(x) to be a NAN when $x < 0$. Suppose x represents a small negative number that has underflowed to zero, with a signed zero, x will be negative so log can return a NAN. If there were no signed zero, however, the log function could not distinguish an underflowed negative number from 0 and would therefore have to return negative infinity.

Although distinguishing between +0 and -0 has advantages, it can occasionally be confusing. For example, signed zero destroys the relation $x = y \longleftrightarrow 1/x = 1/y$, which is false when x = +0 and y = -0. The IEEE committee decided, however that the advantages of using signed zero outweighed the disadvantages.

2.6.4 Denormalized Numbers

Consider normalized floating point numbers with $\beta = 10$, $p = 3$ and $e_{min} = -98$. The numbers $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$ appear to be perfectly ordinary floating point numbers, which are more than a factor of 10 larger than the smallest floating point number 1.00×10^{-98}. They have a strange property, however: $x - y = 0$ even though $x \neq y$. The reason is that $x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$ is too small to be represented as a normalized number and so be flushed to zero.

How important is it to preserve the property

$$x = y \longleftrightarrow x - y = 0 \; ? \qquad (2.1)$$

The IEEE standard uses denormalize numbers, which guarantee(2.1), as well as other useful relations. They are the most controversial part of the standard and probably accounted for the long delay in getting 754 approved. Most high performance hardware that claims to be IEEE compatible does not support denormalized numbers directly but rather traps when producing denormals, and leaves it to software to simulate the IEEE standard. The idea behind denormalized numbers goes back to Goldberg (1967) and is simple. When the exponent is e_{min}, the significand does not have to be normalized. For example, when $\beta = 10$, $p = 3$ and $e_{min} = -98$, 1.00×10^{-98} is no longer the smallest floating point number, because 0.98×10^{-98} is also a floating point number.

There is small snag when $\beta = 2$ and a hidden bit is being used, since a number with an exponent of e_{min} will always have a significand greater than or equal to 1.0 because of the implicit leading bit. The solution is similar to that used to represent 0 and is summarized in table 2.1. The exponent $e_{min}-1$ is used to represent denormals. More formally, if the bits in the significant field are $b_1, b_2, \ldots\ldots, b_{p-1}$ and a value of the exponent is e, then when $e > e_{min} - 1$, the number being represented is $0.b_1b_2 \ldots b_{p-1} \times 2e$, whereas when $e = e_{min}-1$ the number being represented is $0.b_1b_2 \ldots b_{p-1} \times 2^{e+1}$. The +1 in the exponent is needed because denormals have an exponent of e_{min}, not $e_{min}-1$. Recall the example $\beta = 10$, $p = 3$, $e_{min} = -98$, $x = 6.87 \times 10^{-97}$, and $y = 6.81 \times 10^{-97}$ presented at the beginning of this section. With denormals, x - y does not flush to zero but is instead represented by the denormalized number 0.6×10^{-98}. This behaviour is called *gradual underflow* . It is easy to verify that (2.1) always holds when using gradual underflow.

Figure 2.3 illustrates denormalized numbers. The top number line in the figure shows normalized floating point numbers. Notice the gap between 0 and the smallest normalized number $1.0 \times \beta^{e_{min}}$. If the result of a floating point calculation falls into this gap, it is flushed to zero. The bottom number line shows what happens when denormals are added to the set of floating point numbers. The gap is filled in, when the result of a calculation is less than $1.0 \times \beta^{e_{min}}$; it is represented by the nearest denormal. When denormalized numbers are added to the number line, the spacing between adjacent floating point numbers varies in a regular way. Adjacent spacings are either the same length or differ by a factor of β. Without denormals the spacing abruptly changes from $\beta^{-p+1}\beta^{e_{min}}$ to $\beta^{e_{min}}$, which is a factor of β^{p-1}, rather than the orderly change by a factor of β. As a result of this, many algorithms that can have relative error for normalized numbers close to the underflow threshold are well behaved in this range when gradual underflow is used. Figure 2.4 shows the error which results from gradual underflow comparing to flush to zero.

Example 2.1 :

Comparison of various schemes in six- digit decimal arithmetic with $e_{min} = -99$

$(1.23456 \times 10^{-60}) \times (6.54321 \times 10^{-40}) = 8.0779853376 \times 10^{-100}$ (Exact)

→ 0.80780×10^{-99} (Gradual Underflow)

→ 0.0 (Store 0)

→ UN (UN symbol)

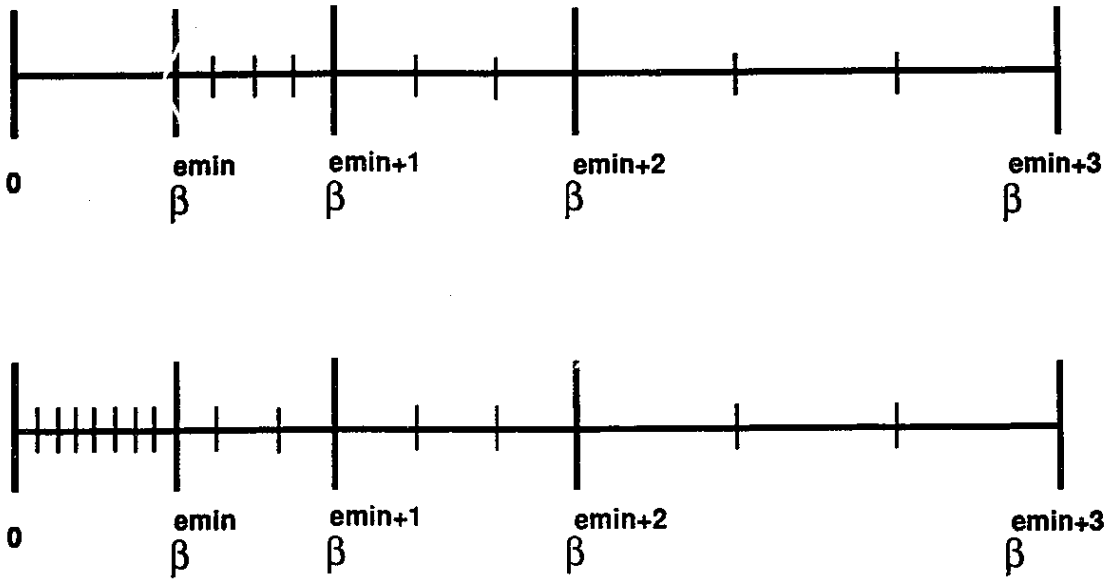


Figure 2.3: Flush to zero compared with gradual underflow

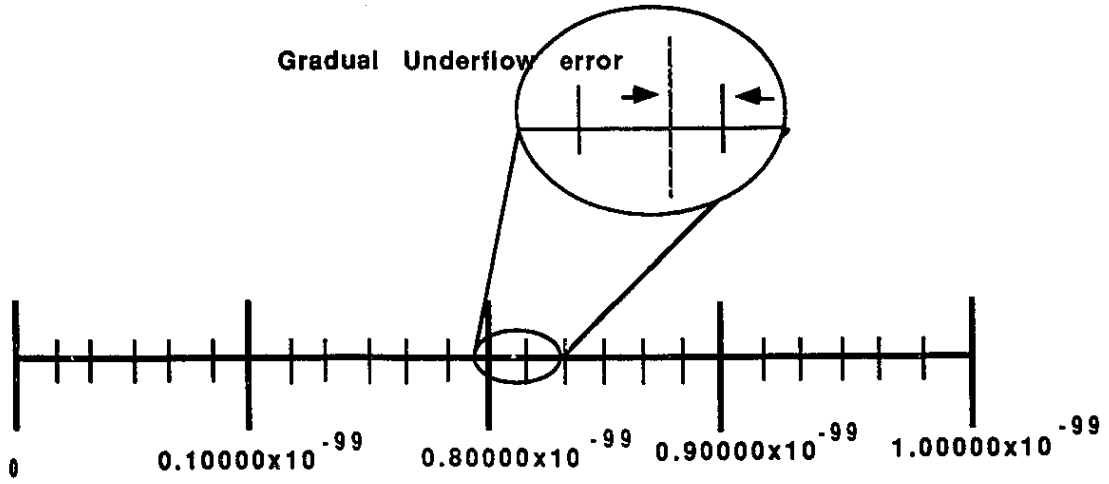


Figure 2.4: Gradual Underflow Error

2.7 Rounding

Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Although there many infinitely many integers, in most programs the result of integer computations can be stored in 32 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore, the result of a floating point calculation must often be rounded in order to fit back into its finite representation.

In IEEE 754 standard there are four rounding modes. Unbiased rounding to nearest, Round toward zero, Round toward minus infinity, and Round toward plus infinity. Of the four only the first is mandatory and the rest are optional. Unbiased rounding is very similar to the conventional round to nearest which is implemented by adding 1/2 of the digit to be discarded and then truncate to the desired precision.

Example 2.2:

39.2	39.7
00.5	00.5
39.7 → 39	40.2 → 40

However, suppose the number to be rounded is exactly half way between two number, which one is the nearest ? To answer the question let's add the same 0.5 to the two following numbers:

38.5	39.5
00.5	00.5
39.0 → 39	40.0 → 40

Notice that in both cases we rounded up even though each number was exactly half way between smaller and larger numbers. Therefore, by simply adding 0.5 and truncating a biased rounding is generated. In order to have unbiased rounding the IEEE 754 standard rounds to even when there is a tie between two numbers. Now, using the previous numbers we get:

38.5 ⟶ 38

39.5 ⟶ 40

In the first case the number is rounded down and in the second case the number is rounded up. Therefore, we have statistically unbiased rounding. Of course, the same unbiased rounding could be obtained by rounding to odd (instead of even) in the tie case. This time the rounding looks like this:

38.5 ⟶ 39

39.5 ⟶ 39

However, rounding to even is preferred because it may result in "nice" integer numbers as in the following examples:

1.95 ⟶ 2

2.05 ⟶ 2

Whereas rounding to odd results in the more frequent occurrence of non-integer numbers:

1.95 ⟶ 1.9

2.05 ⟶ 2.1

Now, we illustrate the implementation of unbiased rounding to even, and introduce the the so called "sticky bit".

The conventional system for rounding is to add half of the LSD position of the desired precision, to the MSD of the portion to be discarded. This scheme has a problem as is illustrated below (the XXXXX are additional bits). Thus:

38.5XXXXXX

00.5

39.0

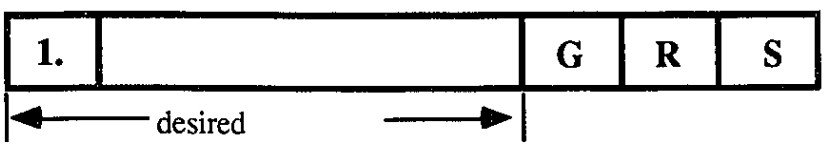
Two cases have to be distinguished:

Case 1: XXXXXX $\neq 0$ and the rounding is correct since 39 is nearest $38.5 + \delta$,

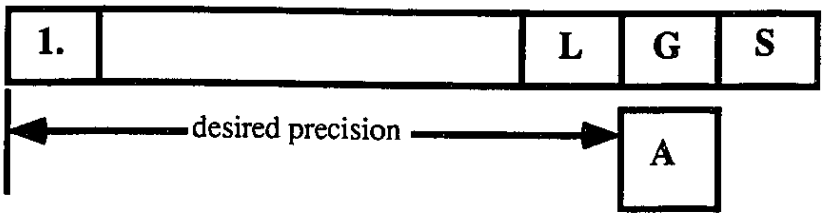
where $0 < \delta < 0.5$.

Case 2: XXXXXX = 0 Note the rounding is incorrect because we have a tie case which requires the result to be rounded to even (38).

It is obvious that regardless of the number of X bits, all possible permutations can be mapped into one of the two above cases. Therefore, one bit can be used to distinguish between Case 1 and Case 2. This bit is called the "sticky bit", and it has the value one for Case 1 and the value zero for Case 2. The logic implementation of the sticky bit is simply ORing of the bits to the right of the second guard bit, as illustrated below for the addition/subtraction operation



In the case of a left shift(normalization after subtraction), S does not participate in the left shift, but instead zeros are shifted into R. In the case of a right shift due to significand overflow (during magnitude addition), the R guard bit is ORed into S during the shift. The above format (with two guard bits) is necessary only during the addition/subtraction process: the final result just before rounding has only one guard bit and the sticky bit.



The proper action to obtain unbiased rounding to even is determined from the following table 2.3:

L	G	S	Action	A
X	0	0	Exact result	X
X	0	1	Inexact result	X
0	1	0	The tie case with even significand	0
1	1	0	The tie case with odd significand	1
X	1	1	Rounding to nearest	1

Table 2.3: Unbiased Rounding

In addition to round to nearest, as mentioned previously the IEEE 754 standard define three other optional rounding modes. These "directed" rounding modes are round toward plus infinity, round toward minus infinity and round toward zero. Once round to nearest has been implemented, the other rounding modes are relatively simple. To begin

with, consider round toward zero. This is simply a truncation. We now look at round toward plus infinity. The standard states that "when rounding toward plus infinity the result shall be the the format's value (possibly plus infinity) closest to and no less than the infinitely precise result". Basically, what this says is that in case of positive result, if all the bits to the right of the LSB of the desired result are zero, then the result is correct. If any of these bits are a one then a 1 should be added to the LSB of the result. If the result is negative it should be truncated. When rounding toward minus infinity the exact opposite holds.

In conclusion, this chapter has described the IEEE 754 floating point standard which is widely accepted by most computer manufacturers. Details on the data formats, exception handling and rounding methods have been explained. In the next chapter, implementation of a floating point multiplier that satisfies the IEEE standard will be discussed.

CHAPTER 3

Floating Point Multiplier

3.1 Introduction

Many applications exist in which integer, or non-IEEE floating point multiplication is sufficient. However, to be widely accepted, current and future floating point co-processors must adhere to IEEE standard for floating point arithmetic [4]. The standard can be implemented in software, hardware, or a combination of the two [5]. The performance requirements of modern digital systems demand direct hardware floating point multipliers. To match the performance of the hardware multipliers, the rounding modes must also be implemented in hardware.

In this chapter a hardware implementation of a floating point multiplier will be presented. Since dealing with tie case for the round to nearest/even mode slows the performance of the multiplier, three algorithms will be presented for implementing round to nearest/up [6].

Then it will be shown how the round to nearest up result can be adjusted to produce the correct IEEE rounded result. In addition two techniques for computing the sticky bit will be presented. All of the rounding algorithms and sticky methods presented are technology independent and can be used with several types of multiplier architectures. Finally the floating point multiplier hardware model obtained by incorporating the fast rounding techniques will be given.

3.2 Hardware Implementation of a FPU Multiplier

A hardware realization of a floating point unit multiplier is shown in Figure 3.1. This algorithm is made up of two distinct sections. On the right side the mantissa are handled as fixed point operands, while the left side computes the exponent. The mantissa, and the characteristic of the "A" operand are designated MA, and CA respectively, while MB, and CB are similar part of the B operand. The result of multiplication is designated by R; and MR, and CR are the mantissa, and the characteristic of the R result. In general the floating point multiplication algorithm involve the following operations:

1) Perform fixed point multiplication of the mantissa of the two operands

2) Round the result

3) Normalize the result of the mantissa multiplier

4) Add the two characteristics and correct for bias

5) Check for underflow and overflow conditions

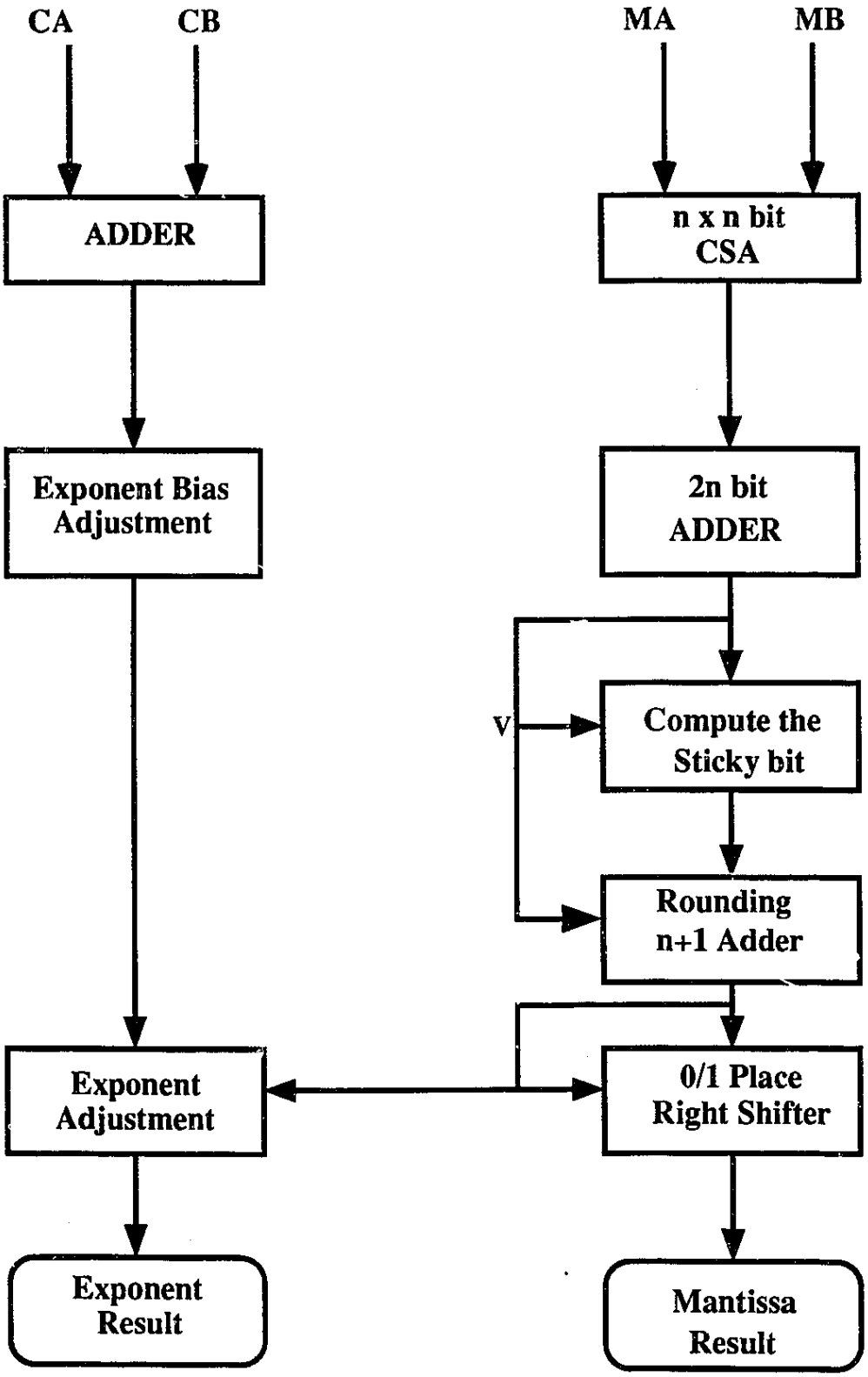


Figure 3.1: Floating Point Multiplier Data Path

The performance of the algorithm shown in Figure 3.1 can be enhanced by removing the computation of the sticky bit from the critical path and incorporating Mark Santario fast rounding techniques [6]. These two techniques will be presented in the next sections.

3.3 A Simple Round to Nearest/up Algorithm

Most high performance VLSI multipliers use some sort of array or tree structure to sum the partial products in the mantissa portion of a floating point multiply [7]. Figure 3.2 shows a flow diagram for the mantissa handling section of a floating point multiply unit. This simple round to nearest/up scheme will be referred to as Algorithm 3a.

The top section (multiply) accepts two normalized mantissas and uses some type of reduction structure which produces the product in carry save form (two 2n bit numbers). These two numbers are then added in the CPADD section to produce a complete 2n bit product. The are two possible rounding operations which then occur, depending on the most significant bit(MSB) of this product. If the resulting product is in the range $2 \leq$ product < 4 (overflow), the constant $2^{(-n+1)}$ is added to the product and the result is truncated to n-2 bits to the right of the decimal point. A normalization shift (Normal) of 1 to the right is then necessary to restore the rounded product to the range $1 \leq$ rounded product < 2, with an appropriate adjustment of the exponent. If the original 2n bit product was in the range $1 <$ product < 2 (no overflow), then the constant $2^{(-n)}$ is added. In most cases this rounded product will be less than 2, and the rounding operation is finished. However, it is possible that the addition of $2^{(-n)}$ could cause the rounded product to be equal to 2, in which case a normalization shift of 1 and an exponent adjustment is necessary (as in the left branch).

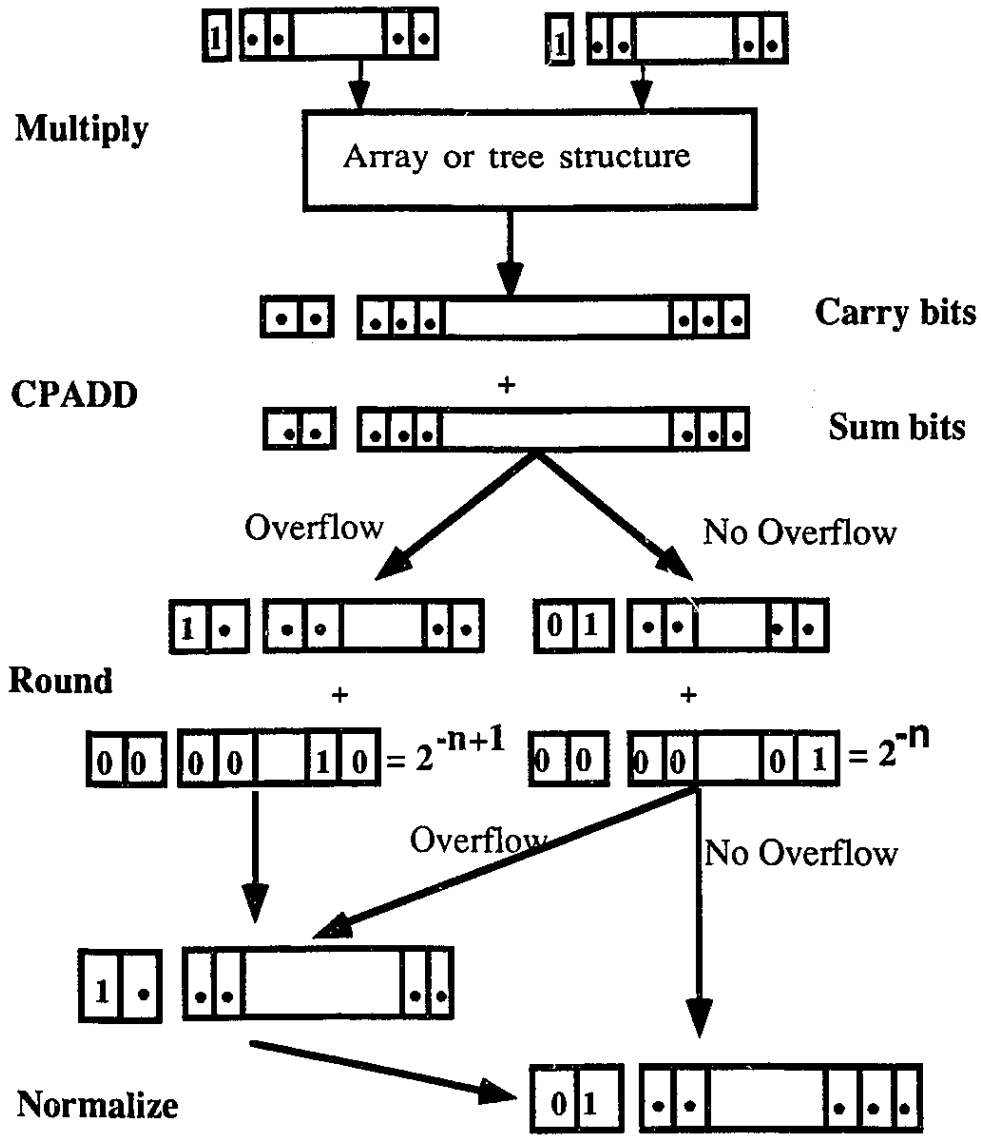


Figure 3.2: Algorithm 3a Data Flow

The low order n-2 bits from the CPADD section of Figure 3.2 are not used in any of the following steps. The only effect that these bits have on the final result is due to the carry they generate into the most significant n+2 bits. Thus, the carry propagate adder need never actually compute the sum of the least significant n-2 bits. The 2n bit carry propagate adder can be replaced by an n+2 bit carry propagate adder, with an input carry from the least significant n-2 bits. The small adder is clearly an advantage where a hardware

implementation is concerned. Algorithm 3a requires two carry propagate additions in series. In section 3.4 algorithm 3b and algorithm 3c concentrate on computing these additions in parallel, which significantly increases performance.

3.4 Parallel Addition Schemes

If an n+2 bit carry propagate adder is used in the **CPADD** section of Figure 3.2, then the carry from the lower bits (**Cin**) will be added at the $2^{(-n)}$ bit position. Assuming that no overflow occurred, an additional $2^{(-n)}$ will be added to the result in the **Round** section. The $2^{(-n)}$ bit position will thus be called the round bit position, or R bit. The 1 that always get added to the R bit position for rounding will be identified as **Rin.** If no overflow occurs, adding Cin and Rin to the R bit position will produce the correct round to nearest/up result.

Now consider the overflow case. The MSB, known as the overflow bit (V), is a 1. By assuming that no overflow would occur, $2^{(-n)}$ was added for rounding. If an overflow did occur, then $2^{(-n+1)}$ should have been added for rounding. The difference of $2^{(-n)}$ must be added to correct the rounding. This can be done by defining a new bit that is added to the $2^{(-n)}$ bit position in the case of an overflow. This bit will be called the overflow rounding bit (**Rv**). The correct rounding can thus be obtained by simply adding the carry from the lower order bits (Cin), the rounding bit (Rin), and in the case of an overflow (Rv) to the R bit position. This bits are shown in Figure 3.3.

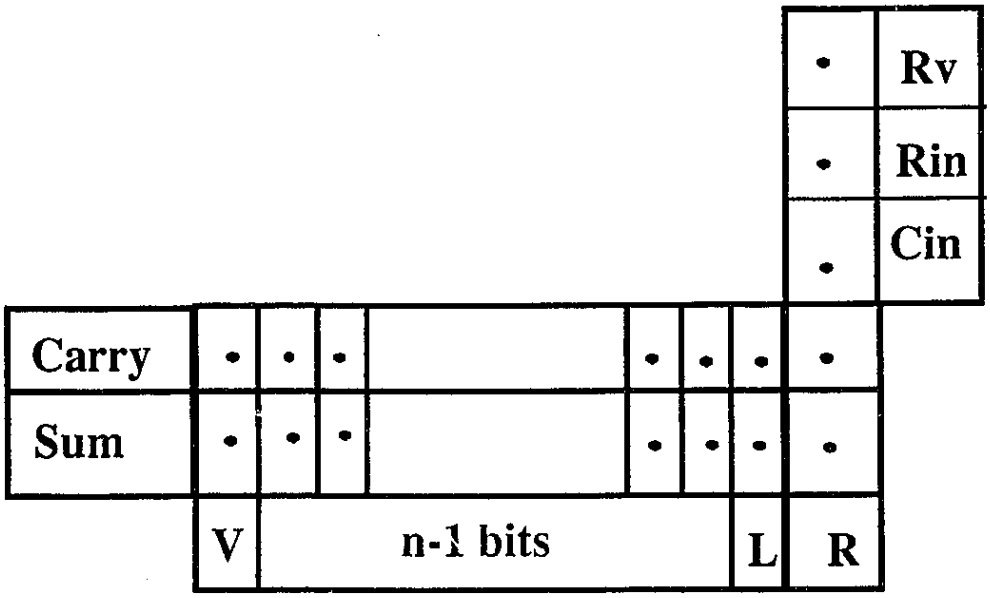


Figure 3.3: Bits to be Summed for Correct Round to Nearest/up

Fast and effective implementation schemes for summing the bits in Figure 3.3 must overcome two problems. First, the value of Rv is not known until the sum of all the other bits have been computed. Second, an adder with 5 input slots at the LSB is required.

The first problem can be overcome by computing two carry propagate additions in parallel. The first assuming Rv=0, and second, assuming Rv=1. When the overflow condition is known, the correct sum can then be selected using a multiplexer. These two additions are related, as the first is simply one larger than the second. This provides many possibilities for the designer. An efficient technique is to simply merge the two carry propagate adders into one. A conditional sum adder (CSAdd), or carry select adder as it is often known, computes two possible outputs [8]. The first assumes the input carry is a zero, and the second assumes the input carry is a one. When the input carry is known the correct output is selected. This compound adder requires much less hardware than two separate adders,

since only the carry chain need be duplicated. In the more general sense, a conditional sum type adder produces two results in the form of A+B and A+B+1.

Now for the second problem Rcarry and Rsum use the carry and sum slots. Rv uses the input carry slot to the CSAdder. This leaves no empty slots for Rin and Cin to be added to the R bit position. Two algorithms were proposed by Mark Santario to fix this problem [6]. Both involve adding Cin and Rin to the R bit position, without propagating the carry, before computing the carry propagate result.

The data flow of algorithm 3b is shown in Figure 3.4. A row of half adders is used to partly sum the carry and sum bits. This leaves a hole in the CSAdder at Rcarry. The Cin from the lower order bits can be placed into this hole. Rin must still be added to the R bit position. An additional row of half adders could be used as on Cin, but there are more economical techniques. Array multipliers typically have empty slots. A 1 can often be injected into the array, or corresponding structure, in the appropriate place so that the effect is to add Rin to the R bit. Once Rin and Cin have been added to the R bit position and the CSAdd has completed, the correct result can be picked based upon the overflow bit from the A+B result. The V bit from the A+B result is used, because the overflow bit must be checked before Rv has been added in. The A+B+1 result has already added Rv to the sum, potentially corrupting the V bit. If the V bit from the A+B result is a 0, the A+B result is chosen. If the V bit is a 1 from the A+B+1 result is picked. In this case an overflow has occurred, the result must be normalized and the exponent adjusted.

In some cases a slot may not exist, or it may be difficult to inject Rin into the multiplier array or accumulator. Figure 3.5 shows the data flow for algorithm 3c. This algorithm is similar to algorithm 3b, except that Rin is not injected into the array. Instead, two least significant half adders are replaced with CSA's, providing two additional slots as the L and R bit positions. Rin, which is always 1, can be combined with Cin and placed into these

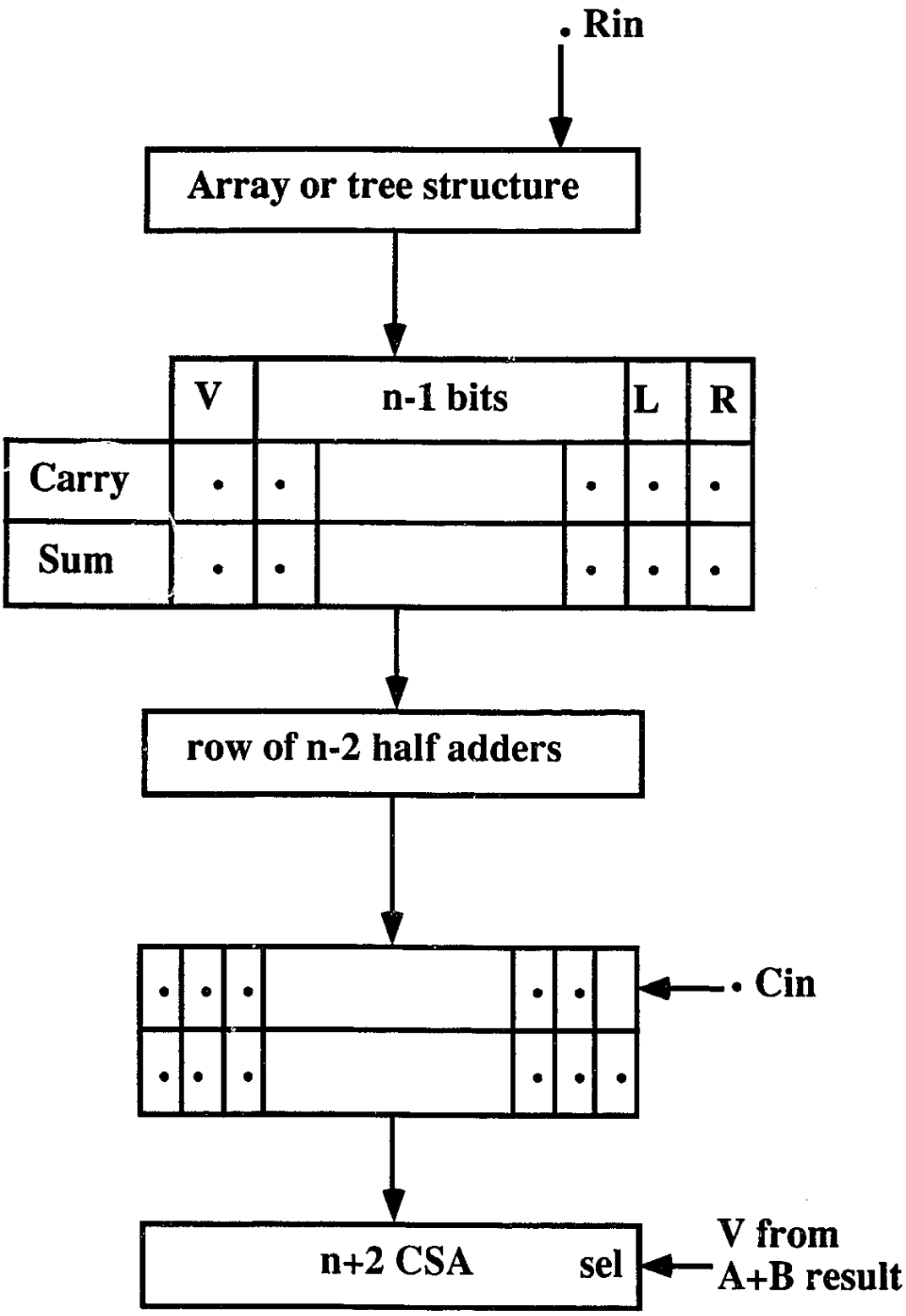


Figure 3.4: Algorithm 3b Data Flow

empty slots. If Cin equals 0, then a 1 from Rin should be added to the R bit. If Cin equals 1, then 2 should be added to the R bit position; one from Rin and one from Cin. Adding 2 to the R bit position is equivalent to adding 1 to the R+1(L) bit position. The output of the half-adder/CSA row may then fed to the CSAdder as in algorithm 3b.

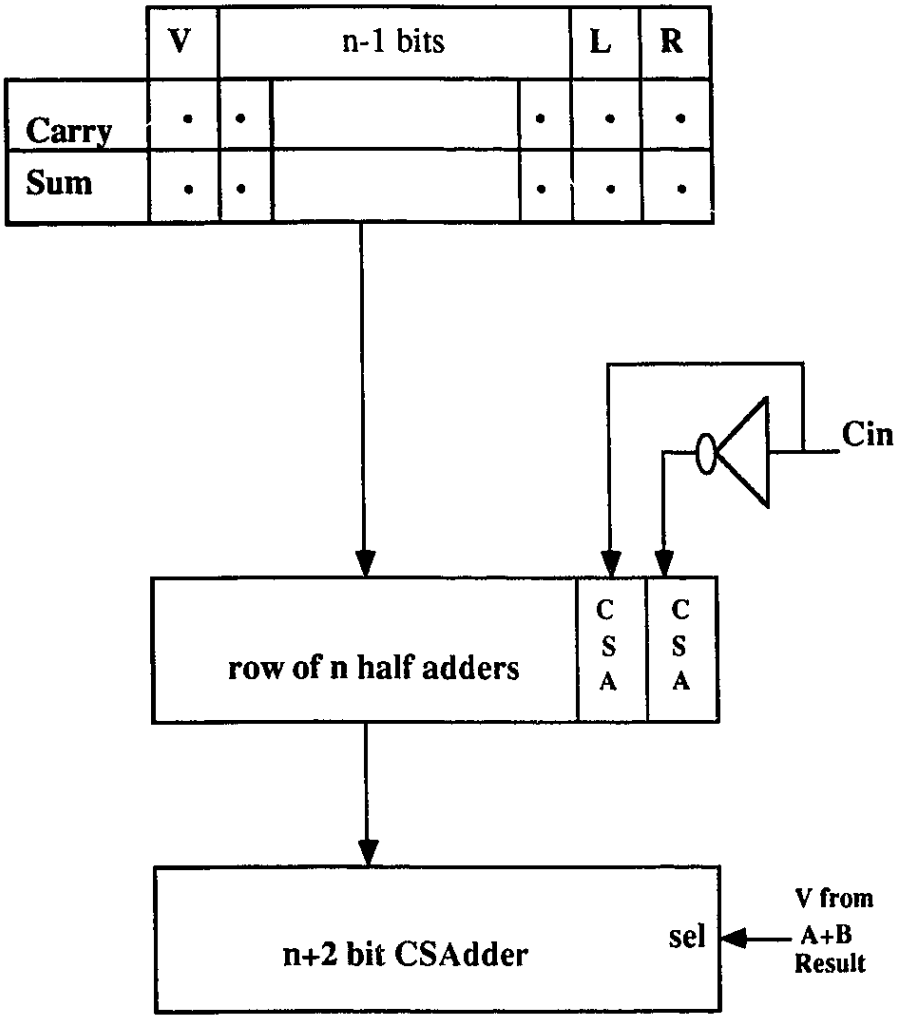


Figure 3.5: Algorithm 3c Data Flow

3.5 Obtaining the IEEE Round to Nearest Result

It was stated earlier that round to nearest/up produces exactly the same result as round to nearest/even, except when a tie occurs. To produce the correct round to nearest/even result from the unrounded result, a '1' is potentially added to the round bit. The bit (E) to be added to the round bit (R) for correct IEEE round to nearest is based up on the L, R, and sticky (S) bits as shown in table 3.1.

Before Rounding			Add to R bit		L after Rounding	
L	R	S	E	U	L_E	L_U
X	0	0	d	1	X	X
X	0	1	d	1	X	X
0	1	0	0	1	0	1
1	1	0	1	1	0	0
X	1	1	1	1	X	X

Where :

E = Bit added to R bit for correct round to nearest/even

U = Bit added to R bit for correct round to nearest/up

L_E = The L bit after round to nearest/even

L_U = The L bit after round to nearest/up

d = Don't care. E can not effect L_E

Table 3.1: Round to Nearest/even versus Round to Nearest/up

In contrast, round to nearest/up assumes that the bit to be added to the R bit for correct rounding (U) is always a 1. The only case where the round to nearest/up bit (U) will produce different result from the round to nearest/even bit (E) is shown in row 3 of table 3.1, where E=0, and U=1. In this case round to nearest/up changed the L bit from a 0 (L=0) to a 1 (L_E=1), while round to nearest/even left the L bit unchanged (L_U=0). The important thing to notice is that when round to nearest/up changed the L bit to a 1, the 1 was not propagated. As such, only the L bit was effected. This means that the correct round to nearest/even result can be obtained from the round to nearest/up result by restoring the L bit to a zero.

By assuming that the round bit will be a 1, the round to nearest/up algorithms have an advantage over the round to nearest/even method in that the carry propagate addition can take place before the sticky bit has been computed. This means that the round to nearest/up result can be obtained using any of the methods presented in the earlier sections. The correct IEEE round to nearest result can then be obtained by observing only the L, R and sticky bits, and forcing the L bit to a zero if required. The circuit used for restoring the L bit depending on L, R, and S bits is shown on Figure 3.6.

3.6 Computing the Sticky Bit

3.6.1 A Simple Method to Compute the Sticky Bit

The first method for determining the sticky bit is conceptually the simplest, as it stems from the very definition of the sticky bit. Recall that the sticky bit was defined to be equal to zero if the value of all the bits to the right of the round bit is zero. To determine the sticky bit, begin with a carry propagate addition on all of the bits. The Sticky bit(S) will be the OR of all the bits to the right of the R bit. This method is very simple in concept, and is

often used in practice. One drawback is that a full carry propagate addition, followed by a logical OR, must be done on all of the lower order carry save bits.

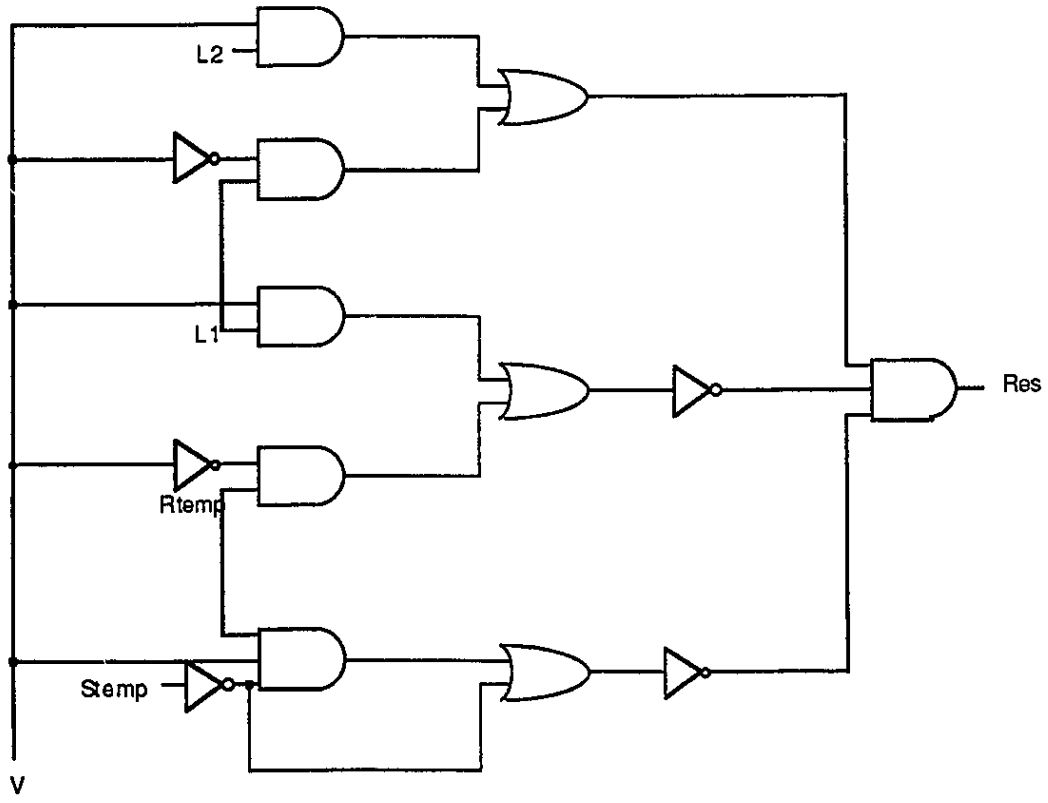


Figure 3.6: L Bit Restorer

3.6.2 Computing Sticky Bit From Input Operands

The sticky bit may also be computed directly from the inputs to be multiplied, bypassing the multiply array completely. The number of trailing zeros in the binary number X●Y(product of X and Y) is exactly equal to the number of trailing zeros in X plus the number of trailing zeros in Y (Note the number of trailing zeros in the product is exactly equal to the sum of the trailing zeros in the operands, for any representation in which the

base is prime. This is true because prime numbers cannot be factored. Non prime bases can be factored; therefore, the number of trailing zeros in the product can be greater than the sum of the trailing zeros in the operands). The trailing zeros in X and Y can be counted and summed while the multiply is taking place. If the sum is greater than the sum of the bits to the right of the round bit, then the sticky bit is a zero. The advantage of using this method is that the sticky bit can be computed in parallel with the actual multiplication, removing the sticky bit from the critical path.

The floating point multiplier which incorporates the fast rounding technique and the new sticky bit computation technique is shown in Figure 3.7. This new floating point algorithm was fully simulated using a VHDL simulator. From the simulation results it was found that even though this new algorithm has a better performance than the one shown in Figure 3.1 it has the following three major drawbacks.

First, since the result of the mantissa multiplier is in range $1 \leq (1.F) < 4$, the exponent must increment when V=1 (or the mantissa is between $2 \leq (1.F) < 4$). This increment of the exponent can not be done until the mantissa multiplication is completed. Thus this operation is on the critical path of the floating point multiplication.

Second, the exponent data path is twenty percent slower than the mantissa data path. Therefore the speed of the multiplier is determined by the exponent datapath.

Thirdly, the performance gain obtained by computing the sticky bit from input operands is offset by its hardware complexity.

The first problem was solved by using a conditional sum adder to remove the computation of the exponent from the critical path. To solve the other problems, new techniques for computing the sticky bit and exponent have been developed. These techniques are explained in the next sections.

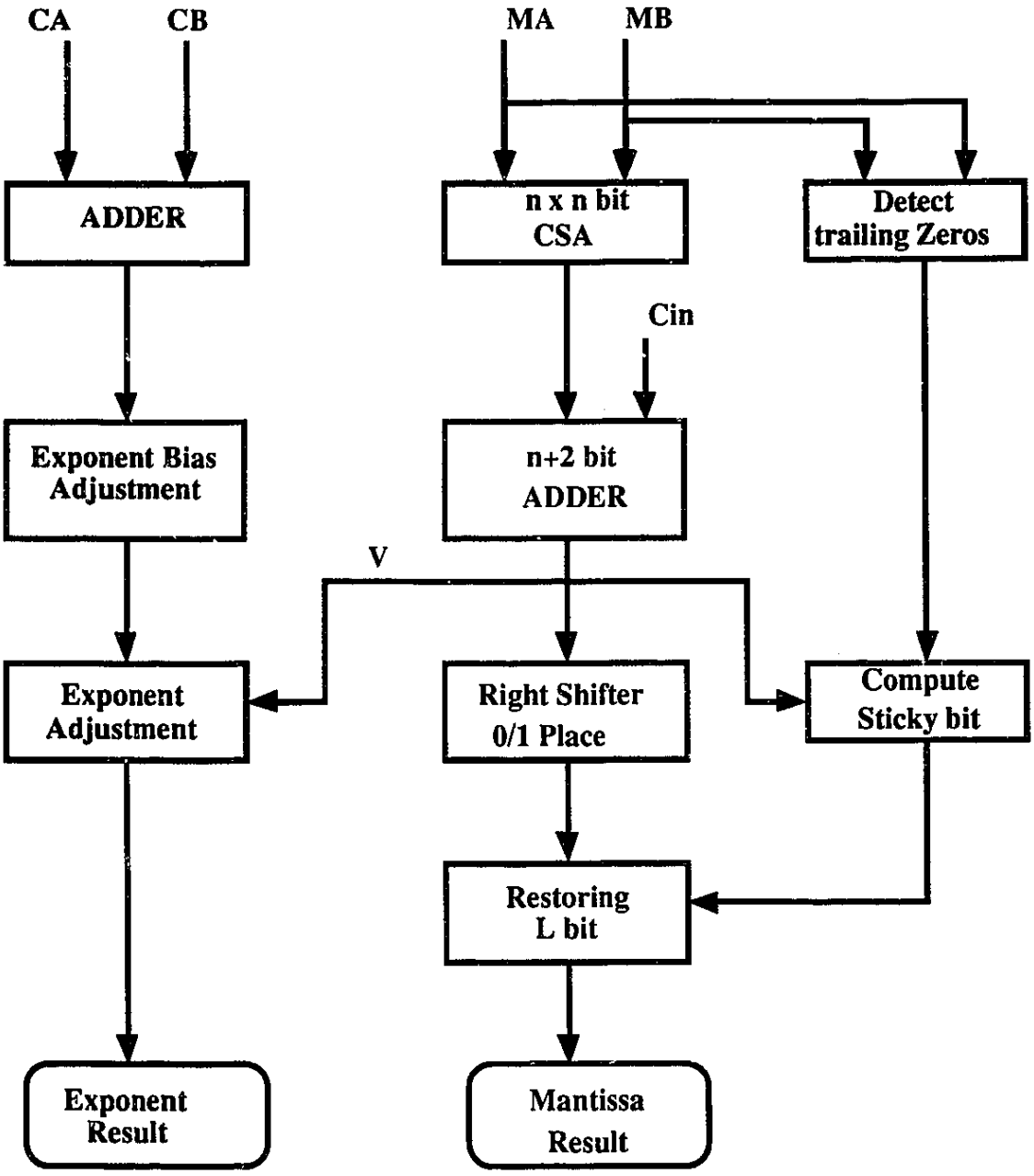


Figure 3.7: Floating Point Data Path which Incorporates the Fast Rounding Techniques

3.7 A New and Efficient Technique for Computing the Sticky Bit

The new technique computes the sticky bit by ORing the Carry Save bits. That is, a simple logical OR on the Carry Save bits formed from the bits to the right of the round bit will yield the correct sticky bit. This simple ORing of the carry save bits works for the following reasons. If the 2n bit carry save result is scanned from right to left, the first non-zero carry/sum pair will contain a single one. That is, either the carry or the sum will be a 1 but not both. This single one could not generate a carry during a carry propagate addition, and since all of the bits to its right are zero, there is no carry to propagate. This will cause a single one to remain in its current position. If this position is to the right of the R bit, the sticky bit will be a 1.

To see why this is true, refer to Figure 3.8. This figure shows a section of the partial products for the multiplication of A●B. Each row represents a single partial product which will be generated and later summed to form the carry save form of the final product. AOBO represents the partial product represented by the logical AND of bit A0 with bit B0, and so on.

Assume A2B2 in column 4 is a 1, and column 4 is the first column in which a 1 appears. Since A2 is a 1, B1 and B0 must both be zero, or there would be a one in an earlier column formed by A2B1 in column 3, row 1, or A2B0 in column 2, row 0. All products above A2B2 in column 4 contain either a B1 or a B0 and thus must be zero. Looking across row 2, B2 is a 1. This means A1 and A0 must be both zero, or a 1 would exist in columns 3 or 2. All products below A2B2 in column 4 contain either A1 or A0, and thus must also be zero. Therefore, A2B2 is the only non-zero partial product in this column. This can easily

be generalized to any element in any column, proving that the first column in which a 1 exist will contain a single 1.

	5	4	3	2	1	0
0	A5B0	A4B0	A3B0	A2B0	A1B0	A0B0
1	A4B1	A3B1	A2B1	A1B1	A0B1	
2	A3B2	A2B2	A1B2	A0B2		
3	A2B3	A1B3	A0B3			
4	A1B4	A0B4				
5	A0B5					

Figure 3.8: Summation of Partial Product

This new technique of computing the sticky bit is much more efficient in terms of both hardware complexity and speed in comparison to the computation of the sticky bit directly from input operands.

3.8 A New technique for Computing the Exponent

It was mentioned earlier that the exponent data path is much slower than the mantissa section. The main reason for low performance of the exponent data path is due to the need of three serial adders for computing the exponent. To improve the performance of the exponent computation a new technique has been developed that reduces the number of

serial adders from three to two. This new technique will be illustrated through the following examples.

In floating point multiplication the exponent of the two operands are added together to generate the exponent of the result. As the exponents are represented by biased code (Characteristic) a correction is necessary.

The characteristic of A is : $CA = EA + 127$

The characteristic of B is : $CB = CB + 127$

Adding the two characteristic gives: $CA + CB = EA + EB + 254$

The characteristic of the result should be: $C(A+B) = (EA + EB) + 127$

Thus, correct the exponent by subtracting the bias : $C(A+B) = CA+CB-127$

Let's call the corrected characteristic of the result CR

$$CR = CA + CR\_temp + V$$

where:

$CR\_temp = CB - bias$

V is the overflow bit from the mantissa

In the next first two examples, only the computation of CR_temp will be analyzed. CR_temp will be computed using 2's complement addition that is

$$CR\_temp = CB - Bias = CB + Bias\_bar + 1$$

Example 1:

$CB = 10000000$ $Bias = 01111111$

CB	10000000	
Bias_bar	10000000	
	00000001	
CR_temp	100000001	**00000001**

Note in 2's complement addition the most significant bit of the result indicates the sign bit. That is if the most significant bit (E) is 1 the sign of the result is positive otherwise it is negative. Therefore, the sign of the result for example 1 is positive.

Example 2:

CB = 11100000 Bias = 01111111

CB	11100000	
Bias_bar	10000000	
	00000001	
CR_temp	101100001	**01100001**

Note the final results of the above two examples can be simply obtained by inverting the most significant bit (MSB) of CB and adding a one to its least significant bit. The result obtained using this technique are shown on the right hand side of the final results written in bold form. The question is how to detect the sign of the result obtained using this new technique. The sign can be simply detected by looking at the most significant bit of CB, that is if MSB of CB is a one then the result is positive otherwise it is negative.

In the previous two examples the final results are positive, the next example will justify that the new technique of computing CR_temp also holds for negative final result.

Example 3:

CB = 00001000 Bias = 01111111

CB	00001000	
Bias_bar	10000000	
	00000001	
CR_temp	010001001	**10001001**
CR _tempbar	01110110	**01110110**
	00000001	**00000001**
	01110111	**01110111**

Since the most significant bit of CR_temp (E) in example 3 is zero, CR_temp must be complemented and then a one is added to its least significant bit (LSB) to obtain the correct result. Both final results obtained using 2's complement addition and the new technique are the same. Therefore the new technique holds for both E=0 and E=1.

In the next examples, it will be investigated whether the adjustment of CR_temp is necessary when E=0, by looking at the full computation of the exponent (that is computing CR). Since in the previous examples it was demonstrated that computing CR_temp using 2's complement addition gives the same result as the new technique, for the coming examples the new technique will be used to compute CR_temp.

Example 4:

CA = 10001100 CB = 00001000 V= 0

CR_temp	10001001
CA	10001100
CR'	100010101
V	00000000
CR	000010101

E1 = 0 E2 = 1 E3 = 0

where:

E1 is MSB of CB

E2 is MSB of CR'

E3 is MSB of CR

Since E1 is zero the intermediate result (CR_temp) is negative. Therefore CR' is the sum of a negative number (CR_temp) and a positive number (CA). In this case the MSB of CR' indicates the sign of CR'. Since E2 equals one for the above example the result is positive 21 as expected. The next example illustrates the condition when both E1 and E2 are zero.

Example 5:

CA = 00001111 CB = 00011000 V = 1

CR_temp	10011001
CA	00001111
CR'	010101000
V	00000001
CR	010101001

E1 = 0 E2 = 0 E3 = 0

Since E1 and E2 are both zero, the output CR is in two's complement form. Is it necessary to adjust CR to get the true result ? No, since the minimum biased exponent in IEEE

standard is zero, negative biased exponent indicates an underflow condition. Therefore, if CR is negative it not necessary to adjust the exponent rather it will be setted to underflow. In general, if both E1 and E2 are zero, they indicate an underflow condition independent of E3.

If E1 is one and at the same time either E2 or E3 are one, it indicates an overflow condition. If E1 equals one it means that the intermediate result (CR_temp) is positive. Therefore, the subsequent additions involve the addition of positive numbers and consequently the most significant bit of the result is an overflow bit, not a sign bit as shown in example 5. Table 3.2 shows how overflow and underflow conditions are detected using the three bits E1, E2 and E3. Figure 3.9 shows the circuit used to detect underflow and overflow conditions.

Example 6:

CA = 11111110 CB = 10000000 V = 1

CR_temp	00000001
CA	11111110
CR'	011111111
V	00000000
CR	100000000

E1 = 1 E2 = 0 E3 = 1

E1	E2	E3	U	V
0	0	X	1	0
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Table 3.2: Detection of Underflow and Overflow Conditions

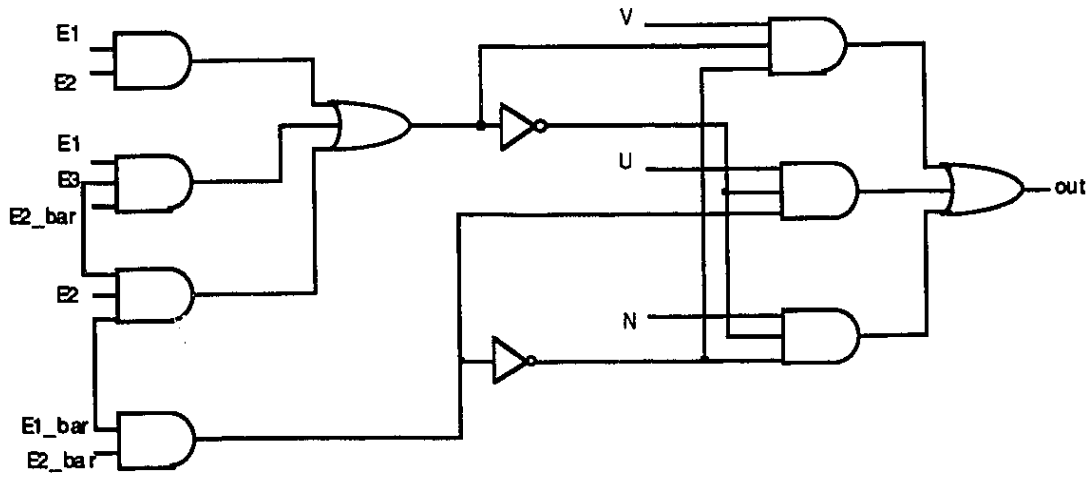


Figure 3.9: Underflow and Overflow Detector

Therefore as illustrated in the previous examples, the number of serial adders required in the exponent data path can be reduced from three to two by using the new technique. CR_temp + CA (CA + CB - Bias) can be computed using only one adder. That

is, CB - Bias is simply computed by inverting the MSB of CB and adding one to its LSB. The one added to the LSB of CB can be placed in the carry_in slot of the adder as shown in Figure 3.10. The new exponent data path is shown in Figure 3.10. From VHDL simulation result it was found that the new technique has a 50 percent higher performance than the normal way of computing the exponent. The simulation result of the exponent data path using the new technique is shown in Figure 3.11.

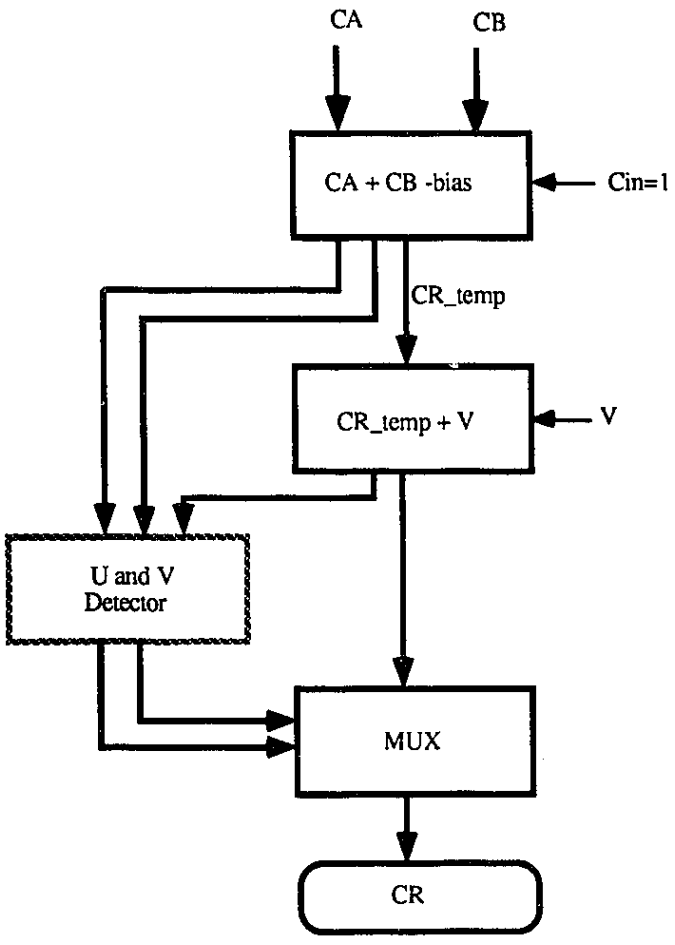


Figure 3.10: Fast and Efficient Exponent Data Path

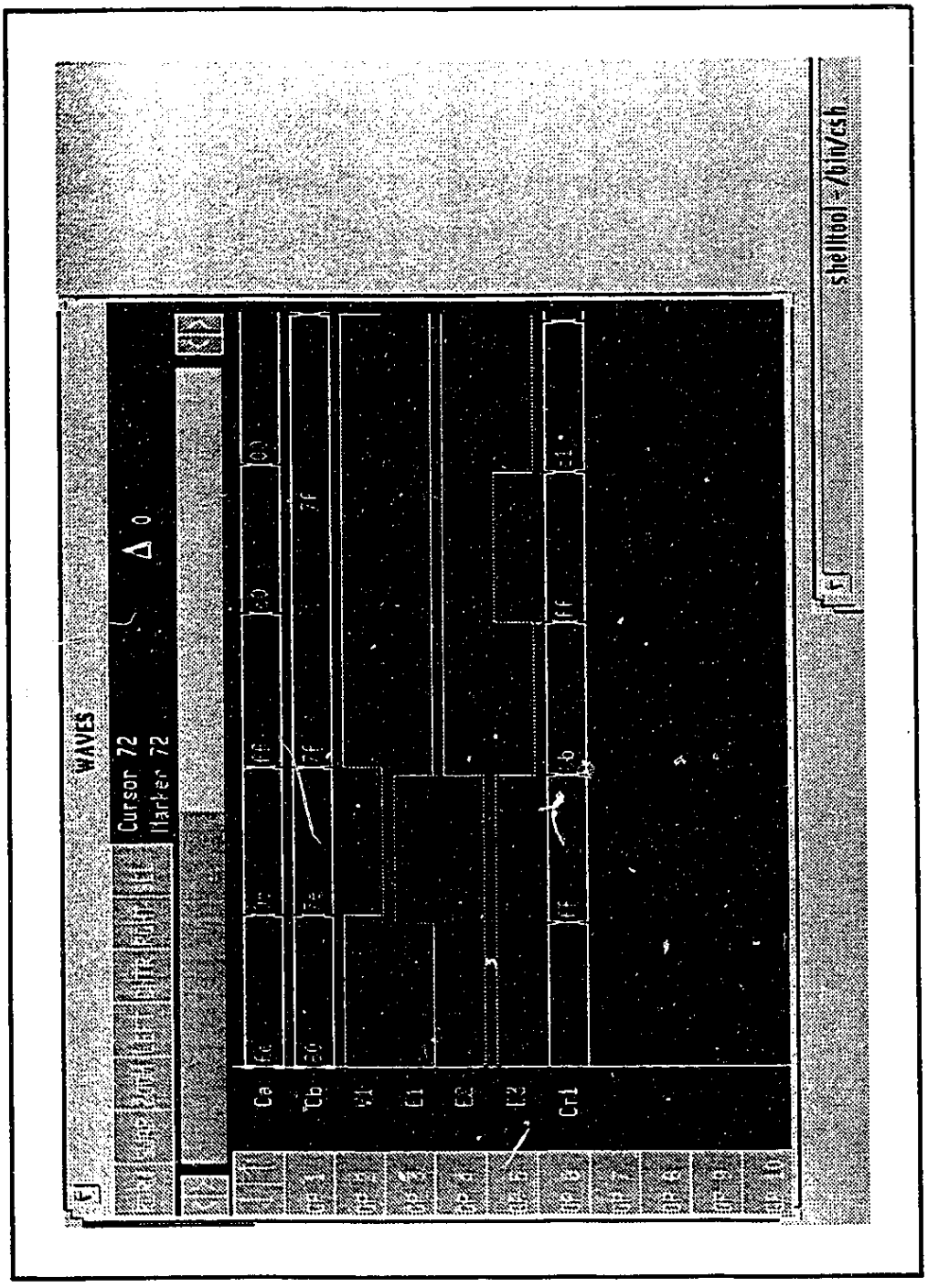


Figure 3.11: VHDL Waveforms for Exponent Data Path

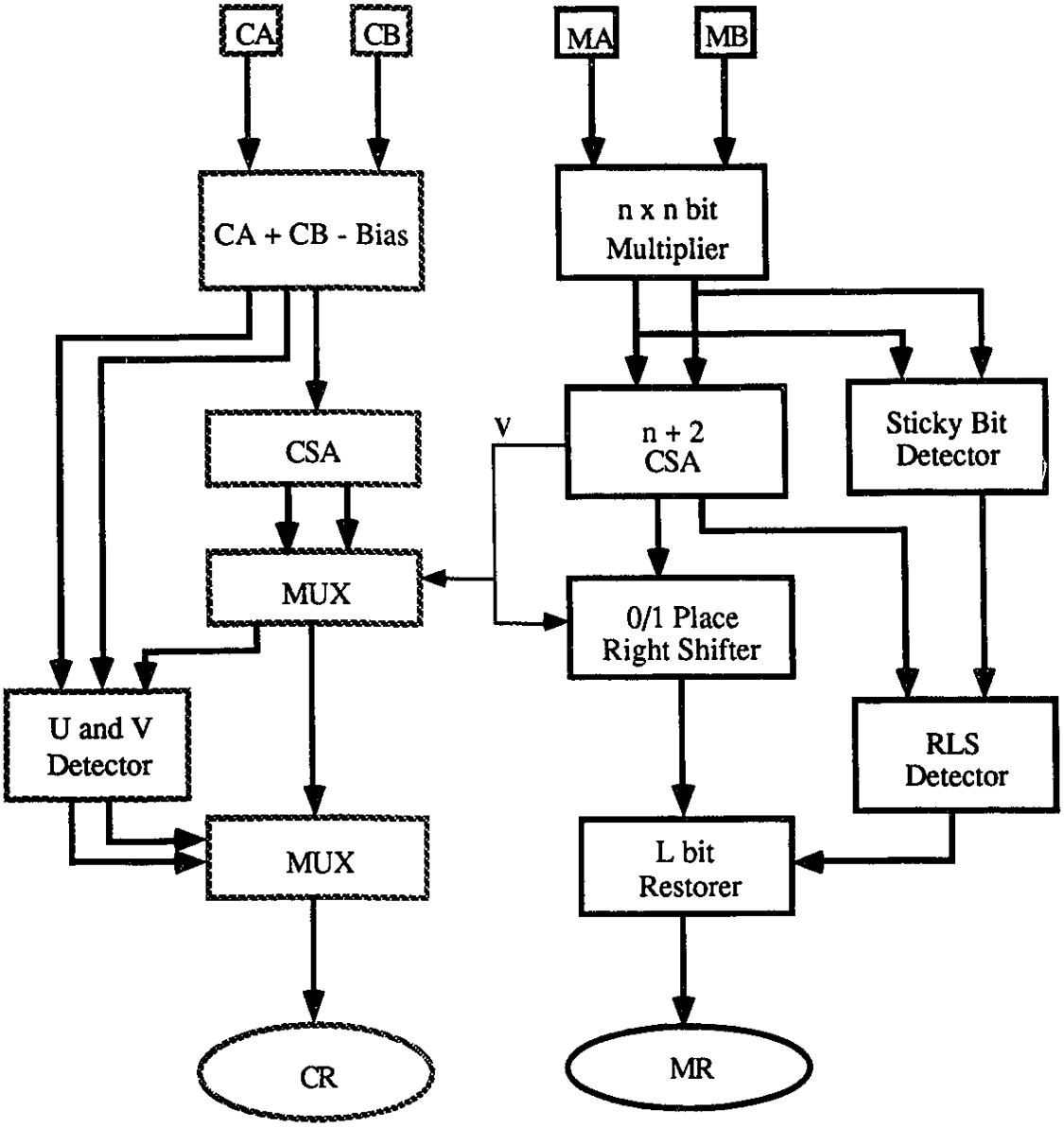


Figure 3.12: Improved Floating Point Multiplier Data Path

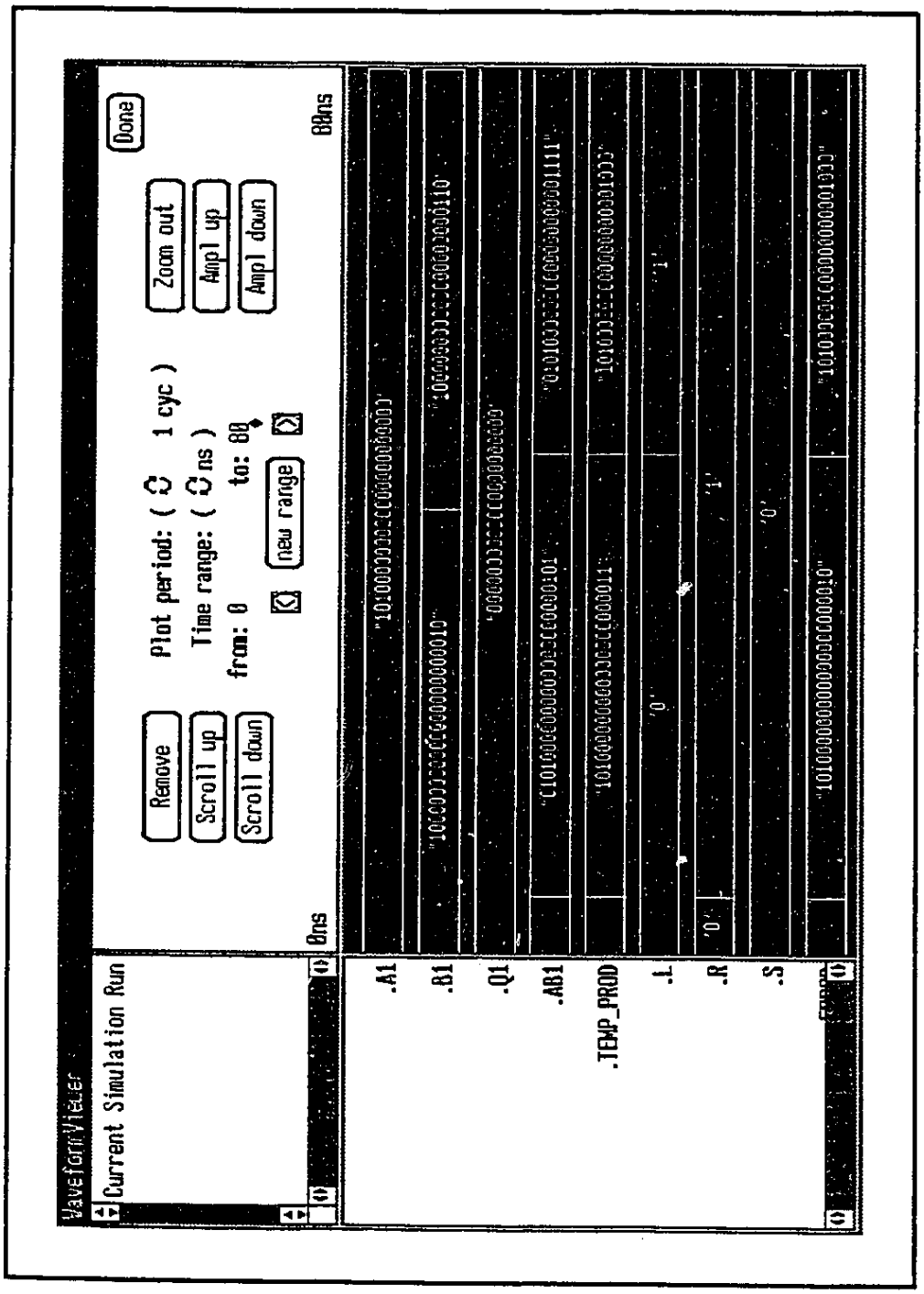


Figure 3.13: VHDL Waveforms for A Single Precision Floating Point Multiplier

The improved floating point multiplier algorithm, which incorporates the fast rounding technique and the new fast and efficient computation of the exponent and the sticky bit, is shown in Figure 3.12. A single precision floating point multiplier using the new algorithm has been successfully simulated in VHDL as shown in Figure 3.13.

From Figure 3.12, it can be seen that many steps are required for implementing floating point operations. However, each floating point operation eventually reduces to an integer operation. Thus increasing the speed of integer operations will also lead to faster floating point operation. For this reason, in the next chapter, a comparison of several parallel multiplier architectures and adder structures, including two new parallel multiplier architectures, in terms of their speed and complexity will be made. In addition the architecture suitable for the new floating point multiplier algorithm will be selected.

CHAPTER 4

Multiplier Architectures

4.1 Introduction

Webster's dictionary defines multiplication as "a mathematical operation that at its simplest is an abbreviated process of adding an integer to itself a specified number of times". A number (**multiplicand**) is added to itself a number of times as specified by another number (**multiplier**) to form a result (**product**). In elementary school, students learn to multiply by placing the multiplicand on top of the multiplier. The multiplicand is then multiplied by each digit of the multiplier beginning with the rightmost, Least Significant Digit (LSD). Intermediate results (**partial products**) are placed one atop the other, offset by one digit to align digits of the same weight. The final product is determined by summation of all the partial products. Although most people think of multiplication only in base 10, this technique applies equally to any base, including binary. Figure 4.1 shows a

data flow for the basic multiplication just described. Each black dot represents a single digit.

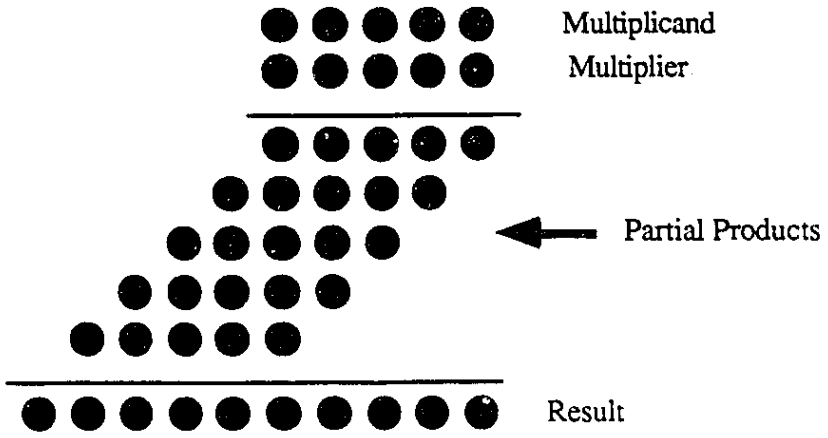


Figure 4.1: Basic Multiplication Data Flow

4.2 Binary Multiplication

In the binary number system the digits, called bits, are limited to the set [0,1]. The result of multiplying any binary number by a single binary bit is either 0, or the original number. This makes forming the intermediate partial-products simple and efficient. Summing these partial-products is the time consuming task for binary multipliers. One logical approach is to form the partial products one at a time and sum them as they are generated. Often implemented by software on processors that do not have a hardware multiplier, this technique works fine, but is slow because at least one machine cycle is required to sum each additional partial product. For application where this approach does not provide enough performance multipliers can be implemented directly in hardware.

Direct hardware implementations of shift and add multipliers can increase performance over software synthesis, but are still quite slow. The reason is that, as each additional partial-product is summed, a carry must propagated from the least significant bit to the most significant bit. This carry propagation is time consuming, and must be repeated for each partial product to be summed.

One method to increase multiplier performance is by using encoding techniques to reduce the number of partial products to be summed. Such a technique was first proposed by Booth [9]. The original 'Booth's algorithm' skips over contiguous strings of 1's by using the property that: $2^n + 2^{(n-1)} + 2^{(n-2)} + \ldots + 2^{(n-m)} = 2^{(n+1)} - 2^{(n-m)}$. Although Booth's algorithm produces at most N/2 encoded partial products from an N bit operand, the number of partial products produced varies. This has caused designers to use modified version of Booth's algorithm for hardware multipliers. Modified 2 bit Booth encoding halves the number of partial products to be summed.

To achieve even a higher performance, advanced hardware multiplier architectures use faster and more efficient methods for summing the partial-products. Most designs increase performance by eliminating the time consuming carry propagate additions. To accomplish this, they sum the partial-products in a redundant number representation. The advantage of a redundant representation is that two numbers, or partial products, can be added together without propagating a carry across the entire width of the number. Many redundant number representations are possible. One commonly used representation is known as carry-save form. In this redundant representation two bits, carry and sum, are used to represent each bit position. When two numbers in carry-save form are added together any carries that result are never propagated more than one bit position. This makes adding two numbers in carry-save form much faster than adding two normal binary numbers where a carry may propagate. One common method that has been developed for summing rows of partial products using carry-save representation is the array multiplier.

4.3 Array Multiplier

Conventional linear array multipliers consist of rows of carry-save adders(CSA). An 8 by 8 bit linear array multiplier can be seen in Figure 4.2. In a linear array multiplier, as the data propagates down through the array, each row of CSA's adds one additional partial product to the partial sum. Since the intermediate partial sum is kept in redundant, carry-save form there is no carry propagation. This means that the delay of an array multiplier is only dependent upon the depth of the array, and is independent of partial-product width. Linear array multipliers are also regular, consisting of replicated rows of CSA's. Their performance and regular structure are most often used in array multipliers for VLSI math co-processors and special purpose DSP chips.

One of the problems with full linear array multipliers is that they are very large. As operand size increases, linear arrays grow in size at a rate equal to the square of the operand size. The other problem of linear multipliers is that their performance (proportional to N) is not time optimal. Due to these facts, linear array multipliers are restricted for small operand sizes, or on special purpose math chips where a major portion of the silicon area can be assigned to the multiplier array. Another multiplier architecture which has a higher performance than the linear array multiplier and regular in structure will be described in the next section.

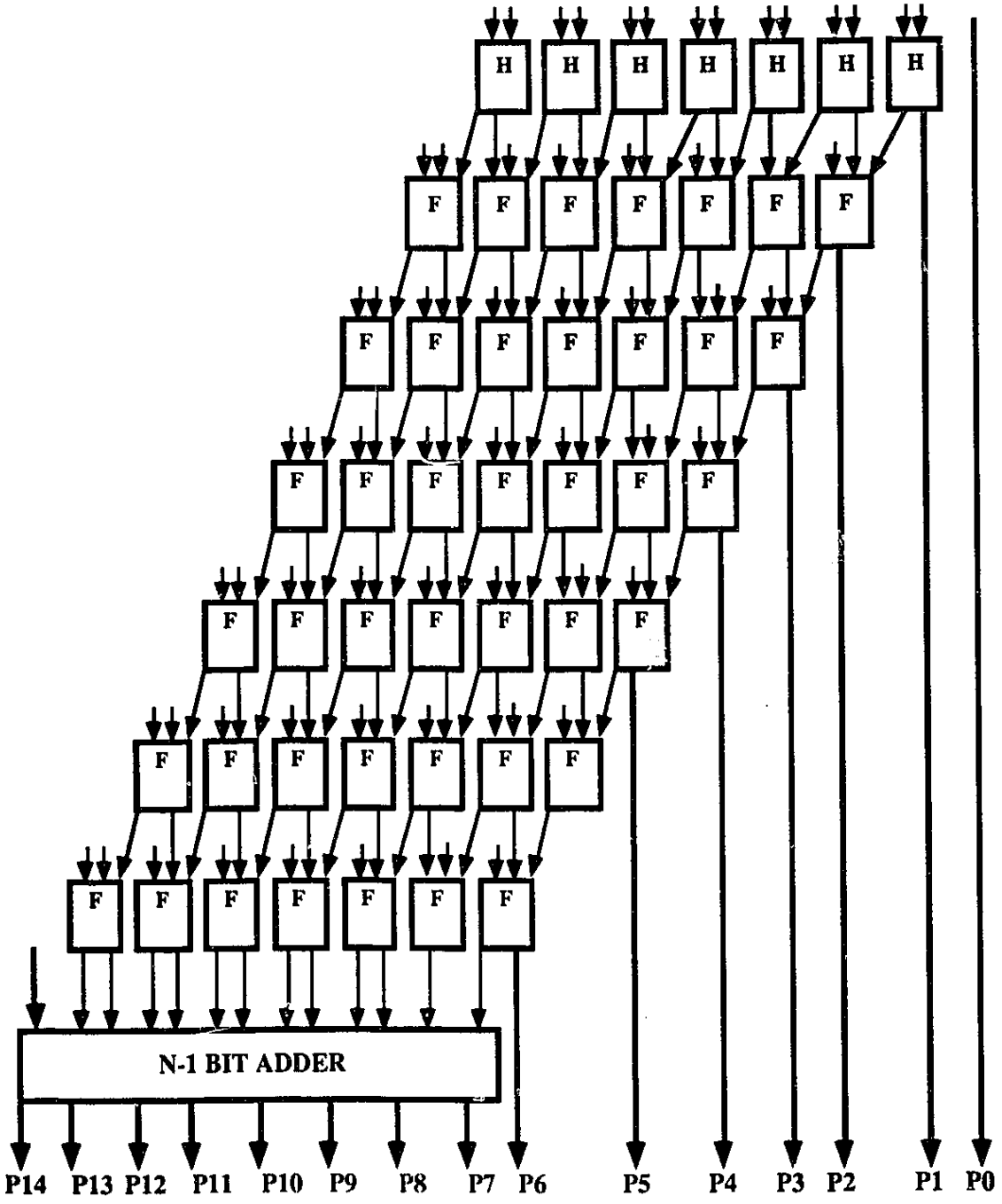


Figure 4.2: 8x8 Bit Linear Array Multiplier

4.4 Five Bit Counter Multiplier

The five-counter parallel iterative multiplier was proposed by Nakamura [10]. The multiplier is composed of five-counter cells and it was claimed [10] the multiplication speed is twice as fast as the conventional array multiplier with approximately the same complexity in hardware. The five-input counter, or simply the 5 counter is expressed by the diagram shown in Figure 4.3. The five inputs to the 5-counter are the three inputs through input lines in_0, in_{-1},in_{-2} and the two partial-products of aibj and ajbi which are shown at the centre of the diagram. The 5-counter counts the number of ones on these 5 inputs and produces a binary number of the count on the three output lines out_0, out_1 and out_2 which correspond to the 2^0, 2^1 and 2^2 positions respectively, i.e. the function of a five counter cell is:

$$2^2(out_2) + 2^1(out_1) + 2^0(out_0) = in_0 + in_{-1} + in_{-2} + aibj + ajbi$$

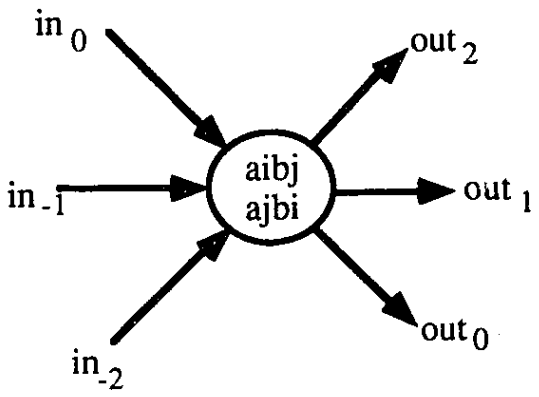


Figure 4.3: A 5-Bit Counter Cell

This five-counter multiplier requires n five-counter cell delays plus one binary adder delay for n bit by n bit multiplication. Another multiplier architecture that possesses almost the same structure as the five-counter multiplier, but with fewer cells and slightly higher speed was proposed by Wang. This architecture uses a two-bit full-adder as the basic building cell. A two-bit full-adder is a device which takes two 2-bit input numbers (m1m0, n1n0) and a carry-in Ci and produces a three bit output. The relationship of the input and outpu number is shown in Figure 4.4.

$$m0 + n0 + ci + 2(m1 + n1) = 4c0 + 2s1 + s0$$

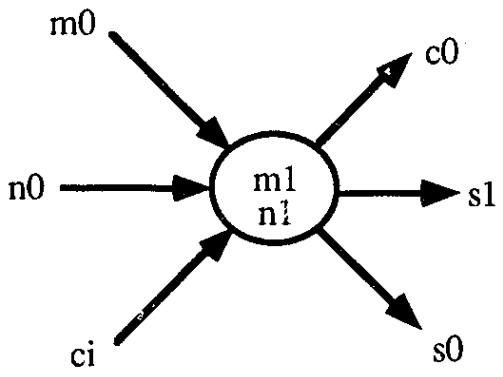


Figure 4.4: A two-bit full-adder cell

One may see that Figure 4.3 is similar to Figure 4.4, because both the 2-bit full-adder and the 5-counter have five input bits and three output bits. The difference relies on the weight of the input bits. All input bits for the 5-counter possess same weights. While input bits of 2-bit full-adder possess different weights. The weight of each input inside the circle possess a weight twice as that of each input bit outside the circle. The logical expressions for s0, s1 and c0 are given by

$$s0 = m0(n0ci + n0ci) + m0(noci + noci)$$

$$s1 = (m1n1 + m1n1)[m0n0 + (m0 + n0)ci] + (m1n1 + m1n1)[m0n0 + (m0 + n0)ci]$$

$$c0 = m1n1 + (m1 + n1)[m0n0 + (m0 + n0)ci]$$

Now we consider the speed of these two types of multipliers. As mentioned earlier, the overall delay for the five bit counter is n five counter delays plus one binary adder delay. While the overall delay for the two-bit full-adder multiplier is (n-1) two-bit full-adder delays plus one binary full-adder delay. It is reasonable to assume that the delay of a two-bit full-adder is the same as the delay of a five-counter. Therefore, the two-bit full-adder possesses a slightly higher speed than the five bit counter multiplier.

In this work, two 8 by 8 bit two_bit full-adder multiplier have been implemented in 0.8µm BICMOS technology. These two multiplier will be referred to as Type_1 and Type_2 multipliers. Type_1 and Type_2 multipliers were implemented using Figure 4.5 and Figure 4.6 as their basic building cell (2-bit full-adder cell). The structural organization of an 8 by 8 bit two-bit full-adder multiplier is shown in Figure 4.7. A comparison of the two multipliers in terms of speed and area is shown in Table 4.1. Implementation of a two-bit full-adder multiplier using type_1 has a higher performance than type_2. However, the area of type_1 is about twice the area of type_2 as shown in table 4.1.

The VHDL simulation results and the layouts of the multipliers are shown in Figure 4.8 and Figure 4.9 respectively. Even though the 2-bit full adder multiplier gives a better performance than linear arrays, it does not give an optimal performance.

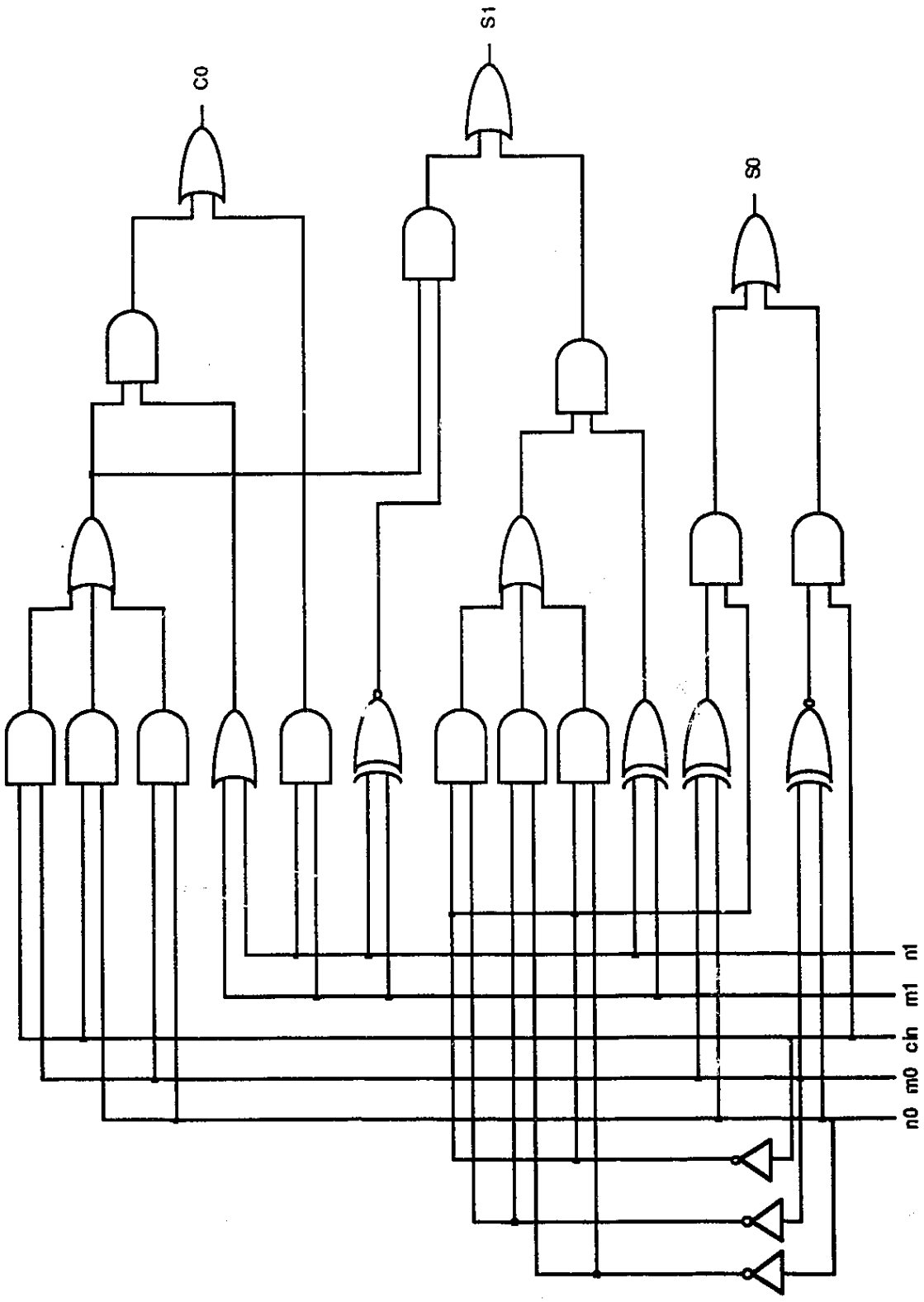


Figure 4.5: Type_1 two-bit full_adder

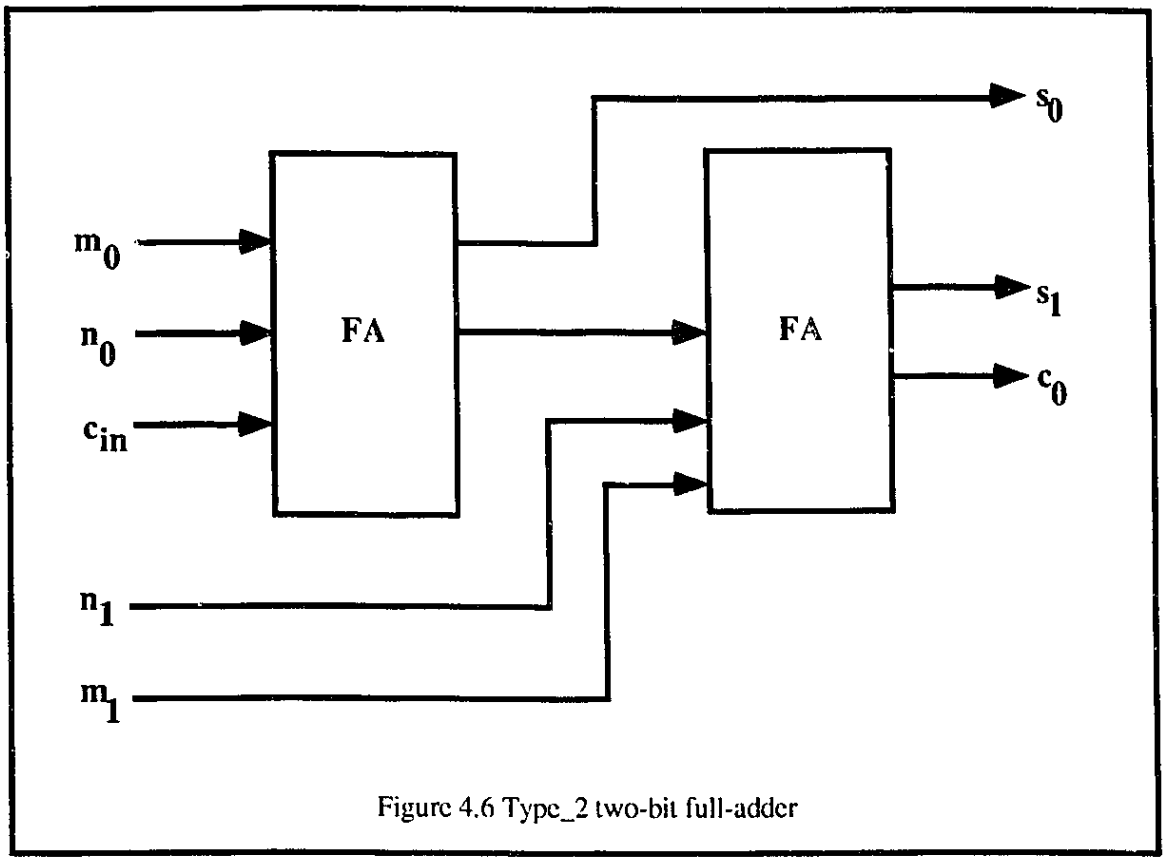


Figure 4.6 Type_2 two-bit full-adder

Type	Latency	Area
Type_1	20 ns	1.3 x 1.1 mm
Type_2	24 ns	0.890 x 0.776 mm

Table 4.1: Comparisons of two two-bit full-adder multipliers

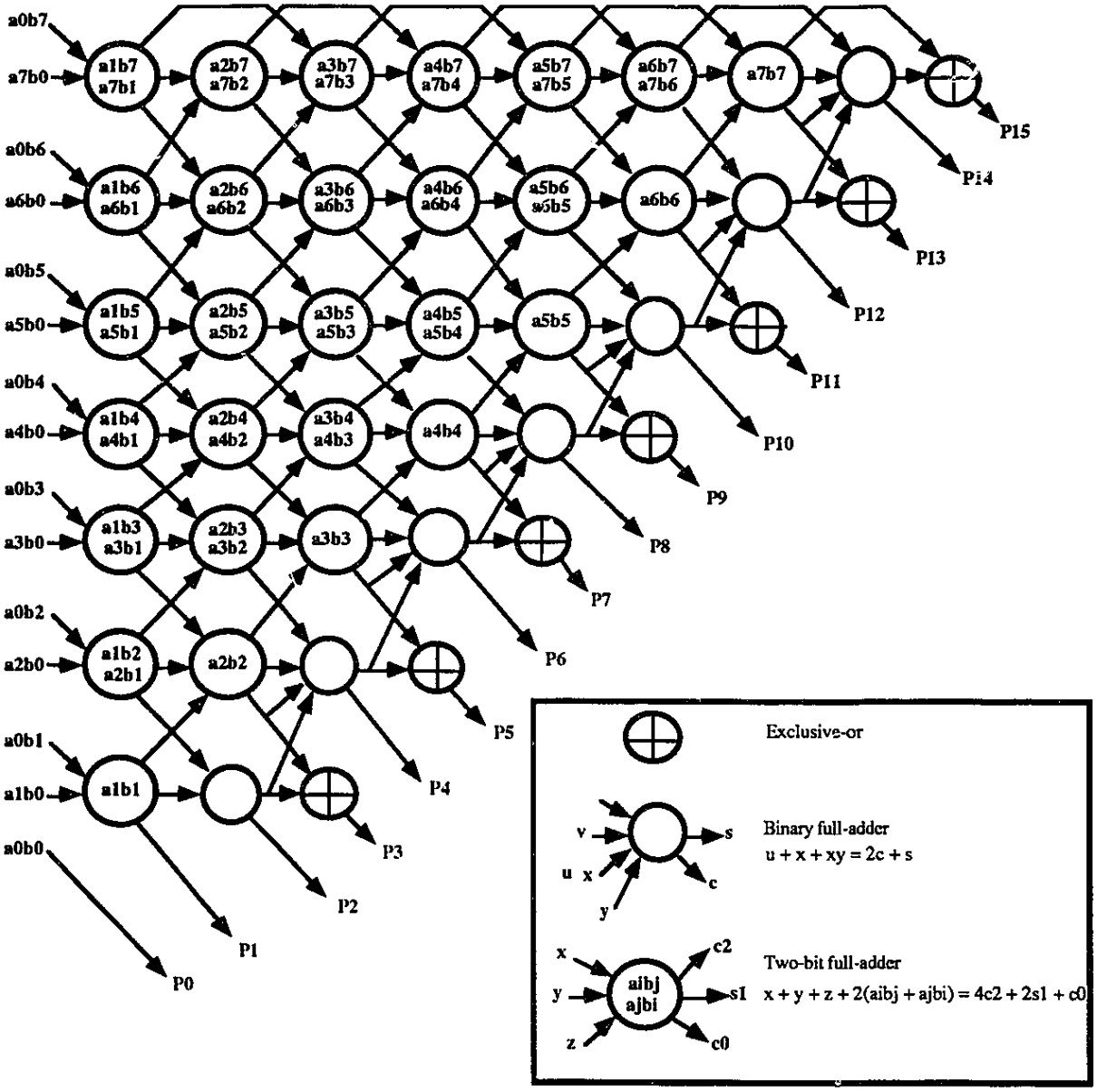


Figure 4.7: 8x8 Bit Two-Bit Full-Adder Multiplier

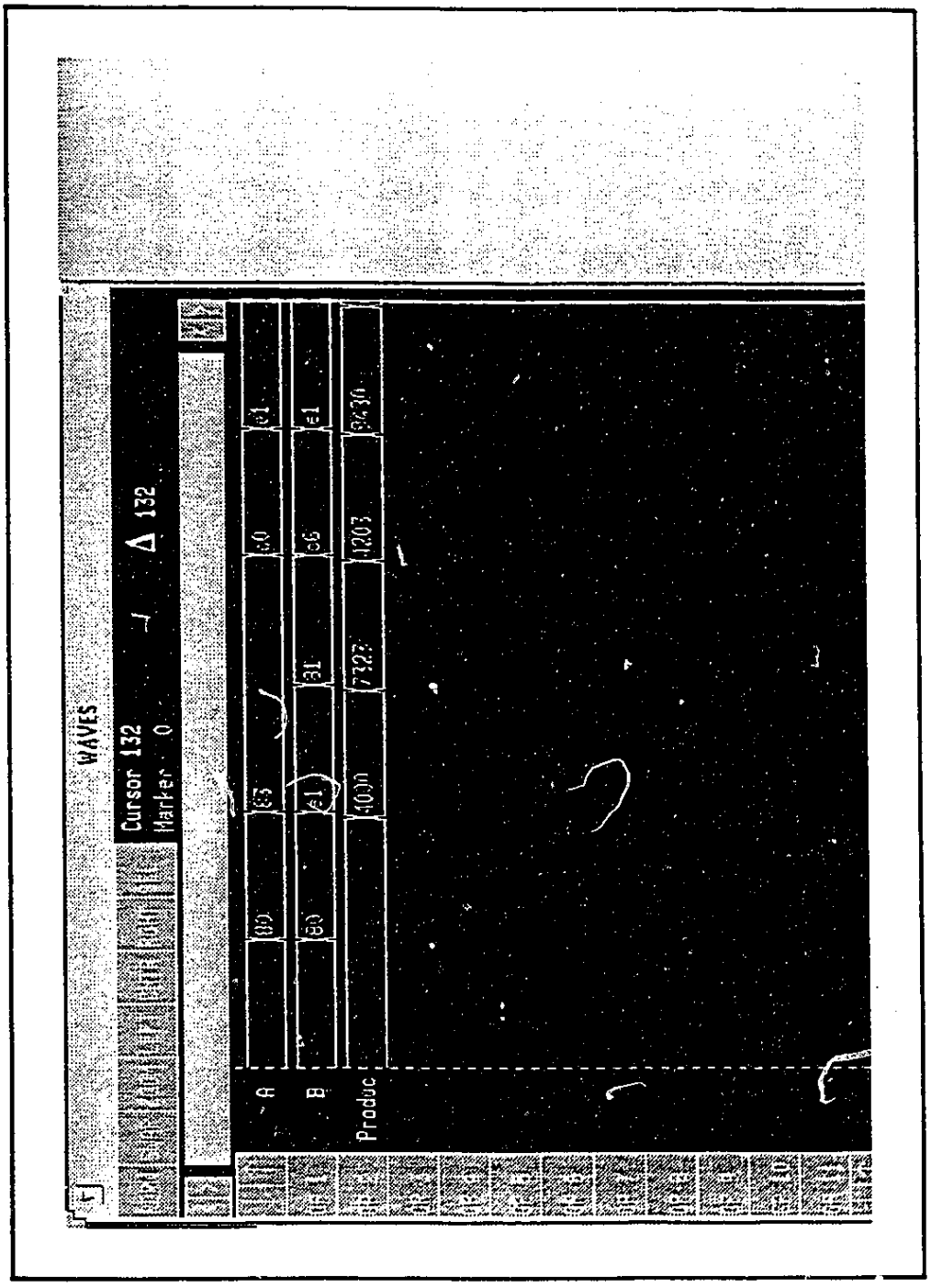


Figure 4.8a: VHDL Waveforms for Type_1 Two-Bit Full-Adder Multiplier

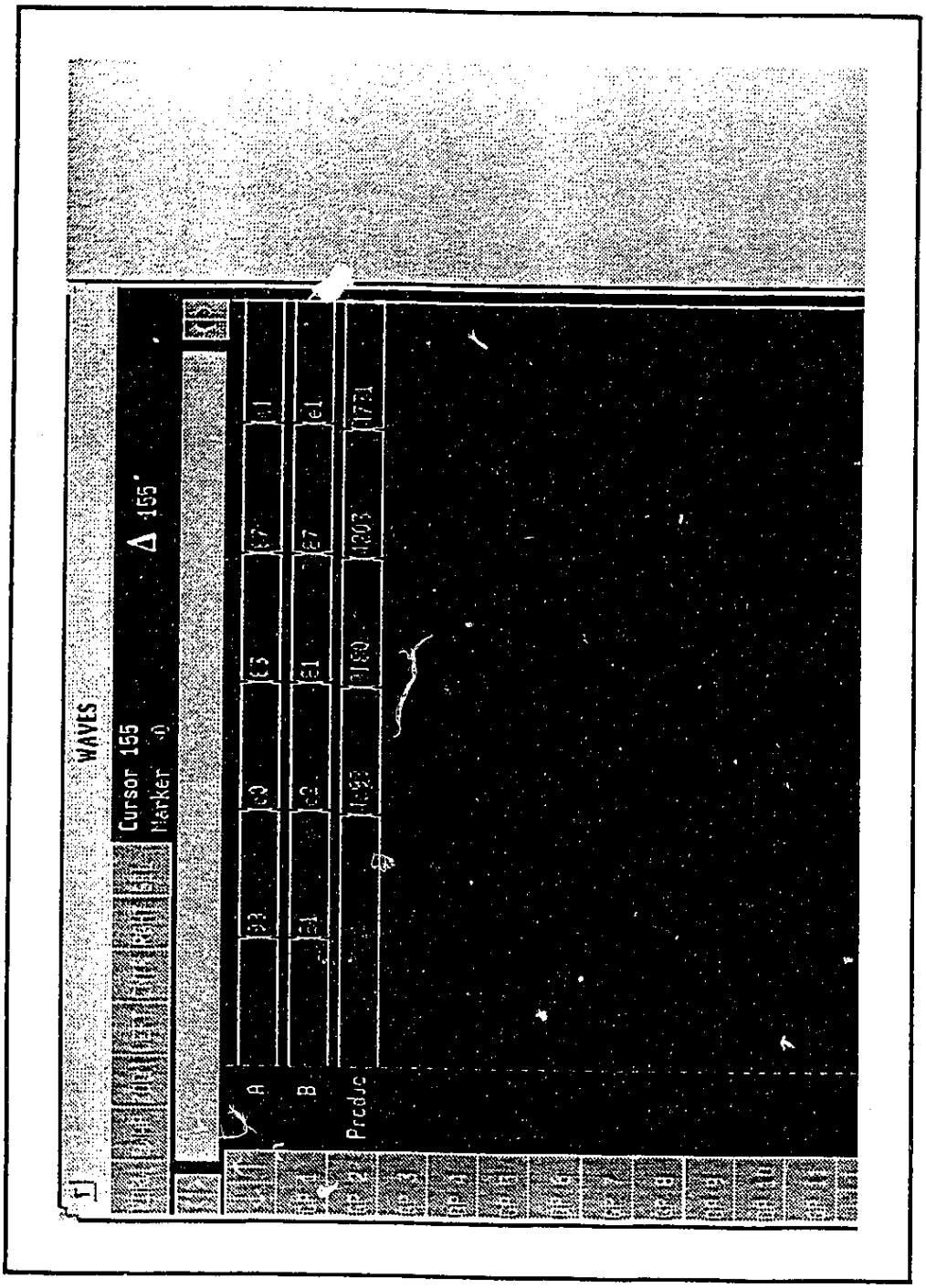


Figure 4.8b: VHDL Waveforms for Type_2 Two-Bit Full-Adder Multiplier

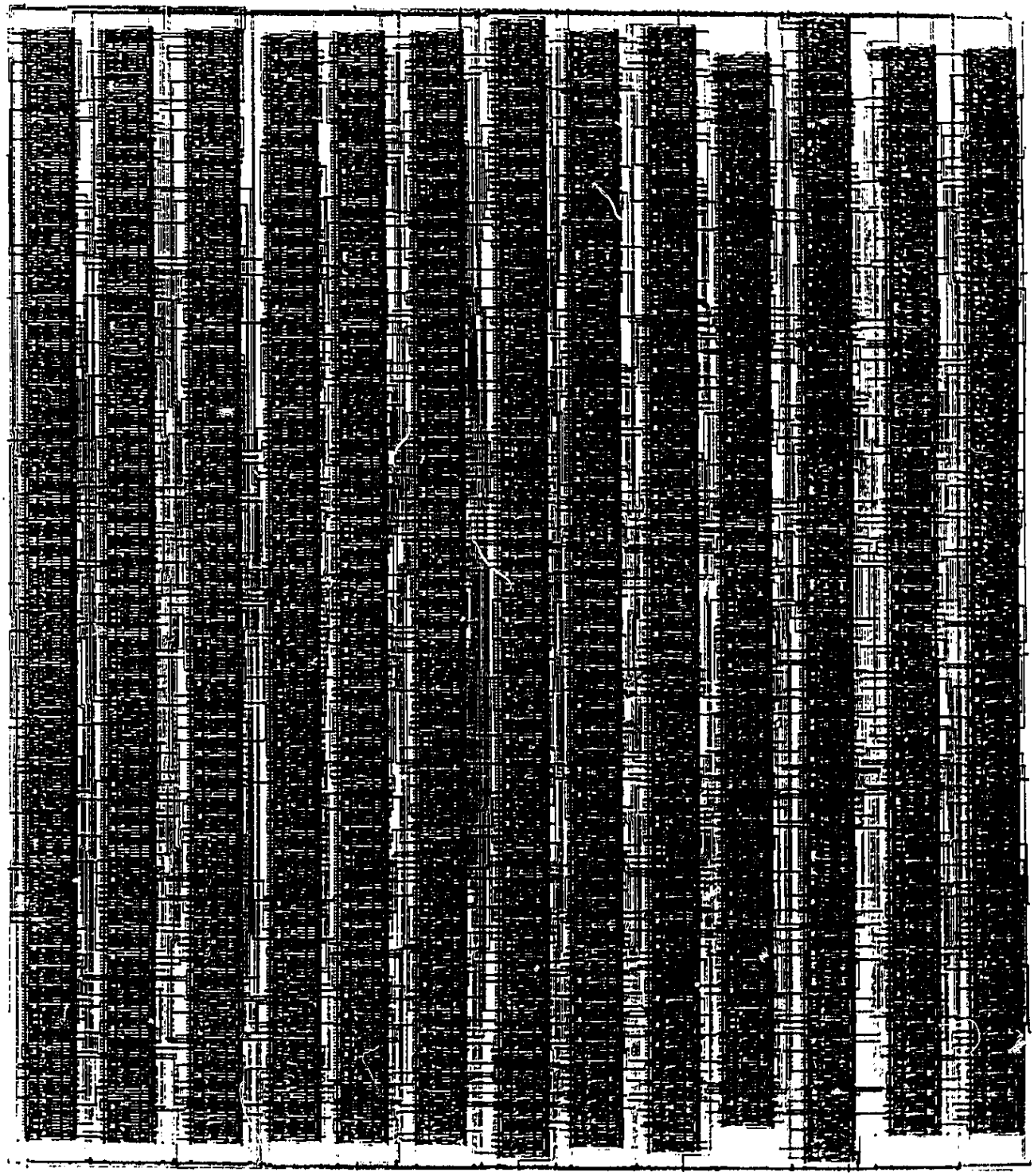


Figure 4.9a: Layout for Type_1 Two-Bit Full-Adder Multiplier

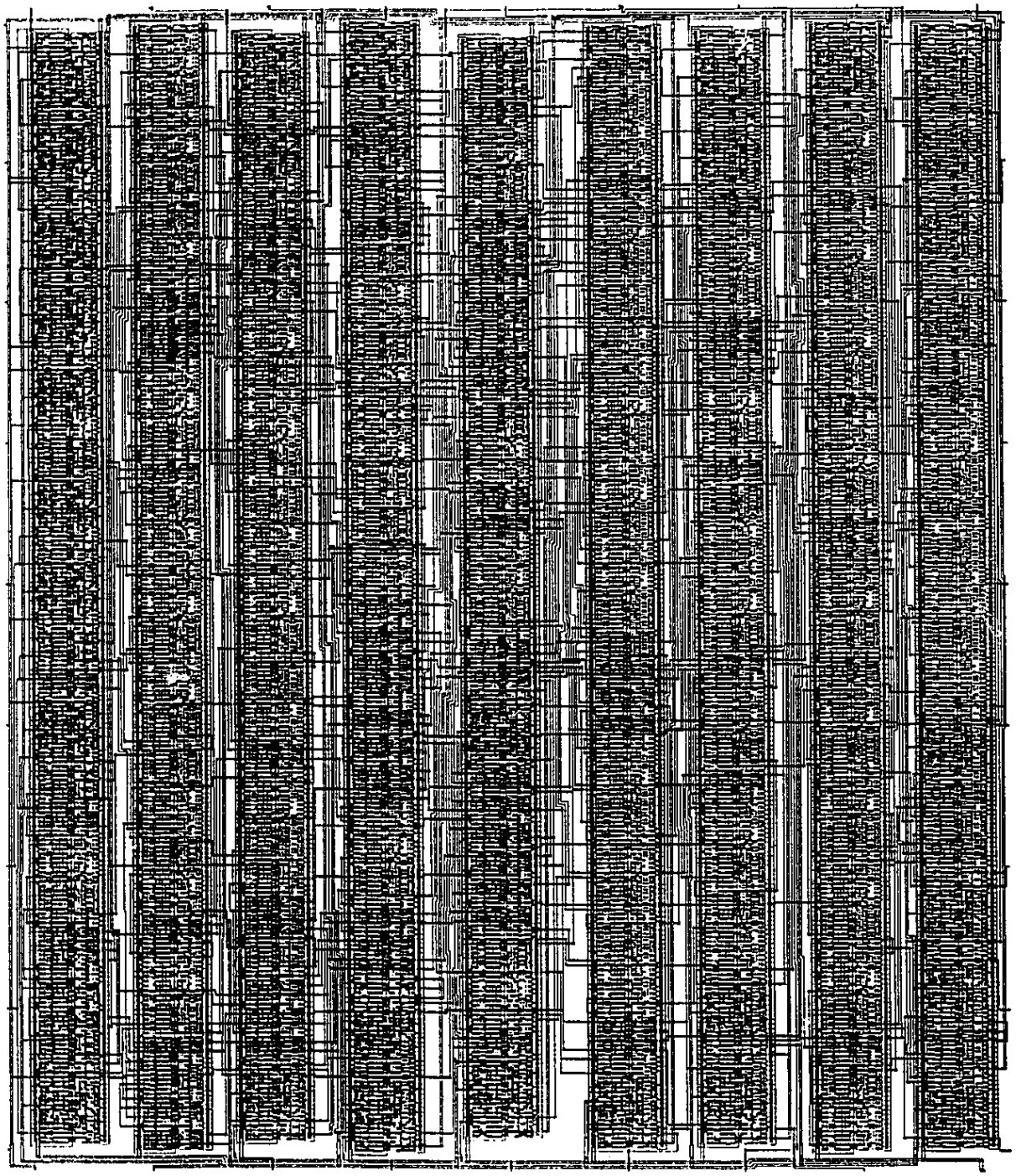


Figure 4.9b: Layout for Type_2 Two-Bit Full-Adder Multiplier

To achieve an additional increase in performance one obvious step is to make the CSA's faster. Another powerful technique for increasing performance is to reduce the number of series additions required to sum the partial-products by using tree structures.

4.5 Wallace and Dadda Multipliers

The Wallace multiplier (tree structure) is extremely fast for summing partial products. As previously discussed, linear arrays require order N stages to reduce N partial products. In contrast, by performing the additions in parallel tree structures only order log(N) stages are required to reduce N partial products.

In Wallace's scheme [7], groups of three partial product vectors are applied as inputs to banks of (3,2) adders, each such triplet of vectors producing two vectors at the outputs. Triplets of these output vectors are then applied as inputs to the next level of adders, and so forth. The number of levels required in the Wallace structure is given approximately by:

$$K = \frac{\log_2(n-1)}{\log_2(3) - 1}$$

This follows from the fact that V_0, the number of output vectors from a given level, is related to V_I, the number of input vectors to that level, by

$$V_0 = \left\lceil 2/3 \times V_I \right\rceil$$

which leads to

$$\left\lceil \left\lceil \left\lceil 2/3 \times n \right\rceil \times 2/3 \right\rceil \times 2/3 \right\rceil \times \bullet\bullet\bullet \times 2/3 \Big\rceil = 2$$

An improvement on the Wallace structure has been made by L. Dadda [11], who has devised an arrangement that uses fewer adders, although the same number of levels are required. Dadda points out that since the last level produces two vectors (the input to the carry propagate adder) the preceding level had at most 3, its predecessor at most 4, and so on the series 2, 3, 4, 6, 9, 13, 19, 28 and so on is formed by the relation

$$T_{j+1} = \left\lceil 2/3 \times T_j \right\rceil$$

Where T_j is the jth term in the sequence. To minimize the number of adder elements in the total structure, we start by using the first level only enough full adders and half adders to reduce the initial n-row matrix to one having a number of rows equal to one of the terms in the series. Then at each succeeding level we use whatever number of adders we need to form the matrices having number of rows equal to the successively smaller number in the series. To illustrate the reduction scheme of dadda, an 8x8 bit multiplier is taken as an example as shown in Figure 4.10. The architecture of this multiplier is also shown in Figure 4.11.

Although trees such as dadda multiplier are faster than linear arrays the wiring required to gather bits of the same weight (as shown in Figure 4.11) makes trees even larger than linear arrays. The additional wiring required of full_trees over linear arrays has caused designers to look at permutations of the basic tree structure to ease the routing [12]. Unbalanced or modified trees make a compromise between conventional full_arrays and full_tree structures. They reduce the routing required of full trees, while slightly increasing the serialization of the partial product summation. Regularity is an important issue in VLSI designs. Regular structure tends to increase performance, reduce the risk of mistakes, and reduce layout time. The irregular nature of most makes them notoriously difficult to design and layout. Module generators can be used to automate the routing process, but the resulting structure requires significant area [13].

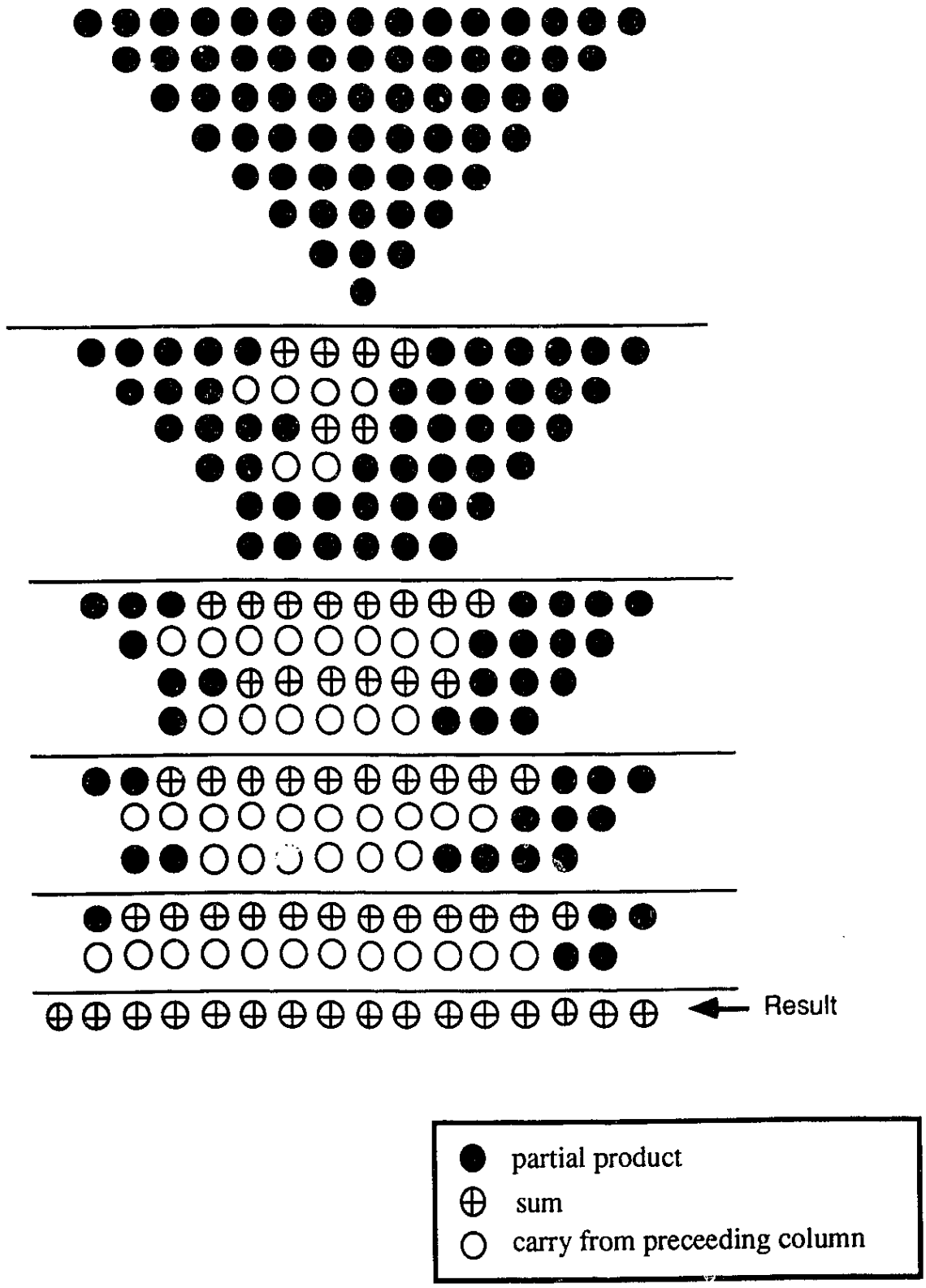


Figure 4.10: Dadda Algorithm Reduction Scheme for an 8x8 Bit Multiplier

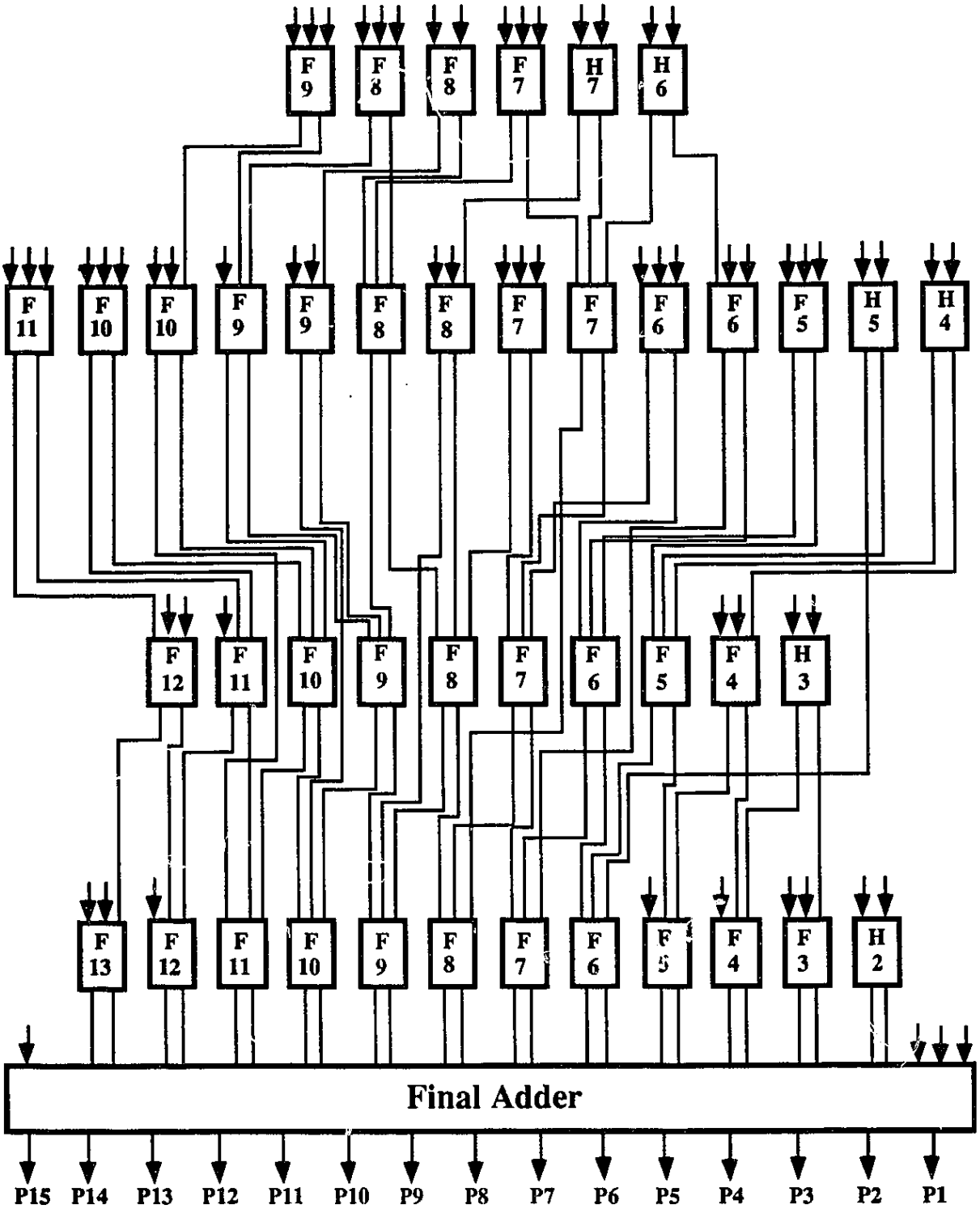


Figure 4.11: Dadda Architecture for an 8x8 Bit Multiplier

To produce a more regular structure, multipliers based upon redundant representation other than carry-save form have been presented [14]. In an attempt to reduce both the size and complexity of the wiring, one architecture uses radix 4 redundant form to produce a binary like tree, then encodes the signals using multiple-valued logic to reduce wire interconnection. The most popular architecture such as DEC MultiTitan Multiplier [15] uses high order counters such as (4:2) compressor and (7:3) counter rather than (3:2) counter to reduce partial products more rapidly and reduce the number of wirings. These architectures will be discussed in the next sections.

4.6 (4:2) Compressor

In designing a (4:2) compressor, a problem arises since a sum output with a weight of one and a carry output with a weight of two are not enough to convey the maximum possible count of four. The problem is circumvented by generating an intermediate carry output which is fed into the next block in the adder array. Thus a (4:2) compressor is effectively a (5:3) counter. The corresponding (4:2) adder truth table is shown in table 4.2. Note in table 4.2 n indicates the number of inputs which equals to 1 and the * indicates either C or Cout may be one if two or three inputs equal to 1 but not both. The (4:2) compressor has the same logical function as that of a carry-save adder constructed by two serial full-adders as shown in Figure 4.12b. This configuration is not time optimal in comparison to the circuit implemented directly from the truth table shown in Figure 4.12c. A 32 by 32 bit CMOS multiplier has been designed using (4:2) adder block [16]. In the (4:2) tree, for every four input taken in at one level, two outputs are produced at the next level.

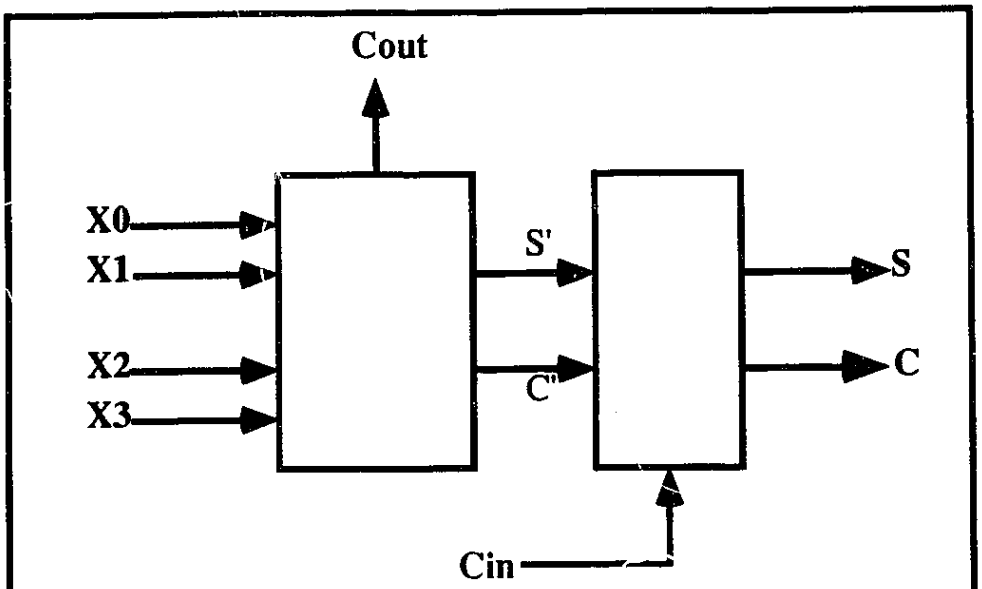


Figure 4.12a (4:2) Compressor

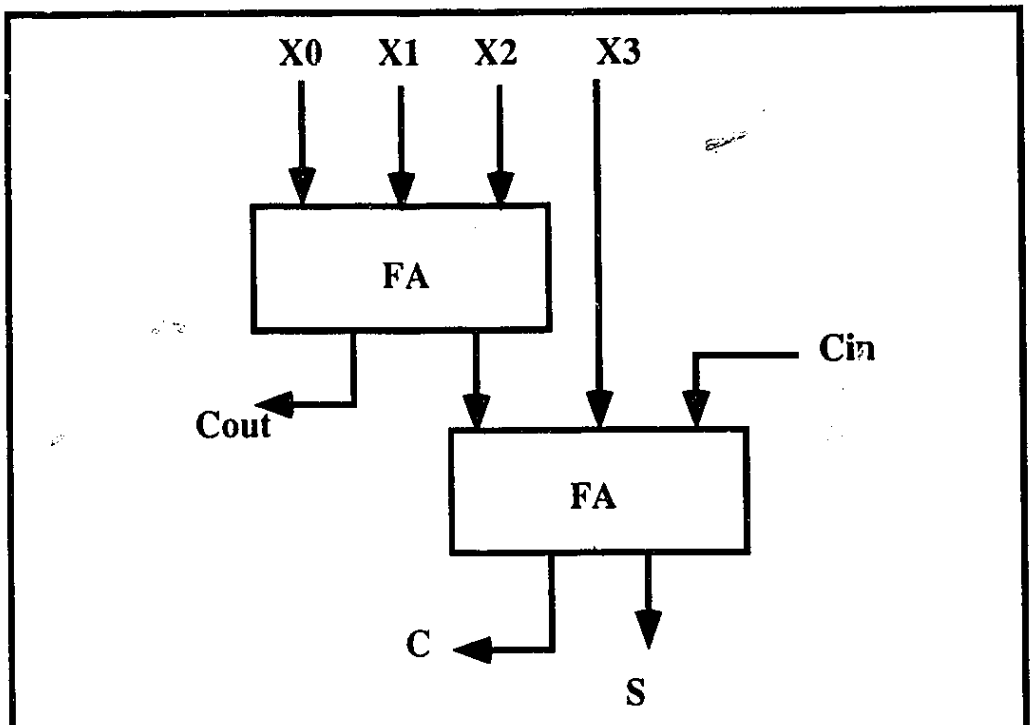


Figure 4.12b (3:2) implementation of (4:2) compressor

n	Cin	Cout	C	S
0	0	0	0	0
1	0	0	0	1
2	0	*	*	0
3	0	1	0	1
4	0	1	1	0
0	1	0	0	1
1	1	0	1	0
2	1	*	*	1
3	1	1	1	0
4	1	1	1	1

Table 4.2: Truth table for the (4:2) Compressor

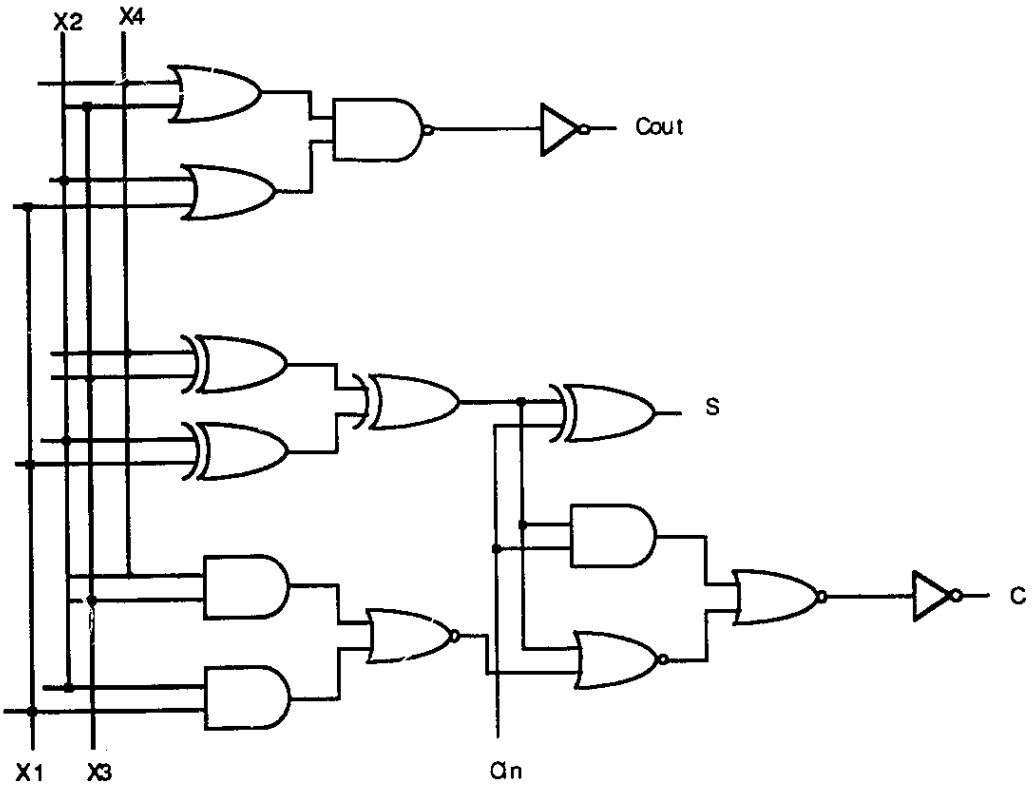


Figure 4.12c: Circuit for the (4:2) Compressor

Since each level of the (4:2) tree reduces two redundant numbers to one redundant number, the (4:2) tree can also be viewed as a redundant binary tree, and, as such, is better suited for VLSI implementation. However, the 4:2 tree has the following drawbacks. First, it has a higher delay than the Dadda structure if the operands are not multiples of two. Secondly, the gate count increases by more than twenty five percent if the 4:2 tree is implemented using Figure 4.12c rather than Figure 4.12b. Thirdly, it has longer interconnection wiring than the Dadda structure if the 4:2 adder is used for a full-tree implementation, as shown in Figure 4.13.

4.7 (7:3) Counter

There are several methods of implementing high input counters Foster and Stocken [17] have described a method for implementing counters with a network of full-adders. As shown in [17] at most N full-adders are required to implement an N-input counter. Swartzlander [18] introduced an alternative method to design a counter with 2k+1 inputs using two k input counters and (1+log2(k)) stage ripple carry adder. Using this philosophy a (7:3) counter was designed using two (3:2) counters and a stage ripple carry adder (which is equivalent to two (3:2) counters) as shown in Figure 4.14a. Figure 4.14b shows an alternative logic diagram of the (7:3) counter proposed by Mayur.M [19]. The seven inputs are divided into groups (x0, x1, x2, x3) and (x4, x5, x6). Internal "A" is the carry for two, three, or four ones in the first group, while internal " B" is the carry for two or three ones in the second group. Finally, internal signal " C " is generated by ORing the carry for four ones in the first group. A, B and C are combined using a conventional full_adder to get outputs C1 and C2, with weights of two and four respectively. The circuit shown in Figure 4.14a has six exclusive-or gate delays. Assuming an exclusive-or gate has twice the delay of a normal gate, and assuming that complex and-and-nor or or-or-nand gates have 1.5 times the delay of a normal gate, the circuit in Figure 4.14b has four

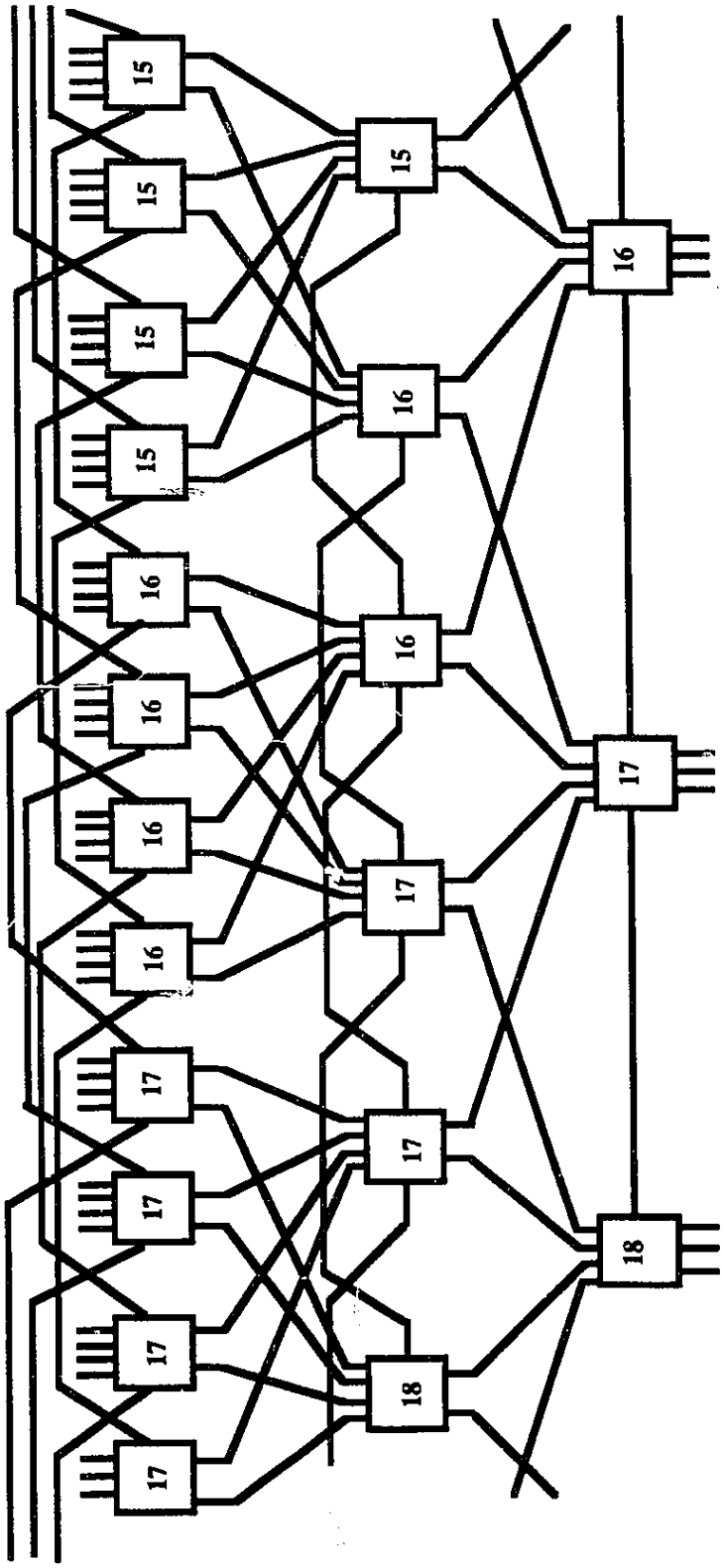


Figure 4.13: Typical (4:2) Compressor Multiplier Structure

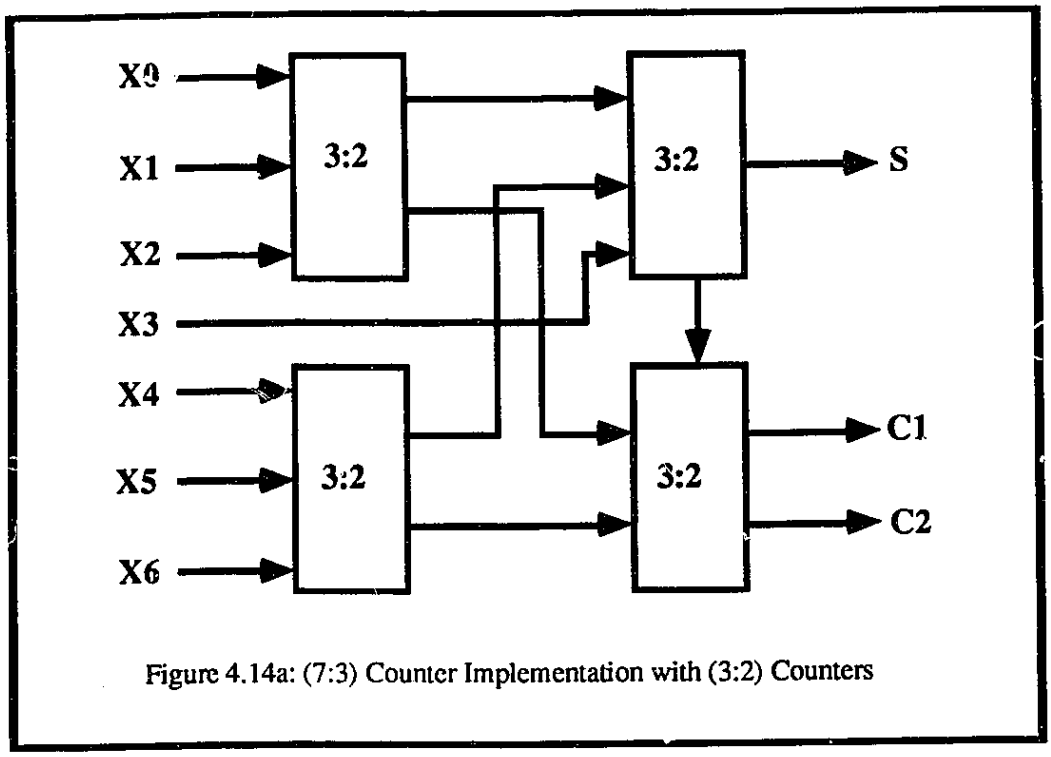


Figure 4.14a: (7:3) Counter Implementation with (3:2) Counters

exclusive-or gate delays which is a 33 percent improvement over the one using (3:2) counters.

Although a multiplier designed using Figure 4.14b (7:3) counter implementation has a higher performance than the other implementation, its gate count increases by more than 13 percent in comparison to the 3.2 counters(Figure 4.14a). To offset this problem Paul J.Sang and Giovannic D.M [20] proposed implementation of (7:3) counter using folded transistors. Figure 4.14c shows the folded transistor implementation of the output C2 of the (7:3) counter. Even though this approach reduces the number of transistor counts it has the following three disadvantages:

a) Increased input capacitance when compared to the (3:2) counter.

b) More intermediate node capacitance and transistor capacitance increases linearly a as the number of inputs increases.

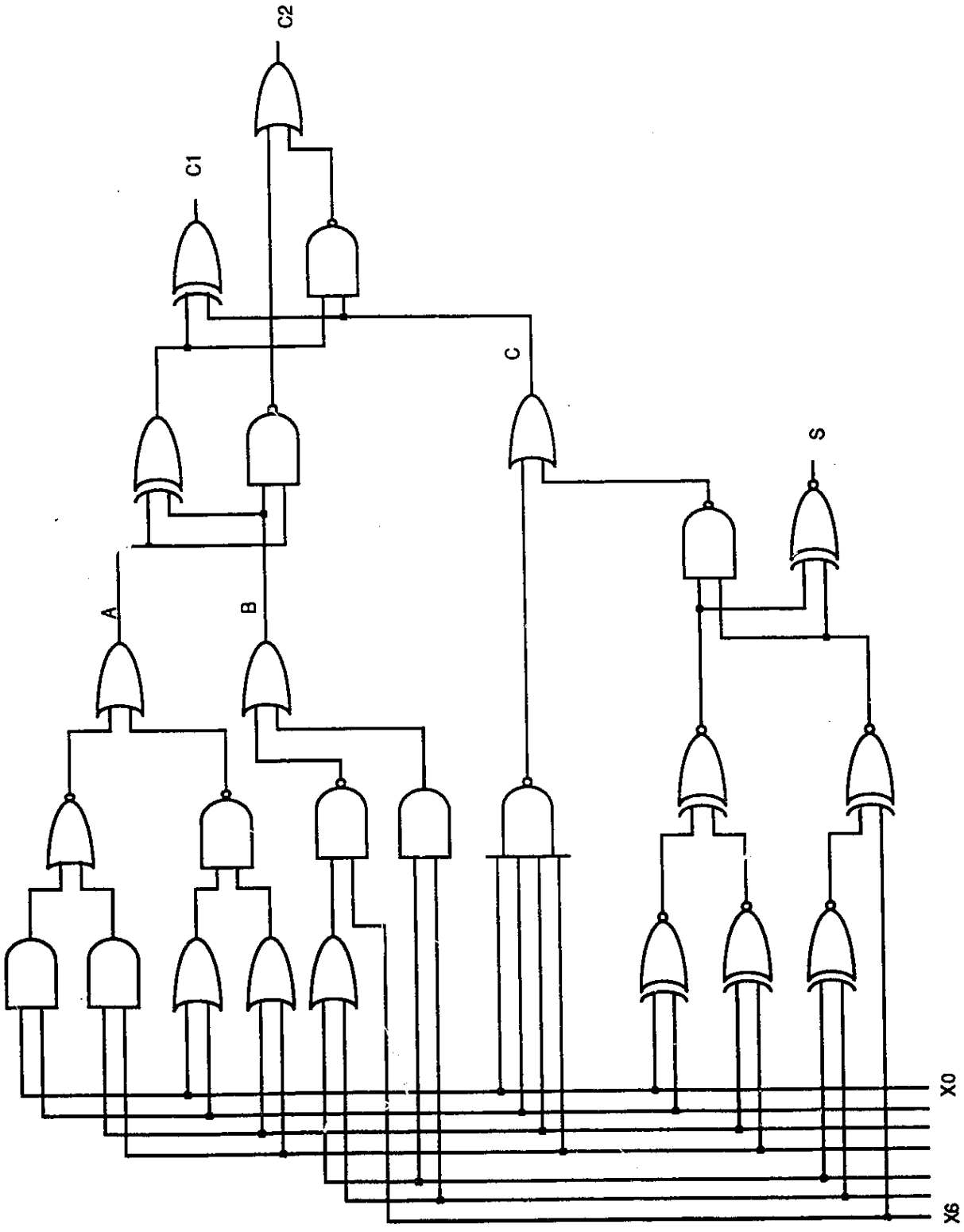


Figure 4.14b: Alternative Implementation of a (7:3) Counter

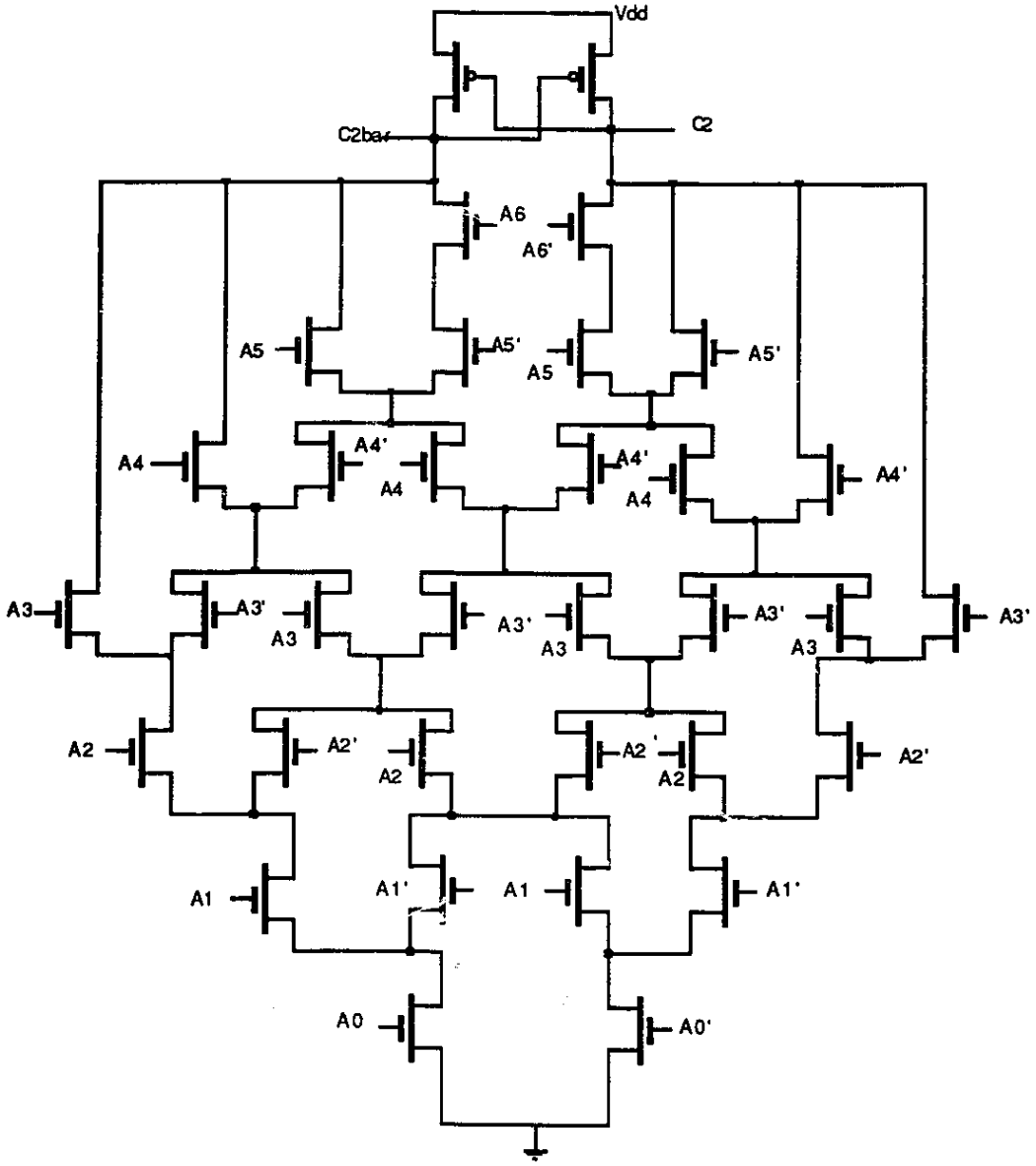


Figure 4.14c: (7:3) Counter Implementation with Folded transistor

c) Longer pull-down path - the longer the pull down transistor path, the smaller the pull down current is.

Due to the combinations of the above three effects the circuit slows down as the number of input increases. For these reasons, it is convenient to construct high input counters such as

7:3 counter using (3:2) counters . A multiplier implemented using 7:3 counter reduces both the number of cross stage wiring and the number of interconnection wiring. However, it has the following drawbacks: First, it has a higher gate delay than the Dadda structure. Second, it has longer interconnection wiring than the Dadda structure, as shown in Figure 4.14d. Thirdly, it has a higher gate count than the Dadda structure.

4.8 New Tree Structure

In the previous sections, it was explained that using higher order counters instead of a (3:2) counters, for a full tree structure implementation, does not give a regular structure. In this section a new multiplier architecture will be described as developed by Dr. Wang (a member of a VLSI research group in University of Windsor) which is both time optimal and regular in structure.

The new multiplier architecture, like the Dadda architecture, uses a (3:2) counter and has a depth proportional to log(N). However, unlike the Dadda algorithm, the new multiplier architecture does not obey the relationship:

$$T_{j+1} = \left\lceil 2/3 \times T_j \right\rceil$$

The structure for an 8 by 8 bit multiplier using the new technique is shown in Figure 4.15. The new multiplier architecture shown in Figure 4.15 will be referred to as the Column Compression Multiplier.

The Column Compression Multiplier has much better regularity when compared to the Dadda structure shown in Figure 4.11.

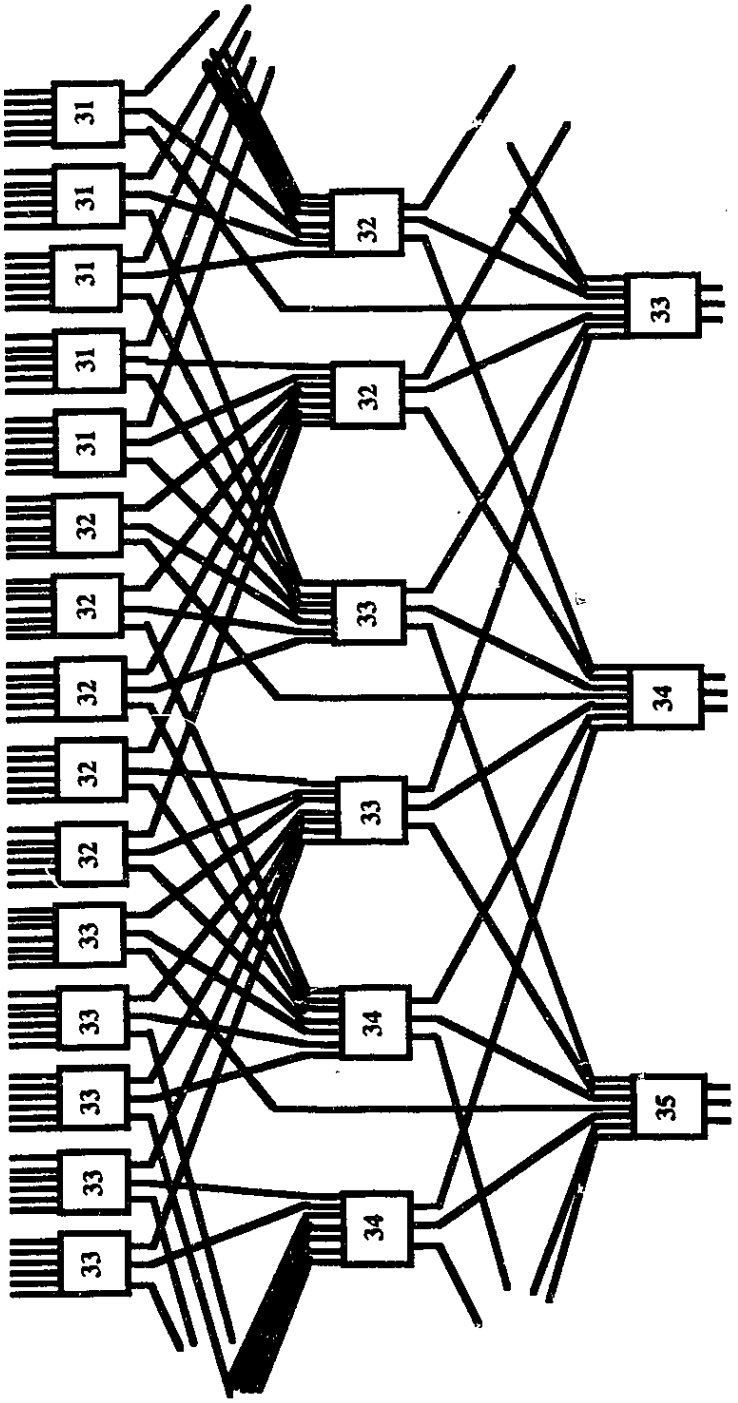


Figure 4.14d: Typical Structure of a (7:3) Counter Multiplier

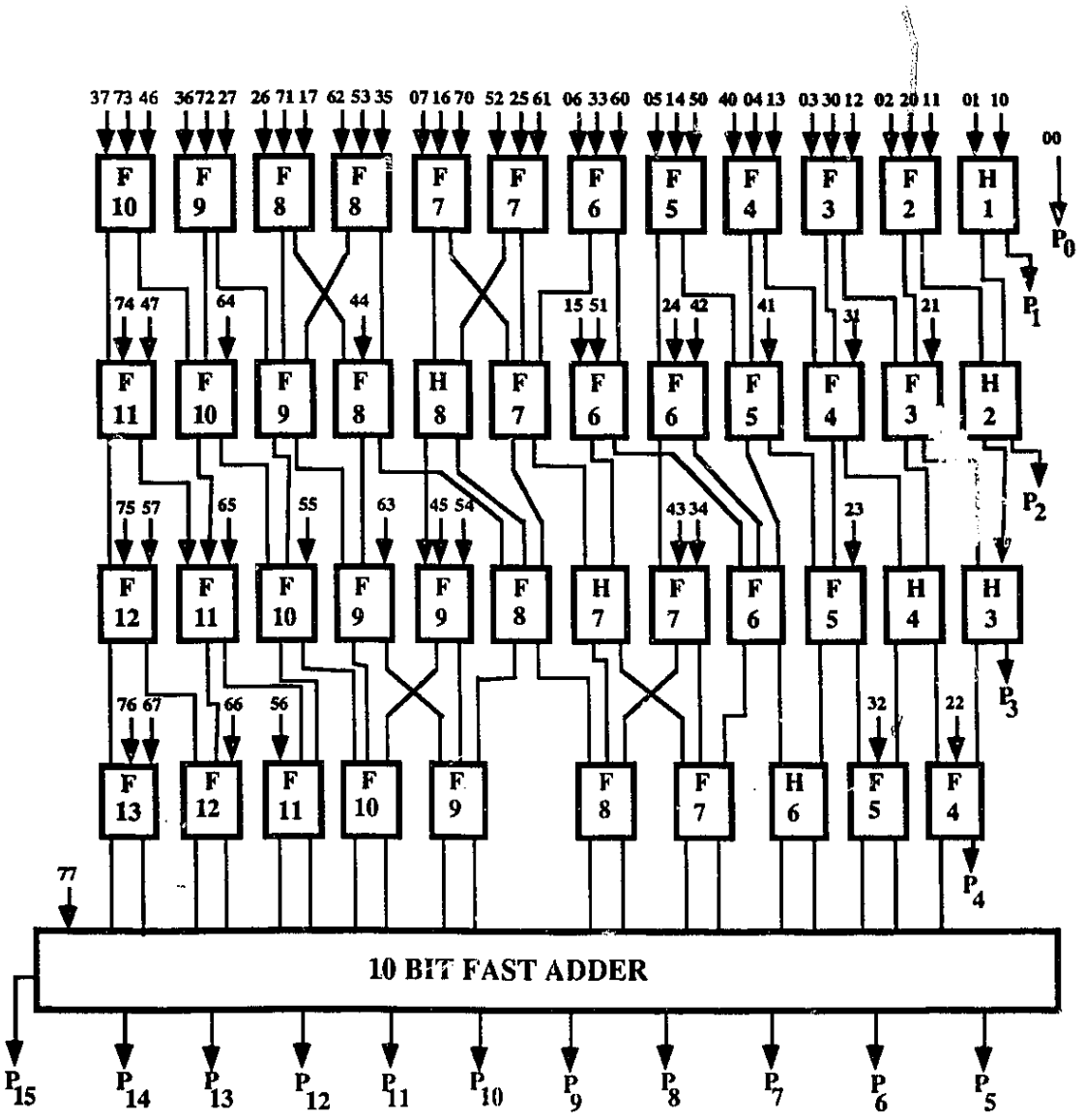


Figure 4.15: 8x8 Bit Column Compression Multiplier

In order to have a better measure of comparison of the various multiplier architectures presented in earlier sections a 16 by 16 bit multiplier has been taken as a base. Table 4.3 shows the number of components (i.e adders, compressors, and counters) and the total equivalent gate count of several multipliers. The equivalent gate count is the sum of the number of "simple gates" (i.e inverters, 2-input and 3-input nand gate and nor gates), 1.5

times the number of complex gates and two times the number of exclusive-or gates. Figure 4.16 shows the performance measure of the multipliers in table 4.3. Note, since the implementation of the (7:3) counter and (4:2) compressor without using (3:2) counter cells has a very high gate count, the (7:3) and (4:2) multipliers in table 4.8 are assumed to be implemented using (3:2) counters.

	Wallace	Dadda	(4:2) Compressor	(7:3) Counter	Column Compressor
Half-Adder	35	16	18	10	16
Full-Adder	200	196	18	30	194
(4:2) Compressor			95		
(4:3) Counter				6	
(5:3) Counter				8	
(6:3) Counter				11	
(7:3) Counter				28	
Total Gate Count	**1505**	**1420**	**1510**	**1464**	**1406**

Table 4.3: Comparison of the Complexity of Various 16 by 16 Bit Multipliers

The multipliers performance was compared by evaluating the number of gate delays to reduce the number of partial products from sixteen to two. Note, however the total time to perform the multiplication must also include the time to generate the partial products (one gate delay) and the time to sum the two words in the carry propagate adder (10 gate delay).

The Wallace, Dadda and Column Compression structures use six-levels of full-adders (four gate delays each) to reduce the number of partial products from sixteen to two. As each full-adder has two exclusive-or gate delays (two "simple" gate delays each), their delay is 24. The (4:2) compressor approach uses four exclusive-or gates per level to reduce the partial products to two giving a total of 24 gate delays. The (7:3) counter has a total of 28 "simple" gate delays.

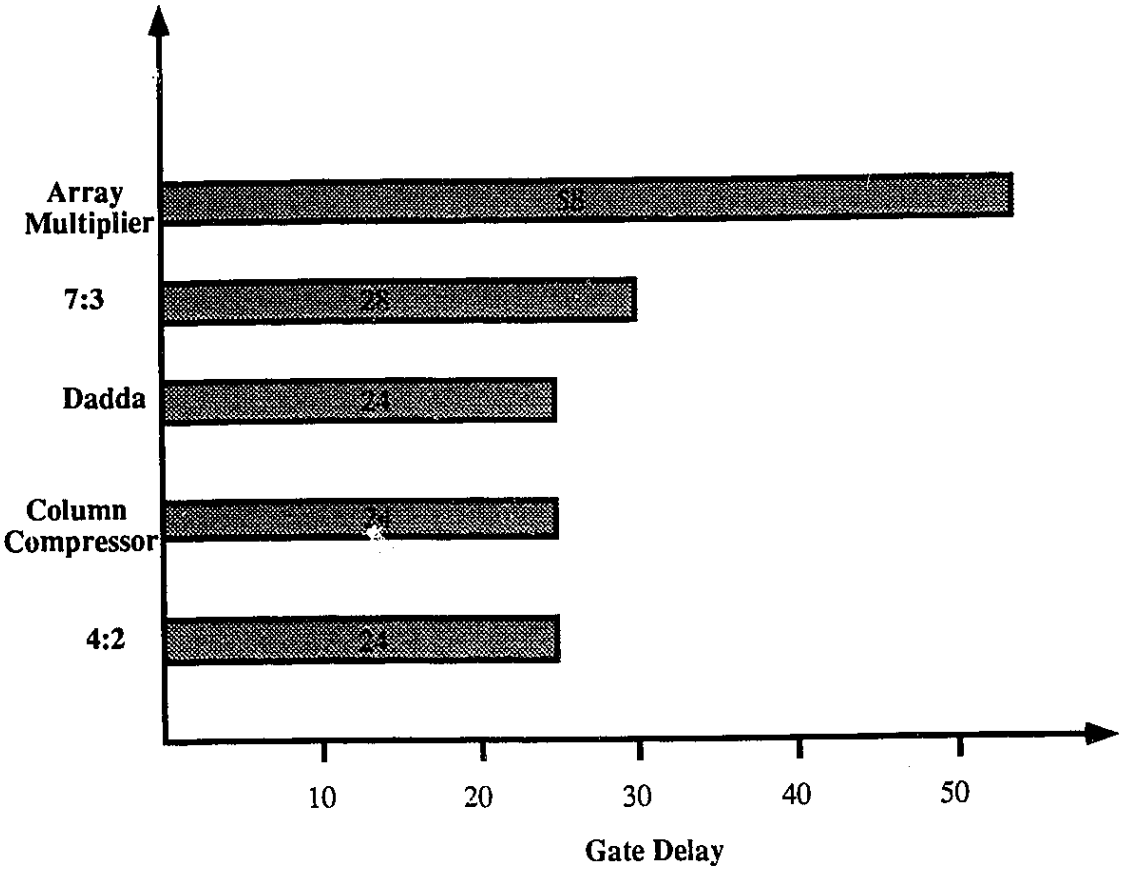


Figure 4.16: Performance Measure Of Various 16 by 16 Bit Multipliers

Implementation based on the Column Compression and (4:2) compressor improve upon the (7:3) counter scheme in terms of overall delay. Even though the (4:2) compressor structure has the same overall delay as the Column Compression multiplier, it has the following

major drawbacks: First, the multiplier implemented with (4:2) compressors has 7.5 percent higher gate count than the Column Compression Multiplier. Second, the (4:2) compressor requires long interconnections between adjacent compressors for intermediate carries.

Implementation based on (7:3) counters has several drawbacks besides its low performance compared to the Column Compression Multiplier. First, the multiplier implemented with (7:3) counters has 5 percent higher gate count than the Column Compression Multiplier. Second, the multiplier implemented with (7:3) counters will require more blocks at the early stages of the partial-product reduction process than the column multiplier architecture; this leads to long interconnections to gather bits of the same weight from outputs of the higher stage, as shown in Figure 4.14d.

Therefore multipliers implemented with either (4:2) compressor or (7:3) counter are expected to be more difficult to lay out than the Column Compression Multiplier. From the comparison results, one can conclude that the new multiplier architecture is superior in both speed and structure compared to the other multiplier architectures. This new multiplier architecture will be used for the mantissa section of the floating point algorithm described in chapter 3.

As mentioned earlier, the performance of a multiplier is not only determined by the column compression part, but also by the carry propagate adder at the last stage of the multiplier. Therefore a comparison of three adders in terms of their complexity and performance will be described in the next section.

4.9 High Speed Adders

4.9.1 Carry Look Ahead

One scheme of adding two n bit operands is by cascading n full-adders; this is effectively a ripple carry adder. The disadvantage of this scheme is that the delay time is linearly proportional to the size of the input operands. Carry look-ahead (CLA) is a technique which is used to speed up the carry propagation in an adder. The carries to each stage of the adder are calculated by additional logic circuitry. As a result, the addition time will improve at the expense of using more hardware for the carry look-ahead unit. Theoretically, one should be able to expand the CLA unit freely, and build adders of any word length, but, because of fan-in and fan-out limitations, single level CLA is applied only to the design of adders of length four in CMOS circuits.

One solution to the high fan-in problem is to break the large single CLA unit into a number of smaller CLA units and let the carries ripple between the units. The organization of a 10-bit carry look-ahead adder with three CLA units is shown in Figure 4.17. The total delay of this type of adder is the sum of the delays due to the propagate/generate unit and sum unit plus delays through CLA units. The circuit layout of this type of adder is quiet irregular.

4.9.2 Binary Carry Look Ahead

It was noted in the previous section that the structure of ripple CLA is very irregular. In fact, Mead and Conway [21] considered several look-ahead schemes, but concluded that " they added a great deal of complexity to the system without much gain in performance". This argument of Mead and Conway was disproved by Brent and Kung [22].

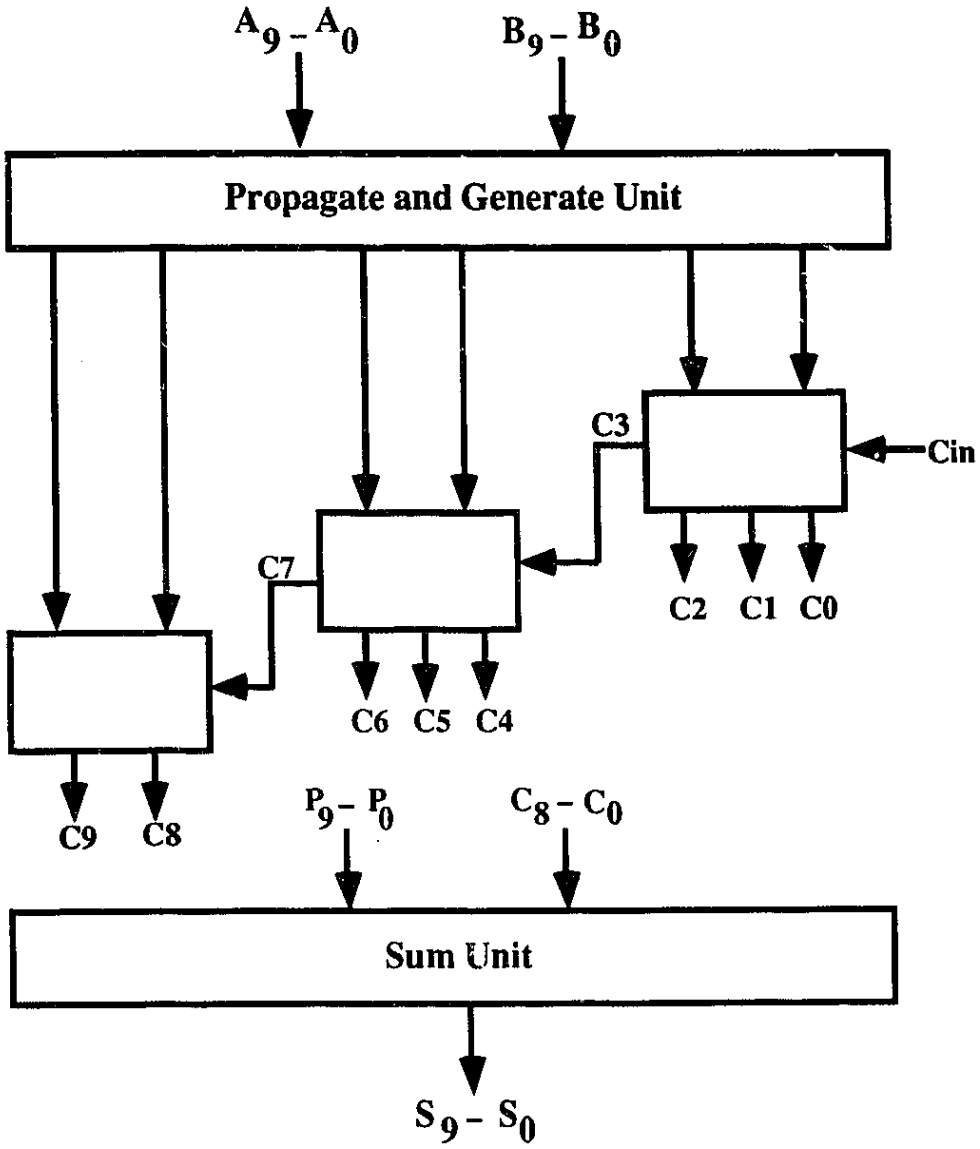


Figure 4.17: Organization of a 10_Bit Ripple Adder

The Brent and Kung's technique can be illustrated by first reviewing the equations for the binary adders i.e.

$$C_i = G_i + P_i C_{i-1}$$

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \text{ and } B_i$$

$$S_i = C_{i-1} \oplus P_i$$

Both Gi and Pi can be determined in constant time, so Ci is the only time critical term that needs to be calculated. Brent and Kung define a new operator, o, which has the following function:

$$(g, p) \text{ o } (g', p') = (g + (pg'), pp')$$

where g, p, g' and p' are Boolean variables. It can be shown that this new operator is associative [22] and the carry signal can be determined by

$$C_i = G_i$$

where

$$(G_i, P_i) = (g_1, p_1) \text{ if } i=1$$

$$= (g_i,p_i) \ldots (G_{i-1}, P_{i-1}) \quad \text{if } 2 \leq i \leq n$$

The associative property of the o operator allows the processing elements to be embedded in a binary tree structure of depth O(logn). The organization of a 10-bit binary CLA adder is shown in Figure 4.18. The carry propagation time in this structure is proportional to $\log_2$ of the size of the adder and its area is proportional to n.

4.9.3 Modified Brent and Kung's Adder

One major drawback of Brent and Kung's approach is that the gate count required to produce a regular structure is rather high, so, in this work, an attempt has been made to modify Brent and Kung's adder to reduce the gate count. This modified structure uses look-ahead blocks of four as opposed to two bits used in the Brent and Kung's approach.

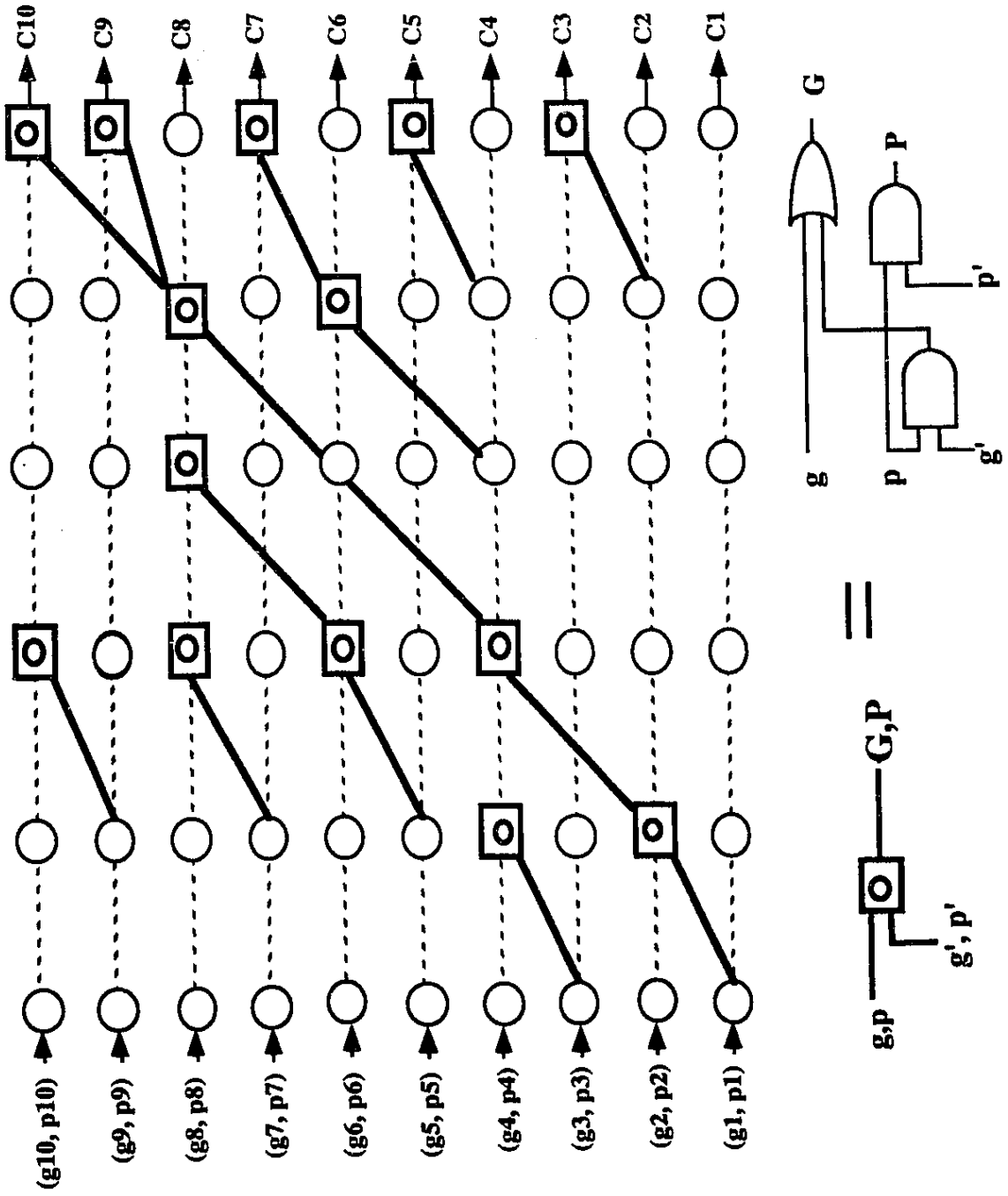


Figure 4.18: 10-Bit Binary Carry Look Ahead Adder

The equations for a 10-bit of the modified adder can be summarized as follows:

Propagate and generate unit:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

Group generate and Group Propagate unit

$$GP_0 = P_0 P_1 P_2 P_3$$

$$GG_0 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$

$$GP_1 = P_4 P_5 P_6 P_7$$

$$GG_1 = G_7 + G_6 P_7 + G_5 P_6 P_7 + G_4 P_5 P_6 P_7$$

$$GP_2 = P_9 P_8$$

$$GG_2 = G_9 + P_9 P_8$$

Group Carry Unit

$$GC_3 = GG_0 + C_{IN} GP_0$$

$$GC_7 = GG_1 + GG_0 GP_1 + C_{IN} GP_0 GP_1$$

$$GC_9 = GG_2 + GG_1 GP_2 + GG_0 GP_1 GP_2 + CIN_0 GP_0 GP_1 GP_2$$

Carry Unit

$$C_0 = G_O + C_{IN} P_0$$

$$C_1 = G_1 + G_O P_1 + C_{IN} P_0 P_1$$

$$C_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_{IN} P_0 P_1 P_2$$

$$C_3 = GC_3$$

$$C_4 = G_4 + GC_3 P_4$$

$$C_5 = G_5 + G_4 P_5 + GC_3 P_4 P_5$$

$$C_6 = G_6 + G_5 P_6 + G_4 P_5 P_6 + GC_3 P_4 P_5 P_6$$

$$C_7 = GC_7$$

$$C_8 = G_8 + GC_7 P_8$$

$$C_9 = GC_9$$

$$C_{10} = G_{10} + GC_9 P_{10}$$

Sum Unit

$$S_i = P_i \oplus C_i$$

The block diagram for a 10-bit of the modified adder is shown in Figure 4.19. Even though this modified structure has 40 percent less gate count than the Brent and Kung's adder it has a rather irregular structure.

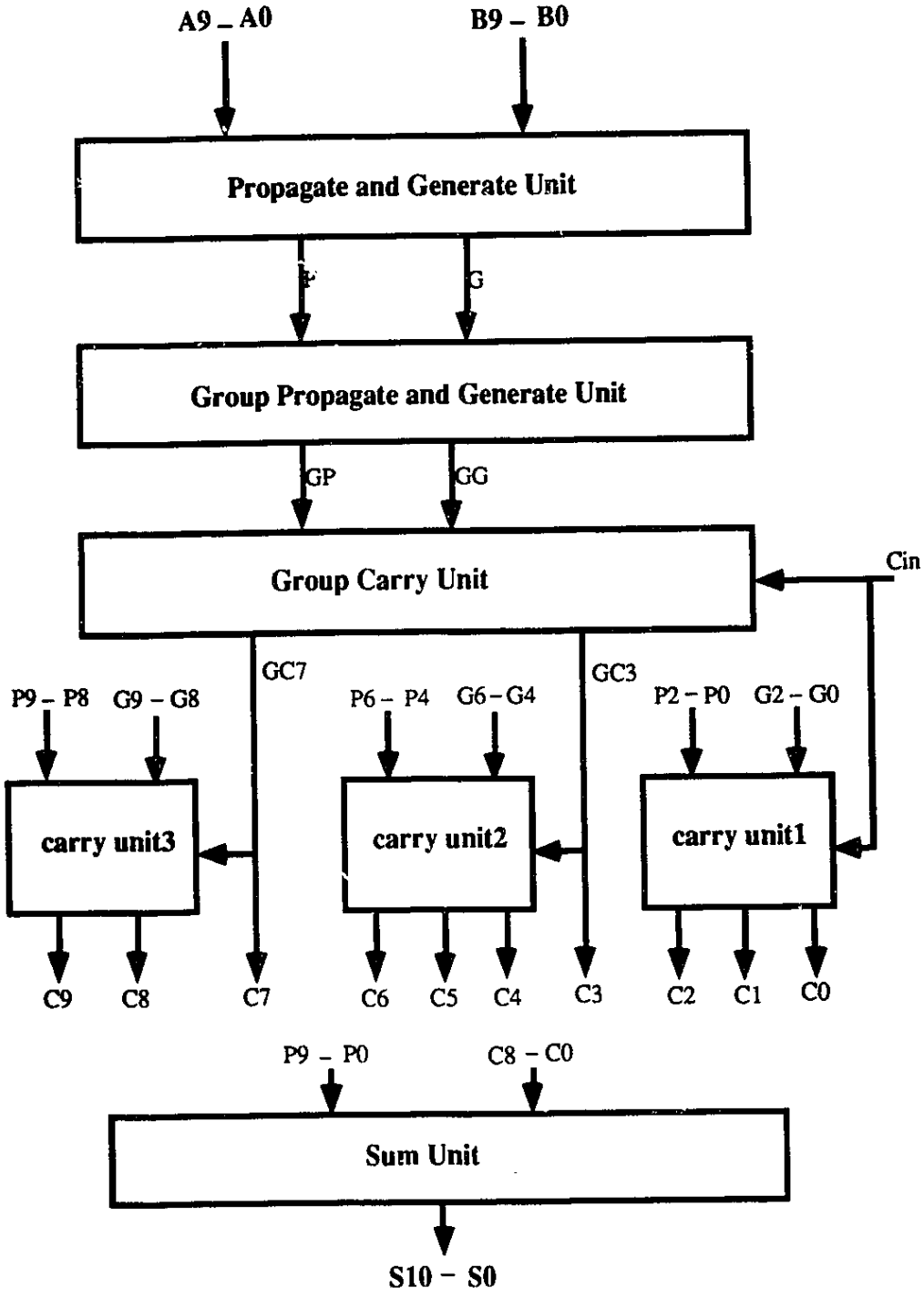


Figure 4.19: 10-Bit Modified Brent and Kung's Adder

4.9.4 A New Adder Structure

It was noted in the previous section that to reduce the gate count, four bit look-ahead blocks were used instead of two bit look-ahead blocks. The penalty is an irregular structure. Now we will see a new adder structure proposed by Wang which requires a lower gate count than Brent and Kung's adder and also possesses a better structure than the modified adder. The new adder structure, as with the modified adder, uses four bit look-ahead blocks, but unlike the modified adder it only generates even carries. Due to the generation of only even carries, the new adder structure has less interconnection wiring than the modified adder, resulting in more regular layouts than the modified adder. The organization of a 10-bit adder (new adder) is shown in Figure 4.20.

Where:

$$g_i = a_i\, b_i$$

$$p_i = a_i \oplus b_i$$

$$GG_{2i} = [(a_{2i} \oplus b_{2i})g_i] + (a_{2i}\, b_{2i})$$

$$GP_{2i} = p_i p_{2i}$$

$$C_{2i+2} = [GP_{2i+2}\, C_{2i}] + GG_{2i+2}$$

$$S_{2i+1} = p_{2i+1} \oplus C_{2i}$$

$$S_{2i+2} = P_{2i+2} \oplus (g_{2i+1} + P_{2i+1}C_{2i})$$

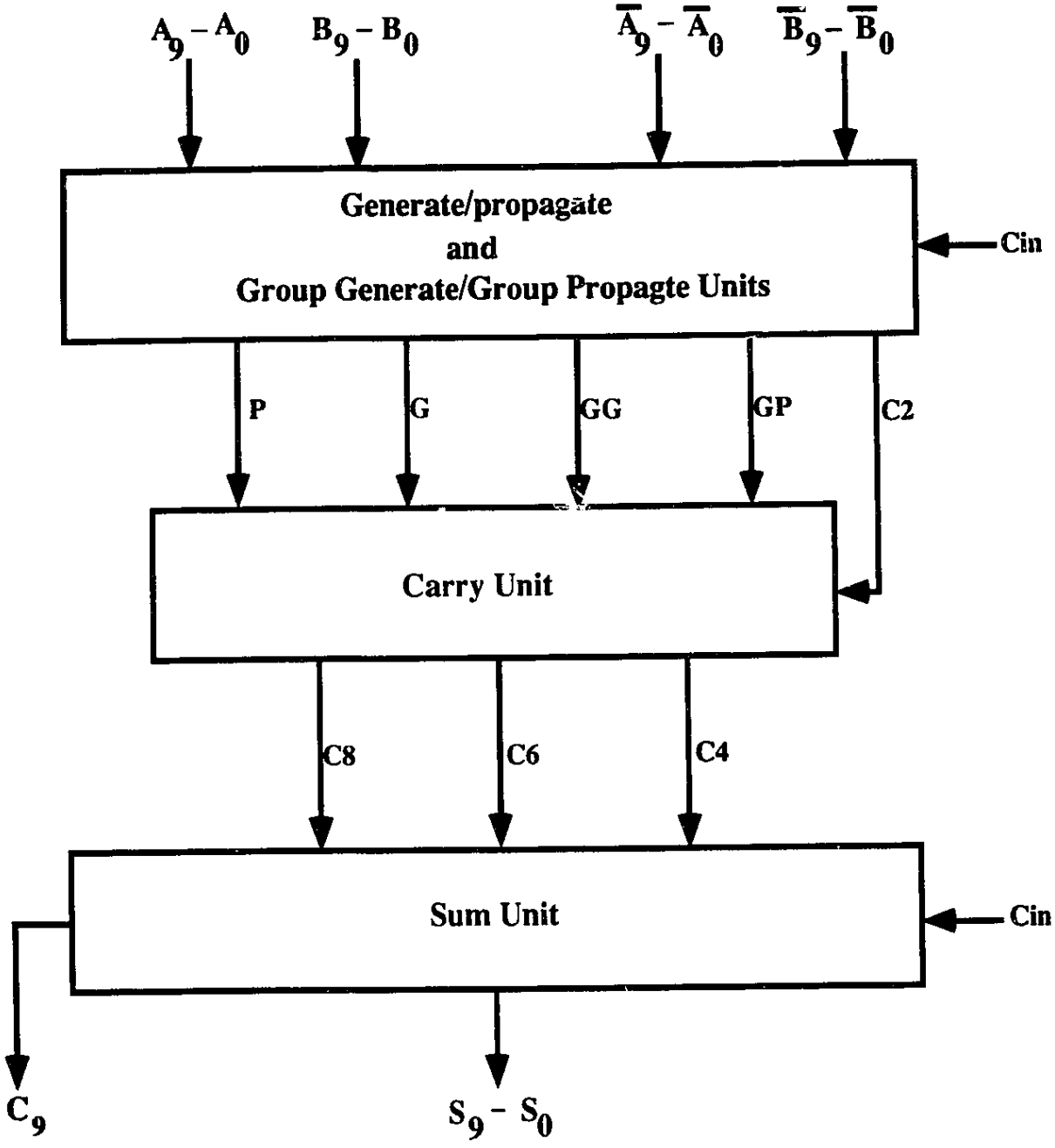


Figure 4.20: Structure of a 10-Bit New Adder

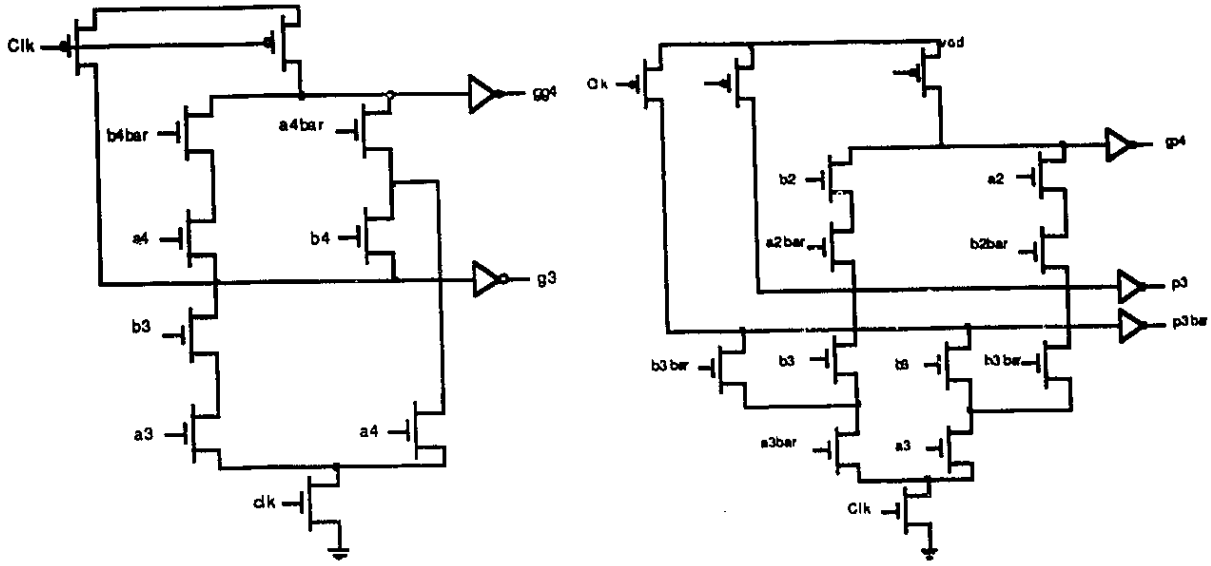


Figure 4.21a: MODL Gates for Propagate and Generate Terms

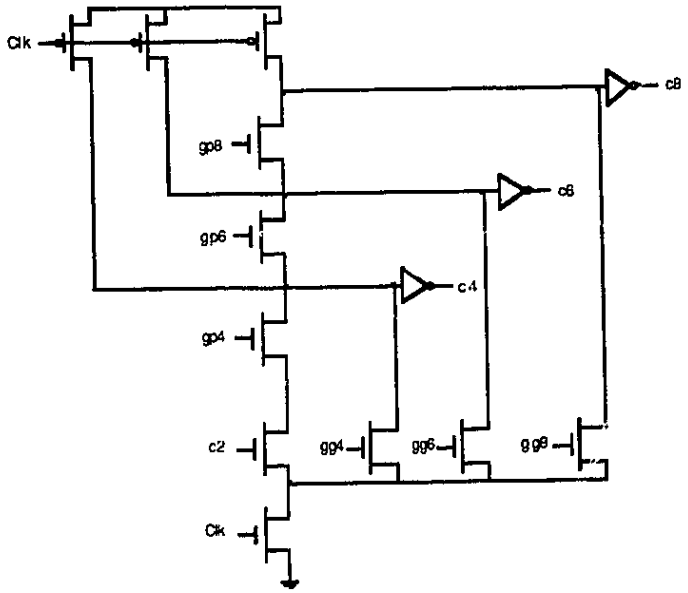


Figure 4.21b: MODL Gate for the Carry Block

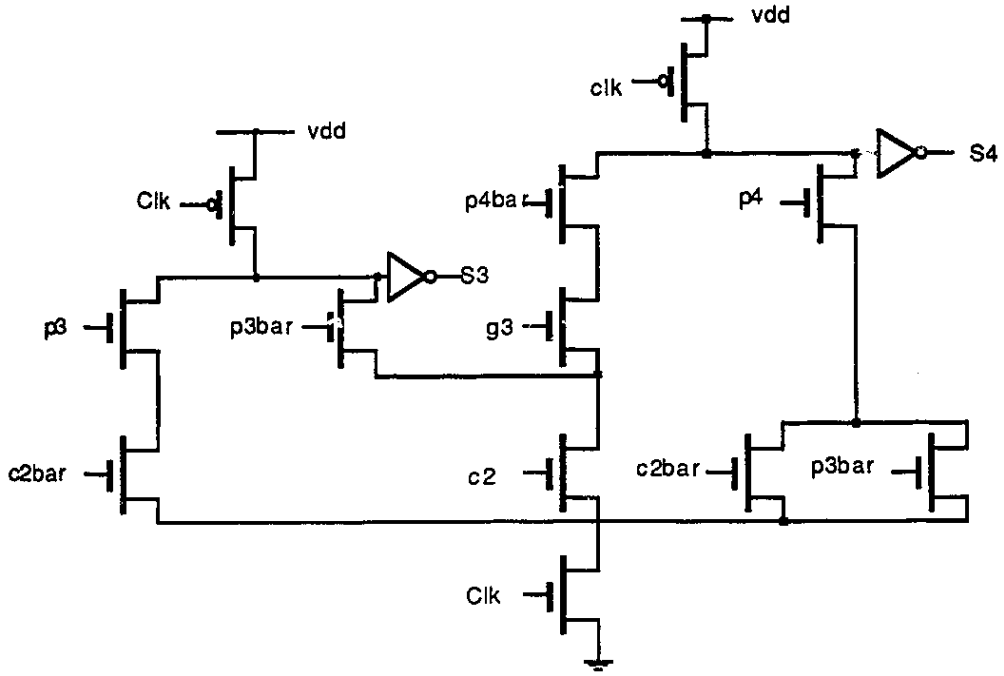


Figure 4.21c: MODL Gate for the Sum Block

The MODL implementations of the 10-bit adder are shown in Figure 4.21a to Figure 4.21c. From Figure 4.21 it can be noted that the implementations of the new adder using MODL (multiple output domino logic) is very efficient due to the recurrent nature of the carry look-ahead equations. Although the area advantage of MODL over standard domino logic is rather apparent, the speed advantage is not as obvious especially at the gate level. This is because each MODL gate implements more than one function, and performance improvement is achieved by capitalizing on this fact at a higher level. In an organization optimized with respect to the MODL technique, the improvement of performance is due to a reduction of load capacitance for a given logic stage. This results from less overall device count (and therefore less fan-out for a given output), and less parasitic capacitance as a consequence of smaller layout.

One drawback of the MODL logic is that it can provide only non-inverting gates. Therefore a modified MODL logic was developed to realize the new adder functions to accommodate both true and complement signals. The functionality of the new adder was fully simulated using the VHDL simulator. The adder was also simulated in HSPICE up to a maximum of 70 Mhz. The simulation and layout of the new adder are shown in Figure 4.22 and 4.23 respectively.

The comparison of the adders described in this section i.e. Binary CLA (type_1), Modified Brent and Kung's adder (type_2) and new adder (type_3) is shown in table 4.4. Since the multiplier has to be implemented using only static logic, the three adders compared in table 4.4 are assumed to be designed using only static logics. Note the delays are calculated assuming that the adders are implemented in 0.8μm BICMOS technology.

	number of stages	overall Delay	Gate Count	Complexity
Type 1	7	6.5 ns	122	regular
Type 2	5	5.0 ns	82	irregular
Type 3	3	5.5 ns	74	rather irregular

Table 4.4: Comparison of the three Adders

If the adders are to be implemented in BICMOS, type_3 adder is the best choice for the following reasons. First, type_3, compared to type_2, has fewer interconnections,

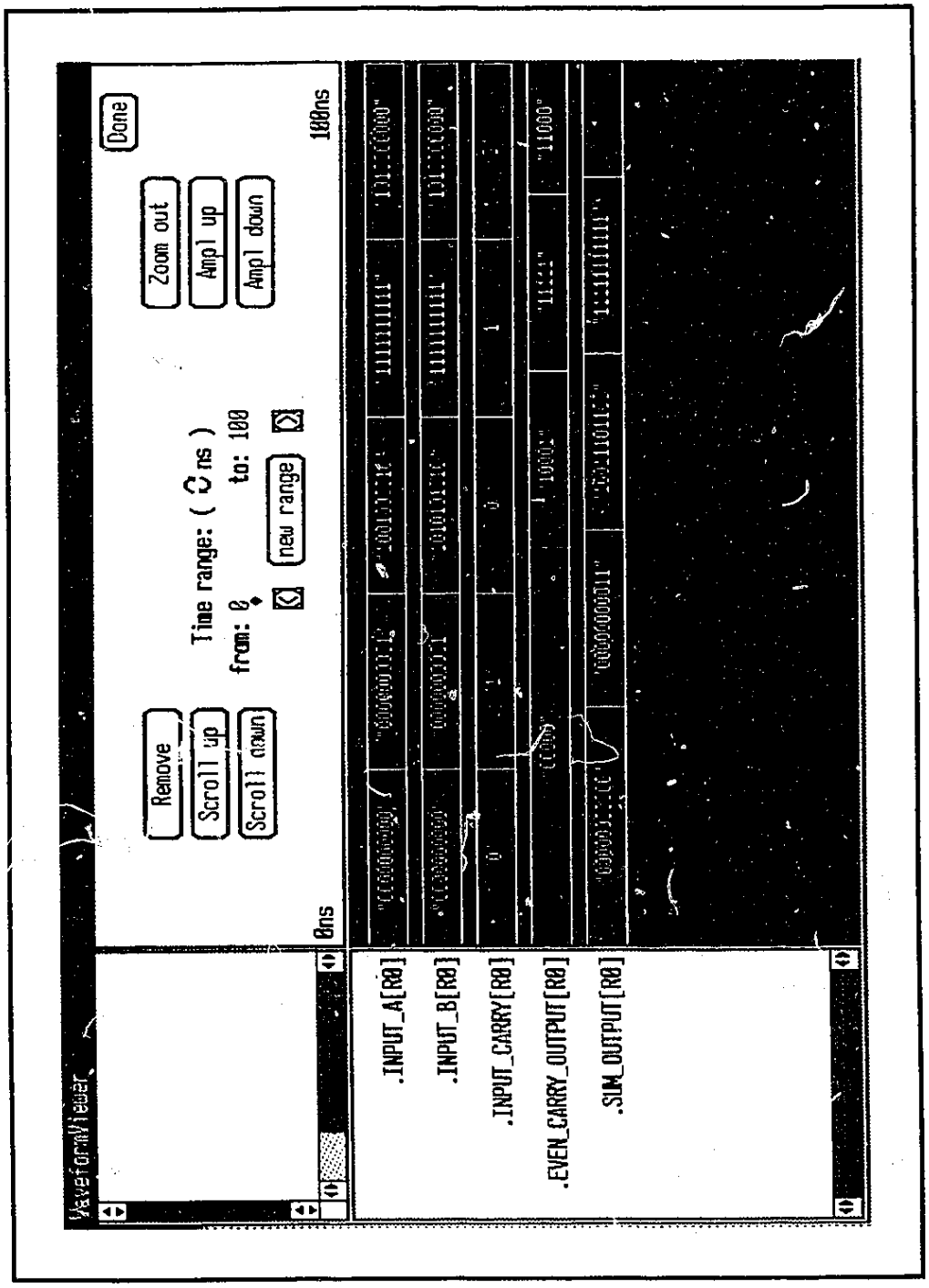


Figure 4:22: VHDL Waveforms for 10-Bit New Adder

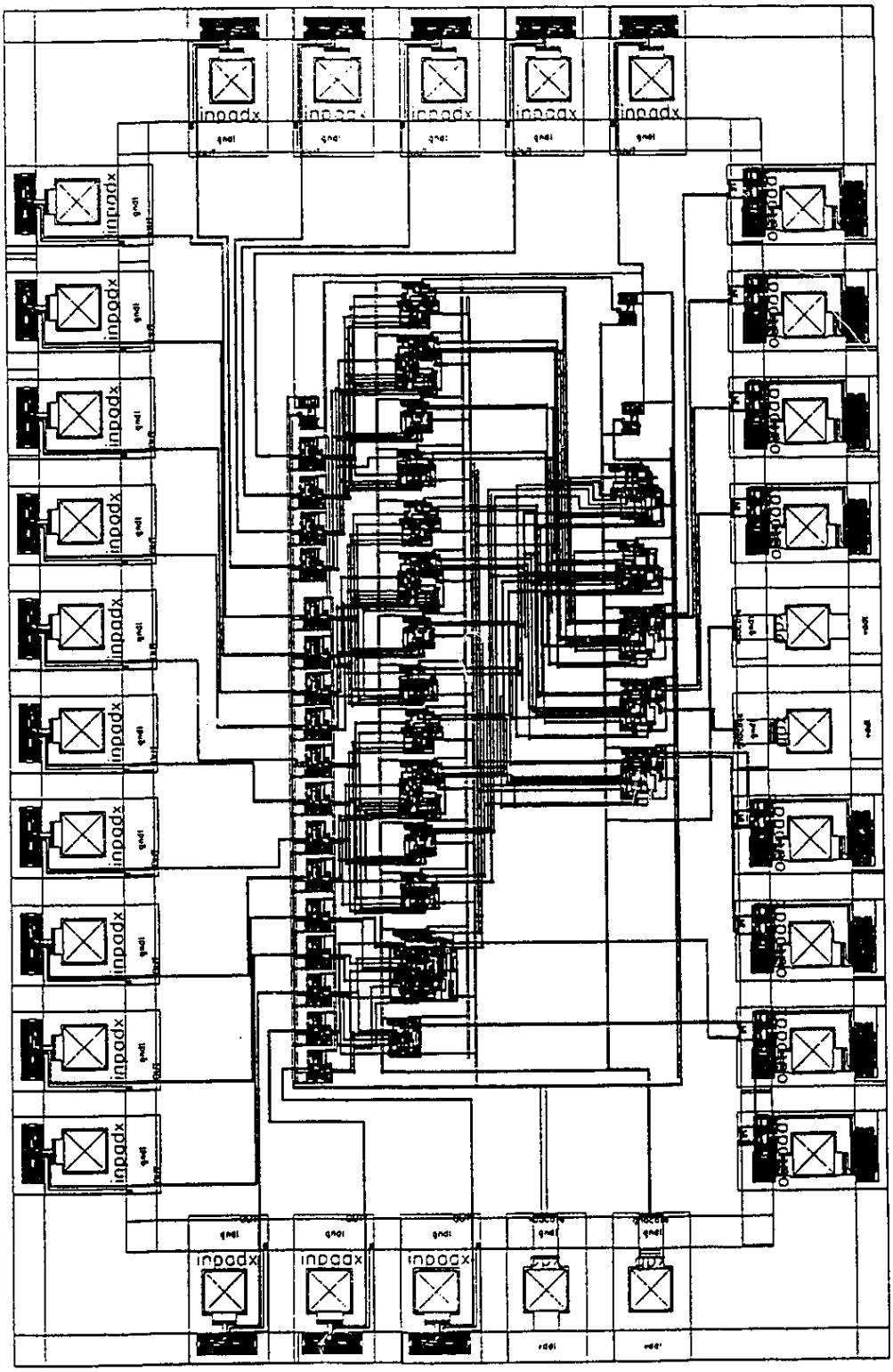


Figure 4.23: Layout for 10-Bit New Adder

as a consequence a type_2 adder is more difficult to layout than a type_3 adder. In addition, due to the smaller number of interconnections, the type_3 adder has less parasitic capacitance. Therefore, a type_3 adder has a better overall performance than a type_2 adder. Second, a type_3 adder is better than a type_1 adder in terms of both speed and gate count. Even though a type_3 adder is less regular than a type_1 adder, this advantage is less significant in the BICMOS process because the BICMOS process has three layers of metals for use as interconnects.

Since the multiplier will be implemented in BICMOS, as mentioned before, a type_3 adder will be used for the adder at the last stage of the multiplier.

4.10 Implementation

The preceding two sections described the new multiplier structure, referred to as the Column Compression Multiplier, and a new adder structure which will be used at the final stage of the multiplier. To demonstrate the feasibility of this new architecture, an 8 by 8 bit multiplier has been designed in a new 0.8μm BICMOS technology. Note, only static logic is used for the implementation of the multiplier. To facilitate the structural testability of the multiplier, scannable D-latches were used for the input and output latches. The VHDL simulation result ,and the layout of the multiplier, are shown in Figure 4.24 and Figure 4.25, respectively. Its latency is 16 ns, including the delay of the input and output latches. The multiplier has a core size of 0.950mm x 0.782mm.

An 8 by 8 bit floating point multiplier has been taken as a model for investigating the VLSI implementation of the new floating point multiplier algorithm developed in this thesis. The column compression multiplier is used for the mantissa section of the floating point multiplier as shown in Figure 4.26. Since, there is an empty slot in the array multiplier, Rin is placed in the empty slot as described in Chapter 3. Similarly, to create an empty slot

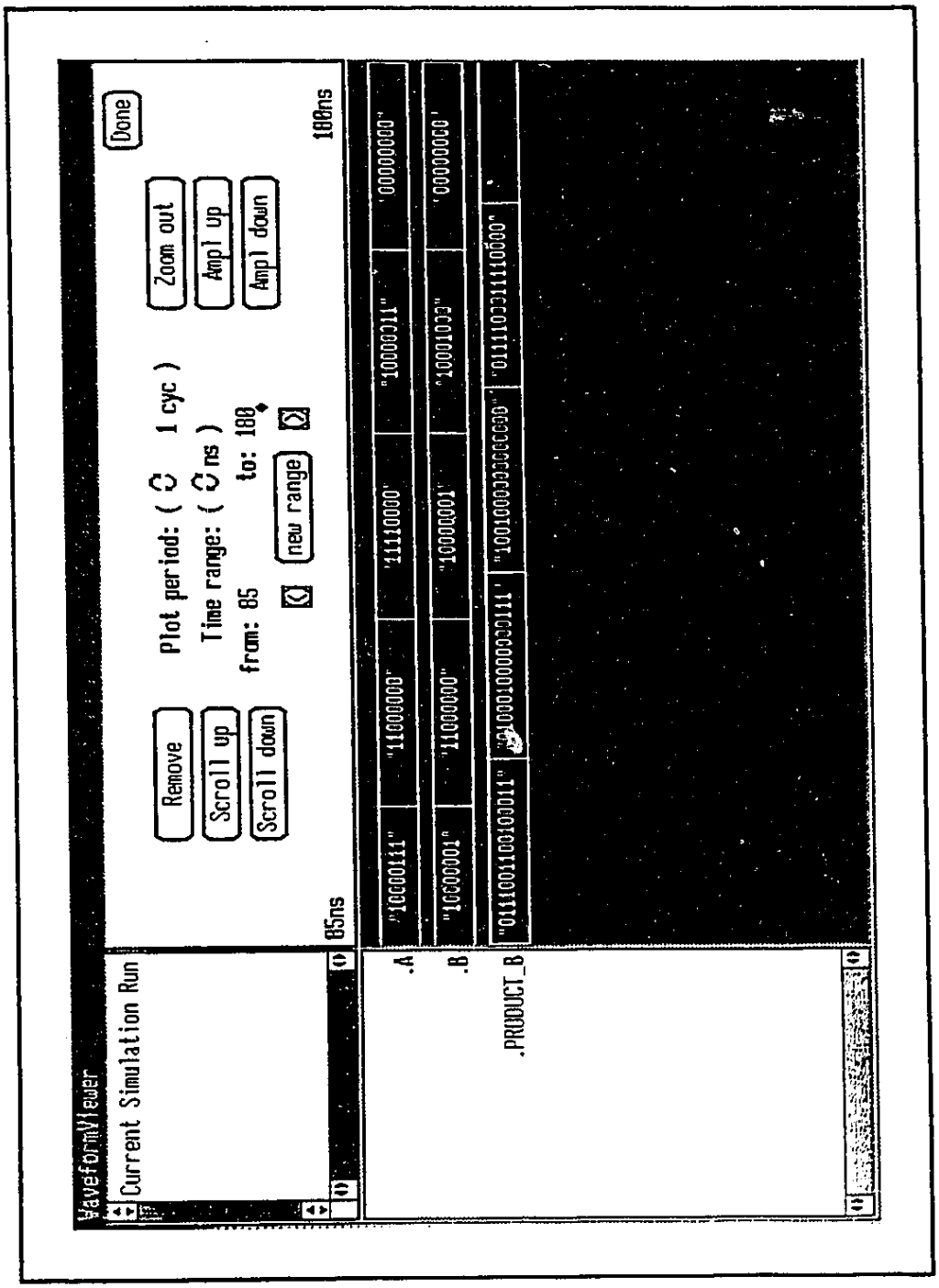


Figure 4.24: VHDL Waveforms for an 8x8 Bit of Column Compression Multiplier

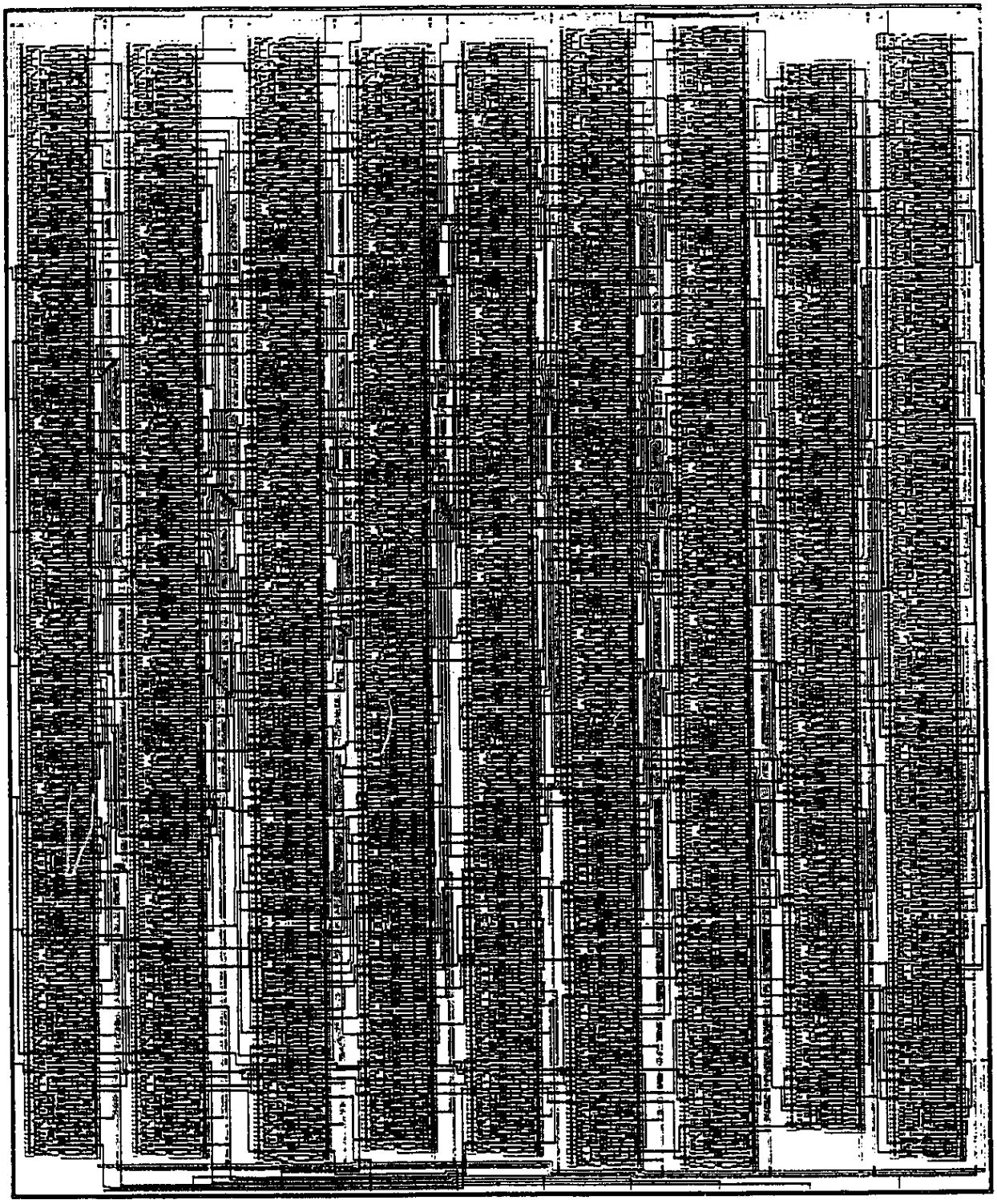


Figure 4:25: Layout for the Column Compression Multiplier

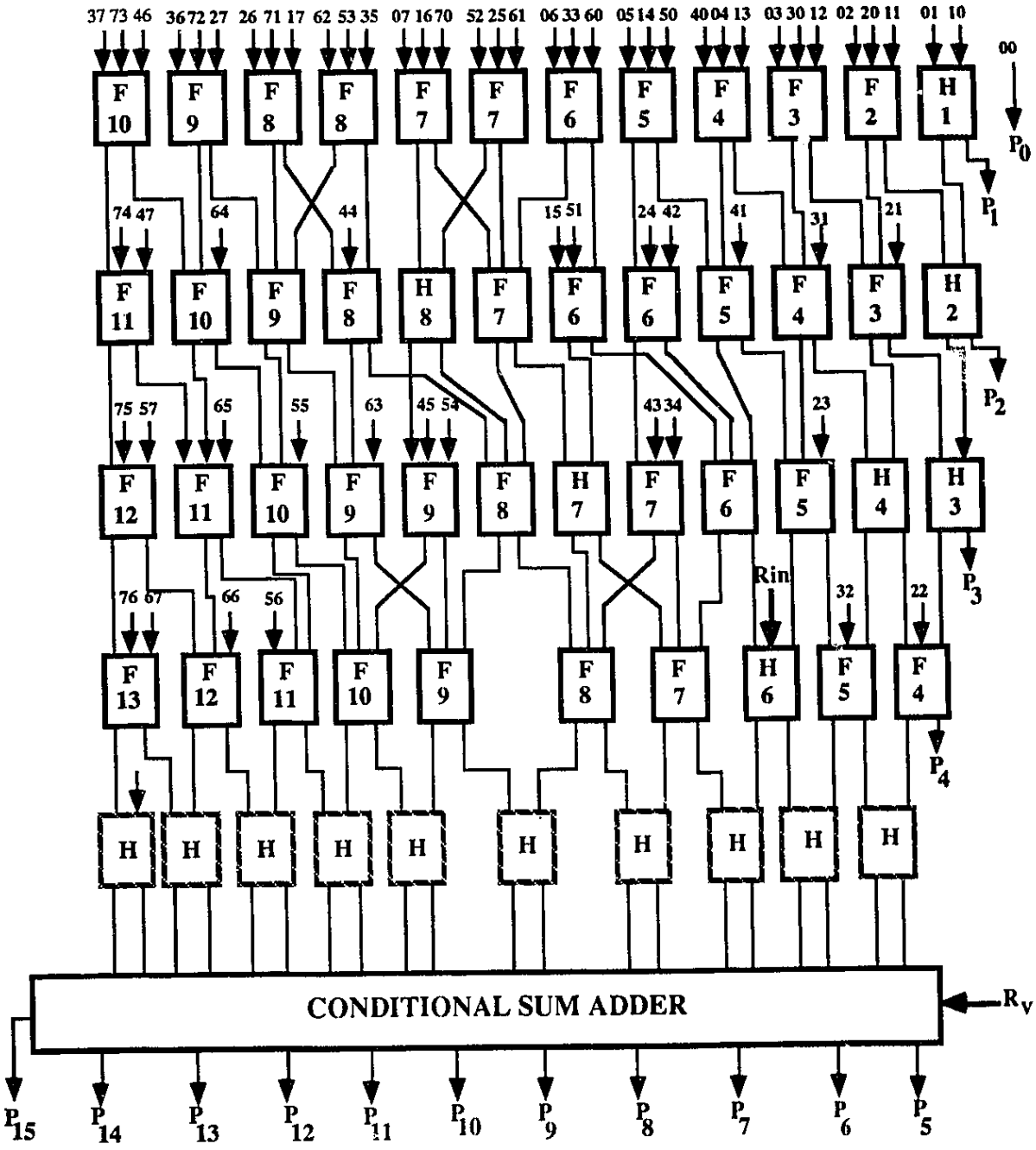


Figure 4.26: Mantissa Section for an 8x8 Bit Floating Point Multiplier

for Cin (a carry from the lower order bits) a row of half adders are used as shown in Figure 4.26. Finally a conditional sum adder is used which produces two outputs, one for Rv=0

and the other for Rv=1. When the overflow bit is known, the correct sum will be selected. The block diagram representation of the conditional sum adder is shown in Figure 4.27. Note, since the carry out (V) is known at the second stage, only the carry block need to be duplicated.

The following example illustrates how the floating point multiplication is processed:

Example:

135 x 192

11000000

10000111

11000000

11000000

11000000

11000000

0110010101000000 V=0 L=0 R=1 S=0

Rin 1

110010110 (result after round to nearest)

11001010 (result after round to nearest/even)

Note, since the carry out of the result before rounding is zero (v=0), Rin is added at the 2^{-n} bit position for rounding to nearest. After rounding to nearest the L bit is changed from

zero to one. Before rounding the L bit was zero, the R bit was one and the S bit was zero, therefore we have a tie case. According to the IEEE standard for rounding, if there is a tie, the number must be rounded to even. Therefore, the L bit is restored to zero as shown in the final result.

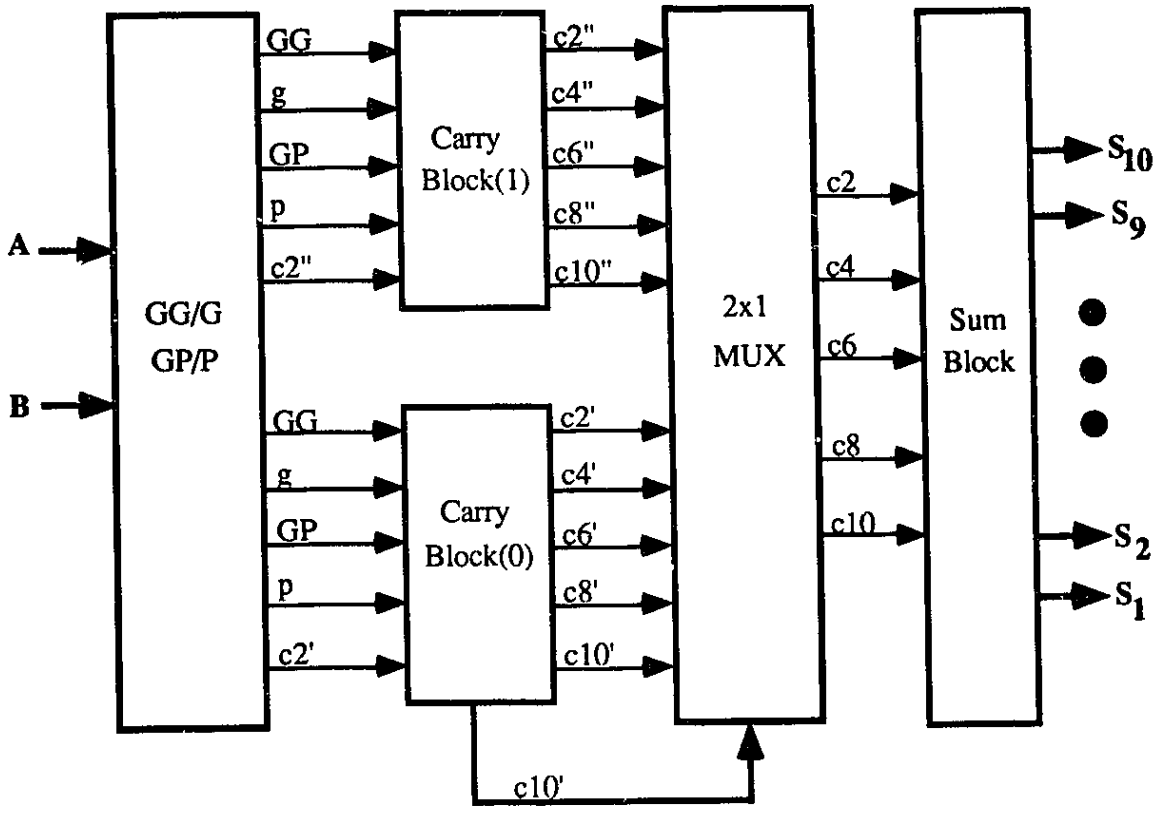


Figure 4.27: 10-Bit Conditional Sum Adder

The VHDL simulation result of the 8 by 8 bit multiplier is shown in table 4.5. The latency of the multiplier is 21 ns.

Time	Ain	Bin	CA	CB	V	L	R	S	CR	MR
0 ns	00000000	00000000	0	0	0	0	0	0	00000000	00000000
21 ns	10000000	10000000	129	127	0	0	1	0	00000000	00000000
41 ns	11000000	10000111	100	192	0	0	1	0	10000001	10000000
61ns	10000101	11000000	254	6	0	1	0	0	10100101	11001010
81 ns	10000011	11000011	108	126	0	0	0	0	10000101	11001000
101 ns	11000000	11000000	92	124	0	0	0	1	01101011	11001000
121 ns	11000110	11000000	200	130	1	0	1	0	01011010	10010000
141 ns	11000000	11000001	131	135	1	1	0	0	11001100	10010100
161 ns	10000111	10000111	80	130	1	1	0	1	10001100	10010001
181 ns	10000011	11111111	1	124	0	0	1	1	01010011	10001110
201 ns	11000000	11000010	254	254	1	0	1	1	00000000	10000010
221 ns	10011000	10000001	63	1	1	0	0	0	11111111	10010010

Table 4.5: VHDL Simulation Result for an 8x8 Bit Floating Point Multiplier

In summary, in this chapter we have described several parallel multiplier architectures and adder structures including the simulation and BICMOS implementation of the two new parallel multiplier architectures. In addition the simulation and VLSI implementation of the new floating point unit multiplier algorithm have been discussed. From the simulation result, it can be said that both the fixed and floating point unit multipliers are very competitive with any of the existing multiplier architectures.

CHAPTER 5

Conclusions and Future Work

5.1 Conclusions

This thesis has presented a new fast and efficient floating point multiplier algorithm. The high performance is obtained by using fast rounding techniques and new fast techniques for computing the exponent and the sticky bit. The novel technique developed for the exponent computation has increased the overall performance of the floating point multiplier by more than twenty percent. The new method for computing the sticky bit, removes the computation of the sticky bit from the critical path without increasing the hardware complexity. In addition, the new floating point multiplier algorithm detects overflow and underflow conditions using only three bits without much loss in the performance of the multiplier.

Since the two main components of the floating point multiplier algorithm are a fixed point multiplier and an adder, a comparison of several multipliers and adders architectures, including two new multiplier architectures, has been made. From the comparison of the multipliers one can draw the following conclusions:

First, if the number of bits in the multiplicand or multiplier is small, the array multiplier or the two-bit full-adder multiplier are favoured over the other architectures. They are more regular and the difference in speed, in comparison to the tree structure, is not significant for small N.

Second, even though trees such as the Dadda architecture get faster than linear arrays, they have a very irregular structure. As a result the VLSI implementation of the Dadda architecture is not feasible.

Thirdly, the new multiplier architecture, referred to as the Column Compression Multiplier, has been found to be both faster and smaller than other multipliers. In addition, a multiplier implemented with (4:2) compressors or (7:3) counters is expected to be more difficult to layout compared to the Column Compression Multiplier.

Similarly from the comparison of various adder structures the following observations are made. The ranking of the adders depends on the process used for their implementation. For instance, the binary carry look-ahead is superior to the others if the adders are to be implemented in CMOS technology. On the other hand, the new adder structure will be the best choice, if the adders are to be implemented in BICMOS. The performance loss due to irregular structure of the new adder when compared to the regular structure of the binary carry look-ahead is insignificant for the following two reasons: First, BICMOS technology uses three layers of metals for interconnection, while CMOS uses only two. The use of more levels of metals reduces the delay associated with the interconnection between cells,

which permits the over-routing of active circuitry. Second, BICMOS technology has a higher driving capability per unit capacitance than the CMOS technology.

Both of the new multiplier architectures, i.e. the Column Compression and two-bit full-adder multipliers have been implemented in 0.8μm BATMOS process. From VHDL simulation result, it was found that the latency for fractional multiply is under 16 ns for an 8 by 8 bit Column Compression Multiplier and 20 ns for an 8 by 8 bit two-bit full-adder multiplier.

The Column Compression Multiplier is also useful for implementing the floating point multiplier (i.e. for the mantissa section). To demonstrate its usefulness for floating-point, an 8 by 8 bit floating multiplier using the new floating multiplier developed in this thesis, was fully simulated in VHDL. From VHDL simulation result the latency of the 8 by 8 bit floating point multiplier was found to be 21 ns. This study proved that both the new floating multiplier algorithm and the new multiplier architectures work and are efficient and faster than current floating point multiplier algorithms and multiplier architectures respectively.

5.2 Future Work

An obvious continuation of this work would be to implement an IEEE compatible floating point multiplier in a more aggressive BICMOS technology. The ability to construct very small high performance multipliers provides many other interesting possibilities. A double precision IEEE multiplier could be placed on the same chip with an existing RISC processor. Multiplication intensive applications, such as DSP or graphics, could benefit significantly from several high performance multipliers on the same chip. A single very high throughput multiplier, or several multipliers working in parallel on the same chip,

could open up new possibilities such as single chip video signal processors. Further investigation into high order counter multipliers should also continue as it may produce a better tradeoff in particular if they have to be implemented in BICMOS.

REFERENCES

[1] Williams, S. B., Bell Telephone Laboratories, "Relay Computing System," Proc. Symp. on Large-Scale Computing Machinery, 1947.

[2] Alt, F. L., Bell Telephone Laboratories, "Computing Machine," Math. Comp., Vol. 3, No. 21, Jan. 1948.

[3] Campell, R.V.D., "Mark II Computer," Proc. Symp. on Large-Scale Computing Machinery, 1950.

[4] "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE std 754, New York, The Institute of Electrical and Electronics Engineers, Inc., August 12, 1985.

[5] J. J. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," Computer Magazine Vol. 13, No. 1, January 1980.

[6] M. Santario, "Rounding Algorithm for IEEE Multipliers," Proc. 9th IEEE Symp. Computer Arithmetic, 1989.

[7] C. S. Wallace, "A Suggestion for Fast Multipliers," IEEE Trans. on Electronic Computers, Vol. EC-13, February, 14-17, 1964.

[8] J. Sklansky, "Conditional Sum Addition Logic," Trans. IRE, Vol. EC-9. No. 2, June, 226-230, 1960.

[9] A. D. Booth, "A Signed Binary Multiplication Technique", Qt. J. Mech. Appl. Math., Vol. 4, 1951.

[10] S. Nakamura, "Algorithms for Iterative Arrray Multiplication", IEEE Trans. on Computers, Vol. c-35, No. 8, August 1986.

[11] L. Dadda, "Some Schemes for Parallel Multipliers", Alta Frequenza, Vol. 34, No. 5, May 1965.

[12] D. Zuras, "Balanced Delay Trees and Combinational Division in VLSI", IEEE Journal of Solid-State Circuits, Vol. sc-21, No. 5, October 1986.

[13] A. Gamal, "A CMOS 32b Wallace Tree Multiplier-Accumlator," ISSCC Digest of Technical Papers, February 1986.

[14] Y. Harata, "A High-Speed Multiplier Using a Redundant Binary Adder Tree," IEEE Journal of Solid State Circuits, Vol. sc-22, No. 1, February 1987.

[15] N. P. Jouppi, "MultiTian: Four Architecture Papers," WRL Research Report, April 1988.

[16] M. Nagamatusu, "A 32b CMOS Multiplier with an Improved Parallel Structure," IEEE Journal of Solid-State Circuits, Vol. 25, No. 2 April 1990.

[17] C. C. Foster and F. D. Stockton, "Counting responders in an associative memory," IEEE Trans. on Computers, Vol. c-20, December 1971.

[18] E. E. Swartzlander, "Parallel Counters," IEEE Trans. on Computers, Vol. c-22, January 1973.

[19] M. Mayur, " High Speed Multiplier Design Using Multi-input Counter and Compressor Circuits," Proc. 9th IEEE Symb. Computer Arithmetic, 1991.

[20] P. J. Song, "Circuit and Architecture Trade-offs for High-Speed Multiplication," IEEE Journal of Solid-State Circuits, Vol. 26, No. 9, September 1991.

[21] C. A. Mead and L. A. Conway, Introduction to VLSI Systems Reading, Addison-Wesley, 1980.

[22] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," IEEE Trans. on Computers, Vol. c-31, No. 12, December 1983.

APPENDIX
1A

Behavioral Model for a Floating Point Multiplier

This Appendix provides the behavioral VHDL Model for the new floating point multiplier algorithm described in chapter 3 . The simulation result is shown at the end.

```
--------------------------------------------------------------------------------

                                fmult1.vhd

--------------------------------------------------------------------------------

entity float_multiplier is
        port(a,b: bit_vector(24 downto 1); q:inout bit_vector(22 downto 1); prod: out
                bit_vector(24 downto 1); ab:inout bit_vector(26 downto 1);temp_prod :out
                bit_vector(24 downto 1); S,L,R: out bit; counta,countb : out integer);
end float_multiplier;

architecture behavior of float_multiplier is
        signal carry_in: bit:='0';

        begin

p1:     process(a,b)

                variable ab2,temprod: bit_vector(24 downto 1);
                        :="000000000000000000000000";
                variable q1:bit_vector(23 downto 1);
                        ="00000000000000000000000";
                variable count,count1,count2: integer:=0;
                variable m:integer;
                variable carry: bit_vector(24 downto 1);
                variable start : integer;
                variable temp,temp1: bit;
                variable rbit,sticky,Lbit: bit;
                variable s1,c1: bit_vector(26 downto 1);
                constant r1: bit:='1';

                begin
```

```
        start := 0;

loop1:
        for j in 1 to 24 loop
        if(b(j)='1') then
loop2:
        for i in 1 to 24 loop
        if(i=1) then
                carry(i):= (ab2(i) and a(i)) or (ab2(i) and
                carry_in) or (a(i) and carry_in);
                ab2(i):= (ab2(i) xor a(i)) xor carry_in;

        else
                temp:=ab2(i);
                ab2(i):= (ab2(i) xor a(i)) xor carry(i-1);
                carry(i) := (temp and a(i)) or (temp and
                carry(i-1)) or (a(i) and carry(i-1))
                end if;
                end loop loop2;

                else
                        count:=count+1;
                end if;

-- shift right
                m:=j;

loop3:

                while(m < 24) loop
loop4:

        for l in 1 to 22  loop
                q1(l):= q1(l+1);
                end loop loop4;
```

```
                q1(23):=ab2(1);
    loop5:
        for k in 1 to 23 loop
         ab2(k):= ab2(k+1);
        end loop loop5;

        ab2(24):=carry(24);
        carry(24):='0';
        m:=m+23;
        end loop loop3;
        end loop loop1;


    loop6:
        for i in 1 to 22 loop
        q(i)<=q1(i) after 5ns;
        end loop loop6;

        ab <= carry(24)&ab2&q1(23) after 5ns;
        s1:=carry(24)&ab2&q1(23);

-- Round
-- First check overflow bit

        if(s1(26)='1') then
    loop7:

        for i in 2 to 26 loop
            if(i=2) then
        rbit:= s1(2);
        Lbit:= s1(3);
        c1(i):= s1(i) and r1;
        s1(i):= s1(i) xor r1;
```

```
            else
            temp1:=s1(i);
            s1(i):= s1(i) xor c1(i-1);
            c1(i):= temp1 and c1(i-1);
            end if;
            end loop loop7;

             else
loop8:

            for j in 1 to 26 loop

                    if(j=1) then
             rbit:= s1(1);
            Lbit:= s1(2);
            c1(j):= s1(j) and r1;
            s1(j):= s1(j) xor r1;

            else
            temp1:= s1(j);
            s1(j):= s1(j) xor c1(j-1);
            c1(j):= temp1 and c1(j-1);
            end if;
            end loop loop8;
            end if;

-- Compute Sticky Bit

        loop9:

            for i in 1 to 24 loop
                    if(a(i)='0')then
            count1:=count1+1;
                    else
                            exit loop9 ;
```

```
            end if;
        end loop loop9;

    loop10:

        for i in 1 to 24 loop
            if(b(i)='0')then
        count1:=count1+1;
        else
        exit loop10;
        end if;
        end loop loop10;

        if(carry(24)='1') then
        count2:=23;
        else
            count2:=22;
        end if;

        if(count1 >= count2) then
            sticky:='0';
        else
            sticky:='1';
        end if;


-- Normalization

        if(s1(26)='1') then

    loop11:
        for i in 2 to 25 loop
        s1(i):= s1(i+1);
        end loop loop11;
        end if;
```

```
loop12:
        for i in 2 to 25 loop
        temprod(i-1):=s1(i);
        end loop loop12;

loop13:
        for i in 1 to 24 loop
        temp_prod(i)<= temprod(i) after 5ns;
        end loop loop13;

        if(rbit='1' and temprod(1)='1')then

            if(sticky='0') then
          temprod(1):='0';
            end if;
            end if;


loop14:
        for i in 1 to 24 loop
        prod(i) <= temprod(i) after 5ns;
        end loop loop14;

            R <= rbit after 5ns;
            S <= sticky after 5ns;
            L <= Lbit  after 5ns;
            counta<= count1 after 5ns;
            countb <= count2 after 5ns;

loop15:

            for i in 1 to 24 loop
                    ab2(i):='0';
            end loop loop15;
```

```
        loop16:
                        for i in 1 to 23 loop
                                q1(i):='0';
                        end loop loop16;
                        carry(24):='0';

                        count1:=0;
                        count2:=0;
        end process;

end behavior;
```

---

tb_fmultp.vhd

```
entity tb_fmultp1 is
end tb_fmultp1;

architecture behavior of tb_fmultp1 is

        component b_multp
                port(a,b:bit_vector(24 downto 1); q:inout bit_vector(22 downto 1); prod:
                        out bit_vector(24 downto 1); ab: inout bit_vector(26 downto
                        1);temp_prod:out bit_vector(24 downto 1); S,L,R:out bit;
                        counta,countb: out integer);
        end component;

        signal a1,b1:bit_vector(24 downto 1);
        signal q1:bit_vector(22 downto 1);
        signal ab1: bit_vector(26 downto 1);
        signal temp_prod,fprod: bit_vector (24 downto 1);
        signal S,R,L:bit;
        signal counta,countb: integer;

for x1: b_multp use entity work.float_multiplier(behavior);
```

```
begin

x1: b_multp  port map(a1,b1,q1,fprod,ab1,temp_prod,S,L,R,counta,countb);

a1<= "10100000000000000000000" after 0ns, "10100000000000000000000"
     after 40ns,"11111111111111111111111" after 80ns "10000000000000-
     -0000000001" after 100ns, "00000000000000000000000" after 120ns;

b1<= "10000000000000000000010" after 0ns, "10000000000000000000110"
     after 40ns,"10000000000000000000001" after 80ns, "11111111111111-
     - 1111111110" after 100ns, "00000000000000000000000" after 120ns;

end behavior;
```

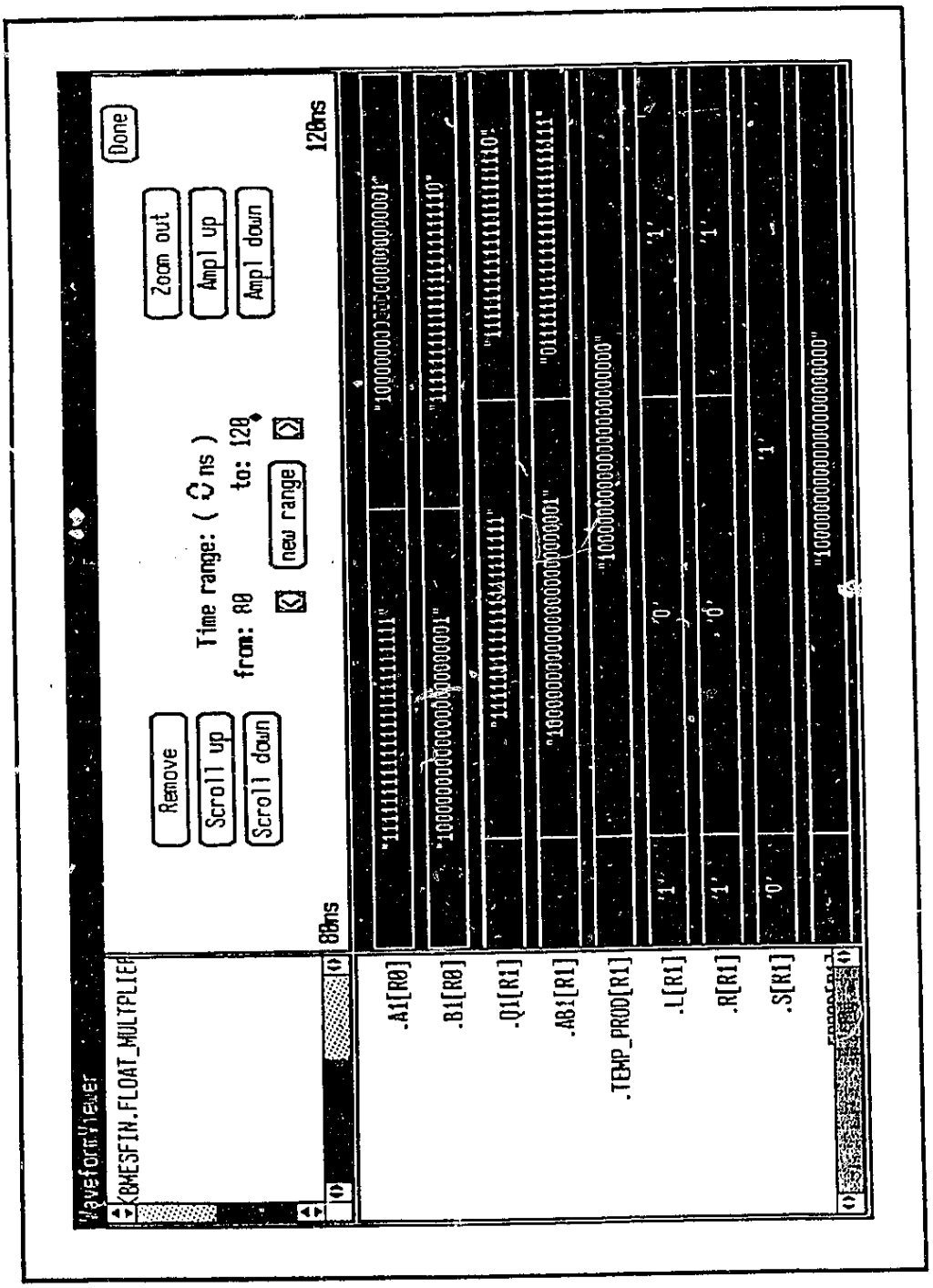


Figure 1A.1: VHDL Waveforms for a Floating Point Multiplier

APPENDIX
2A

Architectural Model for a Floating Point Multiplier

```
--------------------------------------------------------------------------------

                                exponent.vhd

--------------------------------------------------------------------------------

Library XL;
use XL.XL_STD.all;
Library Mgates;
use Mgates.all;

entity exp_adder1 is
        generic(delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8: in time);
        port (clk1: bit;a1,b1:in bit_vector(8 downto 1); V,V_bar: in bit;CR_temp :
                inout bit_vector(8 downto 1); CRR:inout bit_vector(8 downto
                1);CR1: inout bit_vector(8 downto 1); E11,E22,E33:inout bit;
                ain,bin,output:out integer);
end exp_adder1;

    architecture structure of exp_adder1 is

        component Fadd
                generic (delay: in time);
                port (a,b,c: bit; sum, carry: inout bit);
        end component;
        component and_2
                generic (delay: in time);
                port (a,b:bit; c: out bit);
        end component;
        component and_3
                generic (delay: in time);
                port (a,b,c: bit; d:out bit);
        end component;
        component xor_2
```

```
        generic (delay: in time);
        port (a,b:bit; c: out bit);
end component;
component or_3
        generic (delay: in time);
        port (a,b,c:bit; d: out bit);
end component;
component or_2
        generic (delay: in time);

        port (a,b:bit; c:out bit);
end component;
component inverter
        generic (delay: in time);
        port(a: bit; b: out bit);
end component;
component and_7
        port(CB: in bit_vector(8 downto 1); E11: out bit);
end component;
component d_latchN
        generic (delay: in time);
        port(clk1,x:in bit; y: out bit);
end component;
component d_latchP
        generic (delay: in time);
        port(clk1,x: in bit; y:out bit);
end component;
component detector
        generic (delay: in time);
        port(E2,E3: bit; CB: in bit_vector(8 downto 1);CR_temp11: in
                bit_vector(8 downto 1); CR_out: out bit_vector(8 downto
                1));
end component;
signal a,b,m1,CR,temp500, temp501,temp520,temp521,temp522:
        bit_vector(8 downto 1):="00000000";
```

```
        signal g,gg,p,gp: bit_vector(10 downto 1):="0000000000";
        signal g1,gg1,p1,gp1: bit_vector(10 downto 1):="0000000000";
        signal  sum_temp100,sum_tempe0,sum_tempe1,carry_temp100,
                carry_tempe0,carry_tempe1: bit_vector(8 downto 1):="00000000";
        signal temp1,temp2,temp3,carry_temp3: bit_vector(4 downto 1):="0000";
        signal temp4,temp5e0,temp5e1,temp6e0,temp6e1,carry_temp4e0,
                carry_temp4e1: bit_vector(4 downto 1):="0000";
        signal c0,OV,V1: bit:='1';
        signal carry_tempx, carry_tempxx, carry_tempxxx,E1_bar:bit:='0';
        signal E33e0,E33e1,E33_temp1,E33_temp2,E33_temp,E22_temp,
                E11_temp,UN,E2_bar,V_bar1: bit:='0';
        signal CR_out1: bit_vector(8 downto 1):="00000000";
        signal temp10, temp11,temp12,temp13,temp14,temp15,temp16:bit:='0';
        signal temp20, temp21,temp22,temp23,temp24,temp25,temp26: bit:='0';
        signal temp30,temp31,temp32,temp33,temp34,temp35,temp36: bit:='0';
        signal temp510, temp511,temp512,F1,F2,F1_bar,F2_bar: bit:='0';
        signal msb_cB: bit;

        for all: Fadd use entity work.Fadd(structure);
        for all: and_2 use entity work.and_2(behavior);
        for all: or_2 use entity work.or_2(behavior);
        for all: or_3 use entity work.or_3(behavior);
        for all: xor_2 use entity work.xor_2(behavior);
        for all: inverter use entity work.inverter(behavior);
        for all:detector use entity work.detector(behavior);
        for all:and_7 use entity work.and_7(behavior);
        for all: and_3 use entity work.and_3(behavior);
        for all: d_latchN use entity work.d_latchN(behavior);
        for all: d_latchP use entity work.d_latchP(behavior);

    begin

--      CA + CB -127

G2000:      for i in 1 to 8 generate
```

```
d500: d_latchN
        generic map(delay2)
        port map(clk1, a1(i), a(i));
d501: d_latchN
        generic map(delay2)
        port map(clk1, b1(i), b(i));
        end generate;
inv500: inverter
        generic map(delay5)
        port map(b(8), msb_cB);
x500: Fadd
        generic map(delay4)
        port map(a(1),b(1),c0,sum_temp100(1), carry_temp100(1));
x501: Fadd
        generic map(delay4)
        port map(a(2), b(2), carry_temp100(1),sum_temp100(2),
                carry_tempx);

--      especial circuit  to generate carry 2 (c2)

u500: and_2
        generic map(delay1)
        port map(a(1), b(1),temp30);
u501: and_2
        generic map(delay1)
        port map(a(1), c0,temp31);
u502: and_2
        generic map(delay1)
        port map(b(1), c0,temp32);
k500:  or_3
        generic map(delay7)
        port map(temp30, temp31, temp32, temp33);
u503: and_2
        generic map(delay1)
```

```
            port map(a(2), b(2), temp34);
        u504: and_2
            generic map(delay1)
            port map(a(2), temp33, temp35);

        u505: and_2
            generic map(delay1)
            port map(temp33, b(2), temp36);
        k501: or_3
            generic map(delay7)
            port map(temp34, temp35,temp36, carry_temp100(2));

        u506: and_2
            generic map(delay1)
            port map(a(8), msb_CB, g(8));
        z500: xor_2
            generic map(delay2)
            port map(a(8), msb_CB, p(8));

G500: for i in 3 to 7 generate
        u507: and_2
            generic map(delay1)
            port map(a(i), b(i),g(i));
        z501: xor_2
            generic map(delay2)
            port map(a(i), b(i), p(i));
            end generate;

G501: for i in 1 to 3 generate
        u508: and_2
            generic map(delay1)
            port map(p(2*i+2), g(2*i+1),temp1(i));
        y500: or_2
            generic map(delay3)
            port map(g(2*i+2),temp1(i),gg(2*i+2));
```

```
        u509: and_2
              generic map(delay1)
              port map(p(2*i+2), p(2*i+1), gp(2*i+2));
              end generate;


    G502:  for i in 1 to 3 generate
              u510: and_2
                    generic map(delay1)
                    port map(gp(2*i+2), carry_temp100(2*i),carry_temp3(i));
               y501: or_2
                    generic map(delay3)
                    port map(carry_temp3(i),gg(2*i+2),carry_temp100(2*i+2));
                    end generate;

    G503: for i in 1 to 3 generate
              z502: xor_2
                    generic map(delay2)
                    port map(p(2*i+1), carry_temp100(2*i), sum_temp100(2*i+1));
              u511: and_2
                    generic map(delay1)
                    port map(p(2*i+1), carry_temp100(2*i), temp2(i));
              y502: or_2
                    generic map(delay3)
                    port map(temp2(i), g(2*i+1),temp3(i));
              z503: xor_2
                    generic map(delay2)
                    port map(p(2*i+2), temp3(i),sum_temp100(2*i+2));
                    end generate;

              CR_temp <= sum_temp100;

              E22_temp <= carry_temp100(8);

-- for v=0
```

```
x502: Fadd
        generic map(delay4)
        port map(sum_temp100(1),m1(1),V_bar1,sum_tempe0(1),
        carry_tempe0(1));
x503: Fadd
        generic map(delay4)
        port map(sum_temp100(2),m1(2),carry_tempe0(1),
        sum_tempe0(2),carry_tempxx);
u512: and_2
        generic map(delay1)
        port map(sum_temp100(1), m1(1),temp10);
u513: and_2
        generic map(delay1)
        port map(sum_temp100(1), V_bar1,temp11);
u514: and_2
        generic map(delay1)
        port map(m1(1), V_bar1,temp12);
k502:  or_3
        generic map(delay7)
        port map(temp10, temp11, temp12, temp13);
u515: and_2
        generic map(delay1)
        port map(sum_temp100(2), m1(2), temp14);
u516: and_2
        generic map(delay1)
        port map(sum_temp100(2), temp13, temp15);
u517: and_2
        generic map(delay1)
        port map(temp13, m1(2), temp16);
k503: or_3
        generic map(delay7)
        port map(temp14, temp15,temp16, carry_tempe0(2));

G504:for i in 3 to 8 generate
```

```
        u518: and_2
                generic map(delay1)
                port map(sum_temp100(i), m1(i),g1(i));
        z504: xor_2
                generic map(delay2)
                port map(sum_temp100(i), m1(i), p1(i));
                end generate;


G505:  for i in 1 to 3 generate

        u519: and_2
                generic map(delay1)
                port map(p1(2*i+2), g1(2*i+1),temp4(i));
        y503: or_2
                generic map(delay3)
                port map(g1(2*i+2),temp4(i),gg1(2*i+2));
        u520: and_2
                generic map(delay1)
                port map(p1(2*i+2), p1(2*i+1), gp1(2*i+2));
                end generate;

G506:  for i in 1 to 3 generate
        u521: and_2
                generic map(delay1)
                port map(gp1(2*i+2), carry_tempe0(2*i),carry_temp4e0(i));
         y504: or_2
                generic map(delay3)
                port map(carry_temp4e0(i),gg1(2*i+2),carry_tempe0(2*i+2));
                end generate;

        E33e0 <= carry_tempe0(8);

G507: for i in 1 to 3 generate
        z505: xor_2
```

```
            generic map(delay2)
            port map(p1(2*i+1), carry_tempe0(2*i), sum_tempe0(2*i+1));
        u522: and_2
            generic map(delay1)
            port map(p1(2*i+1), carry_tempe0(2*i), temp5e0(i));
        y505: or_2
            generic map(delay3)
            port map(temp5e0(i), g1(2*i+1),temp6e0(i));

        z506: xor_2
            generic map(delay2)
            port map(p1(2*i+2), temp6e0(i),sum_tempe0(2*i+2));
            end generate;

-- for v=1
        x504: Fadd
            generic map(delay4)
            port map(sum_temp100(i),m1(1),V1,sum_tempe1(1),
                carry_tempe1(1));
        x505: Fadd
            generic map(delay4)
            port map(sum_temp100(2), m1(2),carry_tempe1(1),
                sum_tempe1(2),carry_tempxxx);
        u523: and_2
            generic map(delay1)
            port map(sum_temp100(1), m1(1),temp20);
        u524: and_2
            generic map(delay1)
            port map(sum_temp100(1),V1,temp21);
        u525: and_2
            generic map(delay1)
            port map(m1(1), V1,temp22);
        k504: or_3
            generic map(delay7)
            port map(temp20, temp21, temp22, temp23);
```

```
        u526: and_2
                generic map(delay1)
                port map(sum_temp100(2), m1(2), temp24);
        u527: and_2
                generic map(delay1)
                port map(sum_temp100(2), temp23, temp25);
        u528: and_2
                generic map(delay1)
                port map(temp23, m1(2), temp26);

        k505: or_3
                generic map(delay7)
                port map(temp24, temp25,temp26, carry_tempe1(2));

G508:   for i in 1 to 3 generate
        u529: and_2
                generic map(delay1)
                port map(gp1(2*i+2), carry_tempe1(2*i),carry_temp4e0(i));
        y506: or_2
                generic map(delay3)
                port map(carry_temp4e0(i),gg1(2*i+2),carry_tempe1(2*i+2));
                end generate;

        E33e1 <= carry_tempe1(8);
G509:   for i in 1 to 3 generate
        z507: xor_2
                generic map(delay2)
                port map(p1(2*i+1), carry_tempe1(2*i), sum_tempe1(2*i+1));
        u530: and_2
                generic map(delay1)
                port map(p1(2*i+1), carry_tempe1(2*i), temp5e0(i));
        y507: or_2
                generic map(delay3)
                port map(temp5e0(i), g1(2*i+1),temp6e0(i));
        z508: xor_2
```

```
                generic map(delay2)
                port map(p1(2*i+2), temp6e0(i),sum_tempe1(2*i+2));
                end generate;

        G510: for i in 1 to 8 generate
                u531: and_2
                        generic map(delay1)
                        port map(V_bar, sum_tempe0(i), temp500(i));
                u532: and_2
                        generic map(delay1)
                        port map(V, sum_tempe1(i), temp501(i));

                y508: or_2
                        generic map(delay3)
                        port map(temp500(i), temp501(i), CRR(i));
                        end generate;
                u533: and_2
                        generic map(delay1)
                        port map(V_bar, E33e0, E33_temp1);
                u544: and_2
                        generic map(delay1)
                        port map(V, E33e1, E33_temp2);
                y509: or_2
                        generic map(delay3)
                        port map(E33_temp1, E33_temp2, E33_temp);
--      CRR <= sum_temp1  ;

-- underflow and Overflow detector

                a7_500: and_7
                        port map(b, E11_temp);
                u545: and_2
                        generic map(delay1)
                        port map(E11_temp, E22_temp, temp510);
                in501: inverter
```

```
            generic map(delay5)
            port map(E11_temp, E1_bar);
        in503: inverter
            generic map(delay5)
            port map(E22_temp, E2_bar);
        a500: and_3
            generic map(delay8)
            port map(E11_temp, E2_bar,E33_temp, temp511);
        a5000: and_3
            generic map(delay8)
            port map(E1_bar, E22_temp, E33_temp, temp512);

        u546:  and_2
            generic map(delay1)
            port map(E1_bar, E2_bar, F2);
        K508:  or_3
            generic map (delay3)
            port map(temp510, temp511,temp512, F1);
        in504: inverter
            generic map(delay5)
            port map(F1, F1_bar);
        in505: inverter
            generic map(delay5)
            port map(F2, F2_bar);

    G511: for i in 1 to 8 generate
        a501: and_3
            generic map(delay8)
            port map( F1_bar, F2_bar, CRR(i), temp520(i));
        a502: and_3
            generic map(delay8)
            port map(UN,F1_bar, F2, temp521(i));
        a503: and_3
            generic map(delay8)
            port map(OV,F1,F2_bar, temp522(i));
```

```
            k505: or_3
                  generic map(delay7)
                  port map(temp520(i), temp521(i),temp522(i),CR(i));
                  end generate;

      G5000: for i in 1 to 8 generate
            d503: d_latchP
                  generic map(delay2)
                  port map(clk1, CR(i), CRl(i));
                  end generate;
            d504: d_latchP
                  generic map(delay2)
                  port map(clk1, E22_temp, E22);
            d505: d_latchP
                  generic map(delay2)
                  port map(clk1, E11_temp, E11);
            d506: d_latchP
                  generic map(delay2)
                  port map(clk1, E33_temp, E33);

            output <= to_integer(CRl);
            ain <= to_integer(a);
            bin <= to_integer(b);

--       CR <= CR_out1;
end structure;
```

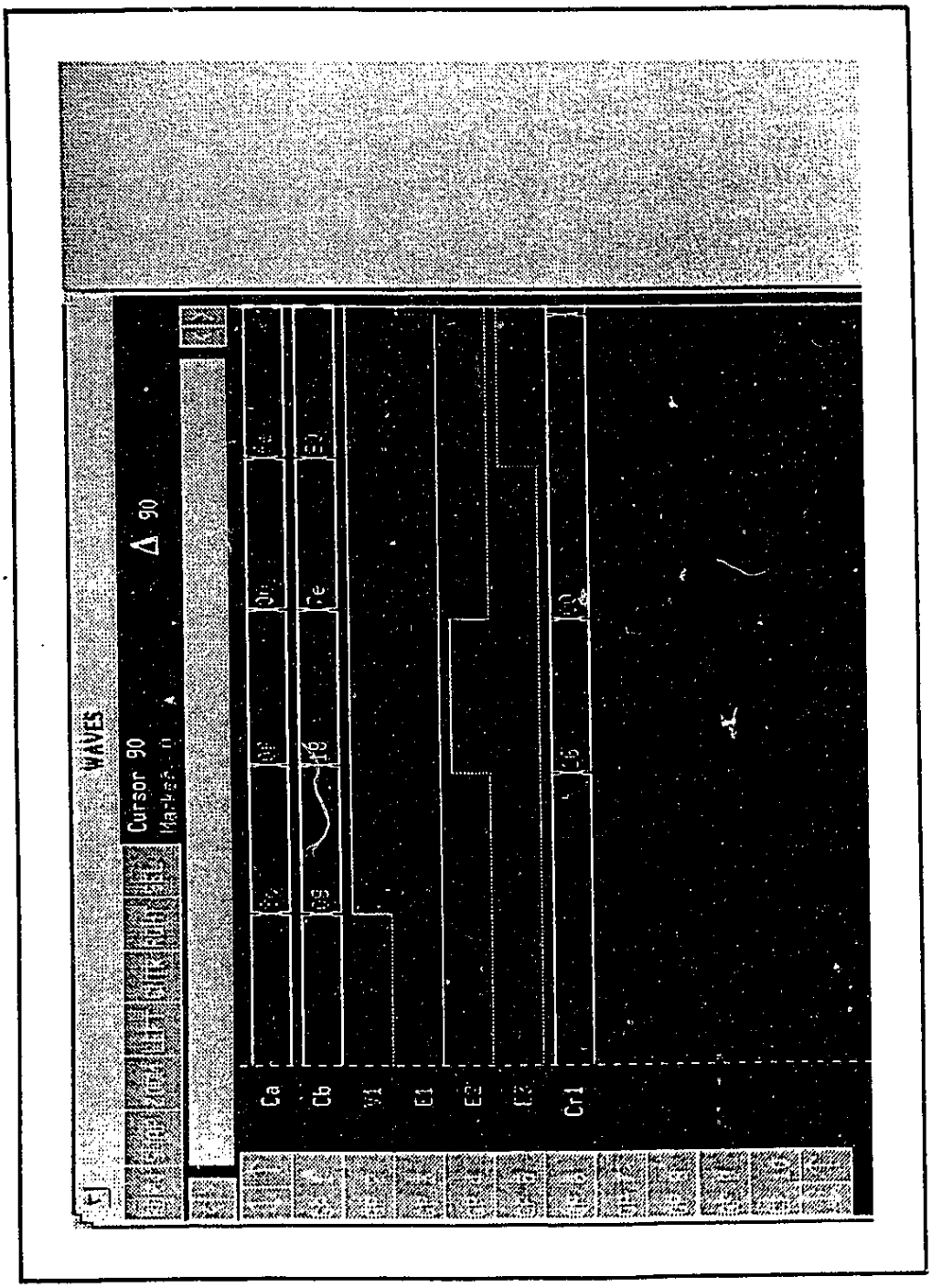


Figure 2A.1: VHDL Waveforms for Exponent Data Path

---

fmultp2.vhd

```
Library XL;
use XL.XL_STD.all;
Library Mgates;
Use Mgates.all;
Library Madder;
Use Madder.all;

entity float_multp is
        generic(delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8: in time);
        port(clk1: bit; a1,b1:in bit_vector(7 downto 0);ca11,cb11: bit_vector(8 downto 1);
                fproduct, CR1: out bit_vector(8 downto 1); ain1,bin1, CA_I1, CB_I1:out
                integer;V,L,R,S,Res:inout bit);
end float_multp;

        architecture structure of float_multp is

                component and_2
                        generic(delay : in time);
                        port(a,b: in bit; c: out bit);
                end component;
                component and_21
                        generic(delay: in time);
                        port(a: bit; c:inout bit);
                end component;
                component xor_2
                        generic (delay: in time);
                        port(a,b: bit; c: out bit);
                end component;
                component or_2
                        generic (delay: in time);
                        port(a,b:bit; c: out bit);
```

```
end component;

component FADD
        generic(delay: in time);
        port(a,b,c: bit; sum,carry:out bit);
end component;
component Hadder
        generic(delay: in time);
        port(x,y:in bit; sum,carry:out bit);
end component;
component Tenbit_adder
        generic(delay1,delay2,delay3,delay4: in time);
        port(a,b: bit_vector(10 downto 1); c0: in bit;  sum: out bit_vector(11
                downto 1));
end component;
        component canca
        port(producta: in bit_vector(11 downto 1); suma: in bit_vector(14
                downto 0); productb: inout bit_vector(15 downto 0));
end component;
component d_latchN
        generic(delay:in time);
        port(clk1,x: in bit; y:out bit);
end component;
component d_latchP
        generic(delay: in time);
        port(clk1,x:in bit; y: out bit);
end component;
component and_3
        generic (delay: in time);
        port(a,b,c: bit; D: out bit);
end component;
component and_7
        port (CB: in bit_vector(8 downto 1); E11:out bit);
end component;
component R_Shifter
```

```
            generic (delay: in time);
            port(v: bit; input: in bit_vector(10 downto 1); output: out
                bit_vector(8 downto 1));
        end component;
        component V_detector
            port(a: bit_vector(10 downto 1); b:out bit);
        end component;
        component RL_detector
            port(a: bit_vector(10 downto 1); b,c,d:out bit);
        end component;
        component inverter
            generic (delay: in time);
            port (a: bit; b: out bit);
        end component;
        component or_3
            generic (delay: in time);
            port (a,b,c:bit; d: out bit);
        end component;
        component restorer
            generic (delay: in time);
            port(a: bit; b: inout bit);
        end component;
        component sticky
            generic (delay: in time);
            port(v1,a,b: bit; c: out bit);
        end component;

signal c01: bit:='0';
signal Rin,RV2 :bit:='1';
signal RV1: bit:='0';
signal carry: bit_vector(14 downto 0);
signal GG1,GG2,GG3,GG4,GG5,GG6,GG7,GG8: bit_vector(7 downto
    0):="00000000";
signal sum: bit_vector(15 downto 0):="000000000000000";
signal tsa2,tsa3,tsa4,tsa5,tsa6,tsa9,tsa10: bit:='0';
```

```
signal tca1,tca2,tca3,tca4,tca5,tca6,tca9,tca10:bit:='0';
signal tsa7,tsa8,tca7,tca8: bit_vector(1 downto 0):="00";
signal tsb3,tcb3,tcb2,tsb4,tsb5,tsb7,tsb9,tsb10,tsb11:bit:='0';
signal tcb4,tcb5,tcb7,tcb9,tcb10,tcb11:bit:='0';
signal tsb6,tcb6,tsb8,tcb8:bit_vector(1 downto 0):="00";
signal tsc4,tsc5,tsc6,tsc8,tsc10,tsc11,tsc12:bit:='0';
signal tcc3,tcc4,tcc5,tcc6,tcc8,tcc10,tcc11,tcc12:bit:='0';
signal tsc9,tcc9,tsc7,tcc7: bit_vector(1 downto 0):="00";
signal tsd5,tsd6,tsd7,tsd8,tsd9,tsd10,tsd11,tsd12,tsd13,tsd14: bit:='0';
signal tcd4,tcd5,tcd6,tcd7,tcd8,tcd9,tcd10,tcd11,tcd12,tcd13,tcd14: bit:='0';
signal tsc14,tsc15:bit :='0';

-- adder
signal a1l,b1L: bit_vector(7 downto 0):="00000000";
signal suml: bit_vector(14 downto 0):="000000000000000";
signal carry_templv0, sum_templv0,carry_templv1,sum_templv1: bit_vector(10 downto
        1):="0000000000";
signal g,gg,p,gp: bit_vector(10 downto 1):="0000000000";
signal sum_tempv0,carry_tempv0,sum_tempv1,carry_tempv1: bit_vector(10 downto
        1):="0000000000";
signal tempv11,tempv12,tempv13,carry_tempv11: bit_vector(4 downto 1):="0000";
signal tempv21,tempv22,tempv23,carry_tempv21: bit_vector(4 downto 1):="0000";
signal tempv14,tempv24,carry_tempV, carry_tempvv: bit:='0';
signal tempp14,z1, Res_bar,out_temp, out_templ:bit:='0';
signal temp70, temp80,temp90,temp100,s_temp:bit:='0';
signal V_bar, S_bar,R_bar,R_temp: bit:='0';
signal fproduct_temp: bit_vector(8 downto 1):="00000000";
signal temp110, temp120 : bit_vector(10 downto 1);
signal T_sum, T_carry :bit_vector(10 downto 1):="0000000000";
signal Product1,Product0: bit_vector(10 downto 1):="0000000000";
signal temp210, temp200: bit_vector(8 downto 1):="00000000";
signal fproduct_test: bit_vector(8 downto 1):="00000000";

-- exponent signals
```

```
signal m1,ca1,cb1,CR,CR_temp, CRR,temp500, temp501,temp520,temp521,temp522:
        bit_vector(8 downto 1):="00000000";
signal Product,ge,gge,pe,gpe: bit_vector(10 downto 1):="0000000000";
signal g11,gg11,p1,gp1: bit_vector(10 downto 1):="0000000000";
signal sum_temp100,sum_tempe0,sum_tempe1,carry_temp100,carry_tempe0,
        carry_tempe1: bit_vector(8 downto 1):="00000000";
signal temp1e,temp2e,temp3e,carry_temp3e: bit_vector(4 downto 1):="0000";
signal temp4,temp5e0,temp5e1,temp6e0,temp6e1,carry_temp4e0,carry_temp4e1:
        bit_vector(4 downto 1):="0000";
signal c0,OV,V1: bit:='1';
signal carry_tempx, carry_tempxx, carry_tempxxx,E1_bar:bit:='0';
signal E33e0,E33e1,E33_temp1,E33_temp2,UN,E2_bar,V_bar1: bit:='0';
signal CR_out1: bit_vector(8 downto 1):="00000000";
signal temp10, temp11,temp12,temp13,temp14,temp15,temp16:bit:='0';
signal temp20, temp21,temp22,temp23,temp24,temp25,temp26: bit:='0';
signal temp30,temp31,temp32,temp33,temp34,temp35,temp36: bit:='0';
signal temp510, temp511,F1,F2,F1_bar,F2_bar,E1: bit:='0';
signal msb_cB: bit;
signal E22, E33:bit;

for all: and_2 use entity work.and_2(behavior);
for all: xor_2 use entity work.xor_2(behavior);
for all: or_2 use entity work.or_2(behavior);
for all: or_3 use entity work.or_3(behavior);
for all: d_latchN use entity work.d_latchN(behavior);
for all: d_latchP use entity work.d_latchP(behavior);
for all: FADD use entity work.FADD(structure);
for all: hadder use entity work.hadder(structure);
for all: inverter use entity work.inverter(behavior);
for all: sticky use entity work.sticky(behavior);
for all: Restorer use entity work.Restorer(behavior);
for all: V_detector use entity work.V_detector(behavior);
for all: RL_detector use entity work.RL_detector(behavior);
for all: and_3 use entity work.and_3(behavior);
for all: R_shifter use entity work.R_shifter(behavior);
```

```
for all: and_21 use entity work.and_21(behavior);
for all: and_7 use entity work.and_7(behavior);
for cc1: canca use entity Mgates.cancation(behavior);

        begin

        G0: for i in 0 to 7 generate
                d0: d_latchN
                        generic map(delay2)
                        port map(clk1, a1(i), a11(i));
                end generate;

        G141: for i in 0 to 7 generate
                d1: d_latchN
                        generic map(delay2)
                        port map(clk1, b1(i), b11(i));
                end generate;

        G1: for i in 0 to 7 generate
                u0: and_2
                        generic map(delay1)
                        port map(a11(i),b11(0),GG1(i));
                end generate;

        G2: for i in 0 to 7 generate
                u1: and_2
                        generic map(delay1)
                        port map(a11(i),b11(1),GG2(i));
                end generate;

        G3: for i in 0 to 7 generate
                u2: and_2
                        generic map(delay1)
                        port map(a11(i),b11(2),GG3(i));
                end generate;
```

```
G4: for i in 0 to 7 generate
        u3: and_2
                generic map(delay1)
                port map(a11(i),b11(3),GG4(i));

                end generate;

G5: for i in 0 to 7 generate
        u4: and_2
                generic map(delay1)
                port map(a11(i),b11(4),GG5(i));
        end generate;

G6: for i in 0 to 7 generate
        u5: and_2
                generic map(delay1)
                port map(a11(i),b11(5),GG6(i));
        end generate;

G7: for i in 0 to 7 generate
        u6: and_2
                generic map(delay1)
                port map(a11(i),b11(6),GG7(i));
        end generate;

G8: for i in 0 to 7 generate
        u7: and_2
                generic map(delay1)
                port map(a11(i),b11(7),GG8(i));
        end generate;

        HA1a: hadder
                generic map(delay3)
                port map(GG1(1), GG2(0), sum(1),tca1);
```

```
y20: or_2
        generic map(delay4)
        port map(GG1(0), sum(1), temp70);
FA2a: FADD
        generic map(delay3)
        port map(GG2(1),GG1(2),GG3(0),tsa2,tca2);

FA3a: FADD
        generic map(delay3)
        port map(GG3(1),GG1(3),GG4(0),tsa3,tca3);

FA4a: FADD
        generic map(delay3)
        port map(GG4(1),GG1(4),GG5(0),tsa4,tca4);
FA5a: FADD
        generic map(delay3)
        port map(GG5(1),GG6(0),GG1(5),tsa5,tca5);
FA6a: FADD
        generic map(delay3)
        port map(GG4(3),GG1(6),GG7(0),tsa6,tca6);
FA7a: FADD
        generic map(delay3)
        port map(GG3(5),GG6(2),GG2(6),tsa7(0),tca7(0));
FA7a1: FADD
        generic map(delay3)
        port map(GG7(1),GG8(0),GG1(7),tsa7(1),tca7(1));
FA8a: FADD
        generic map(delay3)
        port map(GG7(2),GG2(7),GG8(1),tsa8(0),tca8(0));
FA8a1: FADD
        generic map(delay3)
        port map(GG4(5),GG6(3),GG3(6),tsa8(1),tca8(1));
FA9a: FADD
        generic map(delay3)
```

```
        port map(GG7(3),GG3(7),GG8(2),tsa9,tca9);
FA10a: FADD
        generic map(delay3)
        port map(GG8(3),GG4(7),GG7(4),tsa10,tca10);
HA2b: hadder
        generic map(delay3)
        port map(tca1,tsa2,sum(2),tcb2);
y21: or_2
        generic map(delay4)
        port map(temp70, sum(2), temp80);
FA3b: FADD
        generic map(delay3)
        port map(GG2(2),tca2,tsa3,tsb3,tcb3);
FA4b: FADD
        generic map(delay3)
        port map(GG2(3),tca3,tsa4,tsb4,tcb4);
FA5b: FADD
        generic map(delay3)
        port map(GG2(4),tca4,tsa5,tsb5,tcb5);
FA6b: FADD
        generic map(delay3)
        port map(GG3(4),tca5,GG5(2),tsb6(0),tcb6(0));
FA6b1: FADD
        generic map(delay3)
        port map(GG6(1),GG2(5),tsa6,tsb6(1),tcb6(1));
FA7b: FADD
        generic map(delay3)
        port map(tca6,tsa7(0),tsa7(1),tsb7,tcb7);
HA8b: hadder
        generic map(delay3)
        port map(tca7(0),tca7(1),tsb8(0),tcb8(0));
FA8b: FADD
        generic map(delay3)
        port map(tsa8(0),tsa8(1),GG5(4),tsb8(1),tcb8(1));
FA9b: FADD
```

```
        generic map(delay3)
        port map(tca8(0),tca8(1),tsa9,tsb9,tcb9);

FA10b: FADD
        generic map(delay3)
        port map(tca9,tsa10,GG5(6),tsb10,tcb10);
FA11b: FADD
        generic map(delay3)
        port map(tca10,GG5(7),GG8(4),tsb11,tcb11);
HA3c: hadder
        generic map(delay3)
        port map(tcb2,tsb3,sum(3),tcc3);
y22: or_2
        generic map(delay4)
        port map(temp80, sum(3), temp90);
HA4c: hadder
        generic map(delay3)
        port map(tcb3,tsb4,tsc4,tcc4);
FA5c: FADD
        generic map(delay3)
        port map(tcb4,tsb5,GG4(2),tsc5,tcc5);
FA6c: FADD
        generic map(delay3)
        port map(tcb5,tsb6(0),tsb6(1),tsc6,tcc6);
FA7c: FADD
        generic map(delay3)
        port map(tcb6(0),GG5(3),GG4(4),tsc7(0),tcc7(0));
HA7c: hadder
        generic map(delay3)
        port map(tcb6(1),tsb7,tsc7(1),tcc7(1));
FA8c: FADD
        generic map(delay3)
        port map(tcb7,tsb8(0),tsb8(1),tsc8,tcc8);
FA9c: FADD
        generic map(delay3)
```

```
        port map(tcb8(0),GG5(5),GG6(4),tsc9(0),tcc9(0));
FA9c1: FADD
        generic map(delay3)
        port map(tcb8(1),tsb9,GG4(6),tsc9(1),tcc9(1));
FA10c: FADD
        generic map(delay3)
        port map(tcb9,tsb10,GG6(5),tsc10,tcc10);
FA11c: FADD
        generic map(delay3)
        port map(tcb10,tsb11,GG6(6),tsc11,tcc11);
FA12c: FADD
        generic map(delay3)
        port map(tcb11,GG6(7),GG8(5),tsc12,tcc12);
FA4d: FADD
        generic map(delay3)
        port map(tcc3,tsc4,GG3(2),sum(4),tcd4);
y23: or_2
        generic map(delay3)
        port map(temp90, sum(4), temp100);
FA5d: FADD
        generic map(delay3)
        port map(tcc4,tsc5,GG3(3),tsd5,tcd5);
FA6d1: FADD
        generic map(delay3)
        port map(tcc5,tsc6,Rin,tsd6,tcd6);
FA7d: FADD
        generic map(delay3)
        port map(tcc6,tsc7(0),tsc7(1),tsd7,tcd7);
FA8d: FADD
        generic map(delay3)
        port map(tcc7(0),tcc7(1),tsc8,tsd8,tcd8);
FA9d: FADD
        generic map(delay3)
        port map(tcc8,tsc9(0),tsc9(1),tsd9,tcd9);
FA10d: FADD
```

```
        generic map(delay3)
        port map(tcc9(0),tcc9(1),tsc10,tsd10,tcd10);

FA11d: FADD
        generic map(delay3)
        port map(tcc10,tsc11,GG7(5),tsd11,tcd11);
FA12d: FADD
        generic map(delay3)
        port map(tcc11,tsc12,GG7(6),tsd12,tcd12);
FA13d: FADD
        generic map(delay3)
        port map(tcc12,GG7(7),GG8(6),tsd13,tcd13);
FA14d: FADD
        generic map(delay3)
        port map(tsc14,tsc15,GG8(7),tsd14,tcd14);

HA5e: hadder
        generic map(delay3)
        port map(tsd5,tcd4,sum(5),carry(5));
y24: or_2
        generic map(delay4)
        port map(temp100, sum(5), S_temp);
HA6e: hadder
        generic map(delay3)
        port map(tsd6,tcd5,sum(6),carry(6));
HA7e: hadder
        generic map(delay3)
        port map(tsd7,tcd6,sum(7),carry(7));
HA8e: hadder
        generic map(delay3)
        port map(tsd8,tcd7,sum(8),carry(8));
HA9e: hadder
        generic map(delay3)
        port map(tsd9,tcd8,sum(9),carry(9));
HA10e: hadder
```

```
        generic map(delay3)
        port map(tsd10,tcd9,sum(10),carry(10));
HA11e: hadder
        generic map(delay3)
        port map(tsd11,tcd10,sum(11),carry(11));
HA12e: hadder
        generic map(delay3)
        port map(tsd12,tcd11,sum(12),carry(12));
HA13e: hadder
        generic map(delay3)
        port map(tsd13,tcd12,sum(13),carry(13));
HA14e: hadder
        generic map(delay3)
        port map(tsd14,tcd13,sum(14),carry(14));

sum(0) <= GG1(0);
T_sum <= sum(15 downto 6);
T_carry <= carry(14 downto 5);

x1: FADD
        generic map(delay3)
        port map(T_sum(1),T_carry(1),RV1,sum_tempv0(1),
                carry_tempv0(1));
x2: FADD
        generic map(delay3)
        port map(T_sum(2), T_carry(2), carry_tempv0(1),sum_tempv0(2)
                ,carry_tempv0(2));

G9:for i in 3 to 10 generate
    u11: and_2
        generic map(delay1)
        port map(T_sum(i), T_carry(i),g(i));
```

```
        z5: xor_2
                generic map(delay5)
                port map(T_sum(i), T_carry(i), p(i));
        end generate;

G10:    for i in 1 to 4 generate
        u12: and_2
                generic map(delay1)
                port map(p(2*i+2), g(2*i+1),tempv11(i));
         y1: or_2
                generic map(delay4)
                port map(g(2*i+2),tempv11(i),gg(2*i+2));
        u13: and_2
                generic map(delay1)
                port map(p(2*i+2), p(2*i+1), gp(2*i+2));
        end generate;

G1111:          for i in 1 to 4 generate
        u14: and_2
                generic map(delay1)
                port map(gp(2*i+2), carry_tempv0(2*i),carry_tempv11(i));
        y2: or_2
                generic map(delay4)
                port map(carry_tempv11(i),gg(2*i+2),carry_tempv0(2*i+2));
        end generate;

G12:    for i in 1 to 3 generate
        z3: xor_2
                generic map(delay5)
                port map(p(2*i+1), carry_tempv0(2*i), sum_tempv0(2*i+1));
        u15: and_2
                generic map(delay1)
                port map(p(2*i+1), carry_tempv0(2*i), tempv12(i));
```

```
y3: or_2
        generic map(delay4)
        port map(tempv12(i), g(2*i+1),tempv13(i));
z2: xor_2
        generic map(delay5)
        port map(p(2*i+2), tempv13(i),sum_tempv0(2*i+2));
        end generate;
x3: FADD
        generic map(delay3)
        port map(T_sum(9),T_carry(9),carry_tempv0(8),sum_tempv0(9),
                carry_tempv);
u16: and_2
        generic map(delay1)
        port map(p(9), carry_tempv0(8), tempv14);
y4: or_2
        generic map(delay4)
        port map(tempv14, g(9),Carry_tempv0(9));
x4: FADD
        generic map(delay3)
        port map(T_sum(10),T_carry(10),Carry_tempv0(9),
                sum_tempv0(10), carry_tempv0(10));
x5: FADD
        generic map(delay3)
        port map(T_sum(1),T_carry(1),RV2,sum_tempv1(1),
                carry_tempv1(1));
x6: FADD
        generic map(delay3)
        port map(T_sum(2), T_carry(2), carry_tempv1(1),sum_tempv1(2),
                carry_tempv1(2));

G13:  for i in 1 to 4 generate
      u17: and_2
              generic map(delay1)
              port map(gp(2*i+2), carry_tempv1(2*i),carry_tempv21(i));
        y6: or_2
```

```
            generic map(delay4)
            port map(carry_tempv21(i),gg(2*i+2),carry_tempv1(2*i+2));
            end generate;

G14:  for i in 1 to 3 generate
        z6: xor_2
            generic map(delay5)
            port map(p(2*i+1), carry_tempv1(2*i), sum_tempv1(2*i+1));
         u18: and_2
            generic map(delay1)
            port map(p(2*i+1), carry_tempv1(2*i), tempv21(i));
        y7: or_2
            generic map(delay4)
            port map(tempv21(i), g(2*i+1),tempv23(i));
        z7: xor_2
            generic map(delay5)
            port map(p(2*i+2), tempv23(i),sum_tempv1(2*i+2));
            end generate;
        x7: FADD
            generic map(delay3)
            port map(T_sum(9),T_carry(9),carry_tempv1(8),sum_tempv1(9),
                carry_tempvv);
        u19: and_2
            generic map(delay1)
            port map(p(9), carry_tempv1(8), tempv24);
        y8: or_2
            generic map(delay4)
            port map(tempv24, g(9),Carry_tempv1(9));
        x9: FADD
            generic map(delay3)
            port map(T_sum(10),T_carry(10),Carry_tempv1(9),
                sum_tempv1(10), carry_tempv1(10));

        Product0<= sum_tempv0 ;
        Product1<= sum_tempv1 ;
```

```
        v111:  V_detector
               port map(product0, V);

        in4:   inverter
               generic map(delay6)
               port map(V, V_bar);

--      multplexer to Select output

        G200:  for i in 1 to 10 generate
               u200:  and_2
                      generic map(delay1)
                      port map(V_bar, product0(i), temp110(i));
               u201:  and_2
                      generic map(delay1)
                      port map(V, product1(i), temp120(i));
               y200:  or_2
                      generic map(delay4)
                      port map(temp110(i), temp120(i), Product(i));
               end generate;
               v11:   RL_detector
                      port map(product,R_temp,R,L);
--      Correct the sticky bit depending on V

               S1:    sticky
                      generic map(delay4)
                      port map(V,S_temp, R_temp,S);
--      detect the Restorer Bit

               in1:   inverter
                      generic map(delay6)
                      port map(R, R_bar);
               in2:   inverter
                      generic map(delay6)
```

```
                port map(S,S_bar);

            a3: and_3
                generic map(delay7)
                port map(L, R_bar,S_bar, Res);

        G500: for i in 2 to 9 generate
            u500: and_2
                generic map(delay1)
                port map(v_bar, product(i), temp200(i-1));
            u501: and_2
                generic map(delay1)
                port map(V, product(i+1), temp210(i-1));
            y1500: or_2
                generic map(delay1)
                port map(temp200(i-1), temp210(i-1), fproduct_temp(i-1));
                end generate;

--      Check Condtions of L, R, S
            in8: inverter
                generic map(delay6)
                port map(Res, Res_bar);
            u300: and_2
                generic map(delay1)
                port map(Res_bar, fproduct_temp(1),out_temp);

--          CA + CB -127 ----
        G550: for i in 1 to 8 generate
            d500: d_latchN
                generic map(delay2)
                port map(clk1, ca1l(i), ca1(i));
            d501: d_latchN
                generic map(delay2)
                port map(clk1, cb1l(i), cb1(i));
                end generate;
```

```
CA_I1 <= to_integer(ca1);
CB_I1 <= to_integer(cb1);

inv500: inverter
        generic map(delay6)
        port map(cb1(8), msb_cB);
x500: Fadd
        generic map(delay3)
        port map(ca1(1),cb1(1),c0,sum_temp100(1), carry_temp100(1));
x501: Fadd
        generic map(delay3)
port map(ca1(2), cb1(2), carry_temp100(1),sum_temp100(2),
        carry_tempx);
u500: and_2
        generic map(delay1)
        port map(ca1(1), cb1(1),temp30);
u501: and_2
        generic map(delay1)
        port map(ca1(1), c0,temp31);
u502: and_2
        generic map(delay1)
        port map(cb1(1), c0,temp32);
k500:  or_3
        generic map(delay8)
        port map(temp30, temp31, temp32, temp33);
u503: and_2
        generic map(delay1)
        port map(ca1(2), cb1(2), temp34);
u504: and_2
        generic map(delay1)
        port map(ca1(2), temp33, temp35);
u505: and_2
        generic map(delay1)
        port map(temp33, cb1(2), temp36);
k501: or_3
```

```
            generic map(delay8)
            port map(temp34, temp35,temp36, carry_temp100(2));

        u506: and_2
            generic map(delay1)
            port map(ca1(8), msb_CB, ge(8));
        z500: xor_2
            generic map(delay5)
            port map(ca1(8), msb_CB, pe(8));

    G580: for i in 3 to 7 generate
        u507: and_2
            generic map(delay1)
            port map(ca1(i), cb1(i),ge(i));
        z501: xor_2
            generic map(delay5)
            port map(ca1(i), cb1(i), pe(i));
            end generate;

    G501: for i in 1 to 3 generate
        u508: and_2
            generic map(delay1)
            port map(pe(2*i+2), ge(2*i+1),temp1e(i));
        y500: or_2
            generic map(delay4)
            port map(ge(2*i+2),temp1e(i),gge(2*i+2));
        u509: and_2
            generic map(delay1)
            port map(pe(2*i+2), pe(2*i+1), gpe(2*i+2));
        end generate;

    G502: for i in 1 to 3 generate
        u510: and_2
            generic map(delay1)
            port map(gpe(2*i+2), carry_temp100(2*i),carry_temp3e(i));
```

```
        y501: or_2
                generic map(delay4)
                port map(carry_temp3e(i),gge(2*i+2),carry_temp100(2*i+2));

                end generate;
G503:   for i in 1 to 3 generate
        z502: xor_2
                generic map(delay5)
                port map(pe(2*i+1), carry_temp100(2*i), sum_temp100(2*i+1));
        u511: and_2
                generic map(delay1)
                port map(pe(2*i+1), carry_temp100(2*i), temp2e(i));
        y502: or_2
                generic map(delay4)
                port map(temp2e(i), ge(2*i+1),temp3e(i));
        z503: xor_2
                generic map(delay5)
                port map(pe(2*i+2), temp3e(i),sum_temp100(2*i+2));
        end generate;
        x502: Fadd
                generic map(delay3)
                port map(sum_temp100(1),m1(1),V_bar1,sum_tempe0(1),
                        carry_tempe0(1));
        x503: Fadd
                generic map(delay3)
                port map(sum_temp100(2),m1(2),carry_tempe0(1),sum_tempe0(2),
                carry_tempxx);
        u512: and_2
                generic map(delay1)
                port map(sum_temp100(1), m1(1),temp10);
        u513: and_2
                generic map(delay1)
                port map(sum_temp100(1), V_bar1,temp11);
        u514: and_2
                generic map(delay1)
```

```
            port map(m1(1), V_bar1,temp12);
        k502: or_3
            generic map(delay8)
            port map(temp10, temp11, temp12, temp13);
        u515: and_2
            generic map(delay1)
            port map(sum_temp100(2), m1(2), temp14);
        u516: and_2
            generic map(delay1)
            port map(sum_temp100(2), temp13, temp15);
        u517: and_2
            generic map(delay1)
            port map(temp13, m1(2), temp16);
        k503: or_3
            generic map(delay8)
            port map(temp14, temp15,temp16, carry_tempe0(2));

G504:for i in 3 to 8 generate
        u518: and_2
            generic map(delay1)
            port map(sum_temp100(i), m1(i),g11(i));
        z504: xor_2
            generic map(delay5)
            port map(sum_temp100(i), m1(i), p1(i));
        end generate;

G505:  for i in 1 to 3 generate
        u519: and_2
            generic map(delay1)
            port map(p1(2*i+2), g11(2*i+1),temp4(i));
          y503: or_2
            generic map(delay4)
            port map(g11(2*i+2),temp4(i),gg11(2*i+2));
          u520: and_2
            generic map(delay1)
```

```
            port map(p1(2*i+2), p1(2*i+1), gp1(2*i+2));
        end generate;

G506:   for i in 1 to 3 generate
        u521: and_2
            generic map(delay1)
            port map(gp1(2*i+2), carry_tempe0(2*i),carry_temp4e0(i));
         y504: or_2
            generic map(delay4)
            port map(carry_temp4e0(i),gg11(2*i+2),carry_tempe0(2*i+2));
        end generate;

        E33e0 <= carry_tempe0(8);

G507:   for i in 1 to 3 generate
        z505: xor_2
            generic map(delay5)
            port map(p1(2*i+1), carry_tempe0(2*i), sum_tempe0(2*i+1));
         u522: and_2
            generic map(delay1)
            port map(p1(2*i+1), carry_tempe0(2*i), temp5e0(i));
        y505: or_2
            generic map(delay4)
            port map(temp5e0(i), g11(2*i+1),temp6e0(i));
        z506: xor_2
            generic map(delay5)
            port map(p1(2*i+2), temp6e0(i),sum_tempe0(2*i+2));
            end generate;

        x504: Fadd
            generic map(delay3)
            port map(sum_temp100(1),m1(1),V1,sum_tempe1(1),
                carry_tempe1(1));
        x505: Fadd
            generic map(delay3)
```

```
            port map(sum_temp100(2), m1(2),carry_tempe1(1),
                sum_tempe1(2),carry_tempxxx);


        u523: and_2
            generic map(delay1)
            port map(sum_temp100(1), m1(1),temp20);
        u524: and_2
            generic map(delay1)
            port map(sum_temp100(1),V1,temp21);
        u525: and_2
            generic map(delay1)
            port map(m1(1), V1,temp22);
        k504: or_3
            generic map(delay8)
            port map(temp20, temp21, temp22, temp23);
        u526: and_2
            generic map(delay1)
            port map(sum_temp100(2), m1(2), temp24);
        u527: and_2
            generic map(delay1)
            port map(sum_temp100(2), temp23, temp25);
        u528: and_2
            generic map(delay1)
            port map(temp23, m1(2), temp26);
        k505: or_3
            generic map(delay8)
            port map(temp24, temp25,temp26, carry_tempe1(2));
    G508: for i in 1 to 3 generate

        u529: and_2
            generic map(delay1)
            port map(gp1(2*i+2), carry_tempe1(2*i),carry_temp4e0(i));
        y999: or_2
            generic map(delay4)
```

```
            port map(carry_temp4e0(i),gg11(2*i+2),carry_tempe1(2*i+2));
        end generate;

        E33e1 <= carry_tempe1(8);


    G509: for i in 1 to 3 generate
        z507: xor_2
            generic map(delay5)
            port map(p1(2*i+1), carry_tempe1(2*i), sum_tempe1(2*i+1));
        u530: and_2
            generic map(delay1)
            port map(p1(2*i+1), carry_tempe1(2*i), temp5e0(i));
        y507: or_2
            generic map(delay4)
            port map(temp5e0(i), g11(2*i+1),temp6e0(i));
        z508: xor_2
            generic map(delay5)
            port map(p1(2*i+2), temp6e0(i),sum_tempe1(2*i+2));
        end generate;

--          2x1 Multeplixier
    G510: for i in 1 to 8 generate
        u531: and_2
            generic map(delay1)
            port map(V_bar, sum_tempe0(i), temp500(i));
        u532: and_2
            generic map(delay1)
            port map(V, sum_tempe1(i), temp501(i));
        y508: or_2
            generic map(delay4)
            port map(temp500(i), temp501(i), CRR(i));
            end generate;
        u533: and_2
            generic map(delay1)
```

```
        port map(V_bar, E33e0, E33_temp1);
u544: and_2
        generic map(delay1)
        port map(V, E33e1, E33_temp2);

y509: or_2
        generic map(delay4)
        port map(E33_temp1, E33_temp2, E33);
a7_500: and_7
        port map(cb1, E1);
u545: and_2
        generic map(delay1)
        port map(E1, E22, temp510);
in501: inverter
        generic map(delay6)
        port map(E1, E1_bar);
in503: inverter
        generic map(delay6)
        port map(E22, E2_bar);
a500:  and_3
        generic map(delay7)
        port map(E1, E2_bar,E33, temp511);
u546: and_2
        generic map(delay1)
        port map(E1_bar, E2_bar, F2);
y510: or_2
        generic map (delay4)
        port map(temp510, temp511, F1);
in504: inverter
        generic map(delay6)
        port map(F1, F1_bar);
in505: inverter
        generic map(delay6)
        port map(F2, F2_bar);
```

```
G511: for i in 1 to 8 generate
        a501: and_3
                generic map(delay7)
                port map( F1_bar, F2_bar, CRR(i), temp520(i));
        a502: and_3
                generic map(delay7)
                port map(UN,F1_bar, F2, temp521(i));
        a503: and_3
                generic map(delay7)
                port map(OV,F1,F2_bar, temp522(i));
        k505: or_3
                generic map(delay8)
                port map(temp520(i), temp521(i),temp522(i),CR(i));
        end generate;

G512:  for i in 1 to 8 generate
        d503: d_latchP
                generic map(delay2)
                port map(clk1, CR(i), CRI(i));
                end generate:

        ain1<= to_integer(a11);
        bin1 <= to_integer(b11);

G400:  for i in 2 to 8 generate
        d3: d_latchP
                generic map(delay2)
                port map(clk1, fproduct_temp(i), fproduct_test(i));
                end generate;
        d4: d_latchP
                generic map(delay2)
                port map(clk1, out_temp, out_templ);
        fproduct <= fproduct_test(8 downto 2)& out_templ;
end structure;
```

APPENDIX
1 B

Behavioral Model for the New ADDER

---

adder.vhd

```
library XL;
use Xl.XL_STD.all, XL.XL_GATES.all;
entity tenbit_adder is
        generic (delay1,delay2: in time);
        port(a,b: bit_vector(10 downto 1); c0: in bit;  sum: out bit_vector(11 downto 1));
end tenbit_adder;

architecture behavior of tenbit_adder is

        begin

p1:     process(a,b,c0)
                variable g,gg,p,gp: bit_vector(10 downto 1):="0000000000";
                variable sum_temp,carry_temp: bit_vector(10 downto 1);
                variable k: integer;

        begin

        loop1:
                for i in 1 to 2 loop
                if(i= 1) then
                sum_temp(i):= (a(i) xor b(i)) xor c0 ;
                carry_temp(i):= (a(i) and b(i)) or (a(i) and c0) or (b(i) and c0);
                else
                sum_temp(i):= (a(i) xor b(i)) xor carry_temp(i-1) ;
                carry_temp(i):= (a(i) and b(i)) or (a(i) and carry_temp(i-1)) or (b(i) and
                        carry_temp(i-1));
                end if;
                end loop loop1;
```

```
    loop2:

            for i in 3 to 10 loop
            g(i):= a(i) and b(i);
            p(i):= a(i) xor b(i);
            end loop loop2;

    loop3:
            for i in 1 to 4 loop
                    k:=2*i;
            gg(k+2):= (p(k+2) and g(k+1)) or g(k+2);
            gp(k+2):= p(k+2) and p(k+1);
            end loop loop3;

    loop4:
            for i in 1 to 4 loop
                    k:=2*i;
            carry_temp(k+2):= (gp(k+2)and carry_temp(k)) or gg(k+2);
            end loop loop4;

    loop5:
            for i in 1 to 4 loop
                    k:=2*i;
            sum_temp(k+1):= p(k+1) xor carry_temp(k);
            sum_temp(k+2):= p(k+2) xor ((p(k+1) and carry_temp(k)) or g(k+1));
            end loop loop5;
            sum <= carry_temp(10)&sum_temp after delay2;
     end process;

end structure;
```

---

tb_adder.vhd

```
library XL;
use XL.XL_STD.all, STD.TEXTIO.all, XL.XL_IO.all;
entity tb_adder is
end tb_adder;
architecture behavior of tb_adder is

        component CPA_adder
                generic(delay1, delay2,delay3,delay4: in time);
                port(a,b: bit_vector(10 downto 1); c0: in bit; sum: out bit_vector(11 downto
                        1));
        end component;

signal input_a,input_b: bit_vector(10 downto 1);
signal input_carry: bit;
signal sum_output: bit_vector(11 downto 1);
signal Even_carry_output: bit_vector(5 downto 1);

for x1: CPA_adder use entity work.tenbit_adder(behavior);

        begin

                x1: CPA_adder
                        generic map(0.3218 ns,1.0216 ns, 0.5893 ns, 1.486 ns )
                        port map(input_a,input_b,input_carry,sum_output);


 input_a <= "0000000000" after 0 ns, "0000000001" after 7 ns,"1001000010" after 14
                ns, "1111111111" after 21 ns, "1010000000" after 28 ns, "1010101010"
                after 35 ns,"0000000000" after 42 ns;

 input_b <= "0000000000" after 0 ns, "0000000001" after 7 ns,"1010101010" after 14
                ns, "1111111111" after 21 ns, "1010000000" after 28 ns, "1110111101"
                after 35 ns, "1010111111" after 42 ns;
```

```
input_carry <= '0' after 0 ns, '1' after 7 ns, '0' after 14 ns, '1' after 21 ns, '0' after 28 ns,
               '1' after 35 ns, '0' after 42 ns;

--waves_monitor(a,b,c);
-- display result

        process
                variable L:line;
        begin
                write (L, "  TIME input_a  input_b  input_carry  sum_output   ");
                writeline (output, L);
                write (L, "   ---   ---   ---   ---");
                writeline (output, L);
                write (L, " ");
                Writeline (output, L);
                wait;
        end process;
monitor_process: process(input_a, input_b,input_carry )
                variable dline: line;

        begin
                write (dline, NOW, right, 10);
                write (dline, input_a, right, 15);
                write (dline, input_b, right, 15);
                write (dline, input_carry, right, 7);
                write (dline, sum_output, right, 15);
                writeline (output, dline);

        end process;

end behavior;
```

APPENDIX
2B

Architectural Model for the New Adder

```
--------------------------------------------------------------------------------

                                adder1.vhd

--------------------------------------------------------------------------------


Library XL;
use XL.XL_STD.all;
Library Madder;
use Madder xor_2,or_2,and_2;

entity tenbit_adder1 is
        generic (delay1,delay2,delay3,delay4: in time);
        port(a,b: bit_vector(10 downto 1); c0: in bit; sum: out bit_vector(11 downto 1));
end tenbit_adder1;

architecture structure of tenbit_adder1 is

                component Fadd
                        generic (delay: in time);
                        port (a,b,c: bit; sum, carry: inout bit);
                end component;
                component and_2
                        generic (delay: in time);
                        port (a,b:bit; c: out bit);
                end component;
                component xor_2
                        generic (delay: in time);
                        port (a,b:bit; c: out bit);
                end component;
                component or_2
                        generic (delay: in time);
                        port (a,b:bit; c:out bit);
                end component;
 signal g,gg,p,gp: bit_vector(10 downto 1):="0000000000";
```

```
signal sum_temp,carry_temp: bit_vector(10 downto 1):="0000000000";
signal temp1,temp2,temp3,carry_temp1: bit_vector(4 downto 1):="0000";
signal temp4:bit:='0';

for all: Fadd use entity work.FA(structure);
for all: and_2 use entity work.and_2(behavior);
for all: or_2 use entity Madder.or_2(behavior);
for all: xor_2 use entity Madder.xor_2(behavior);

    begin

        x1: Fadd
            generic map(delay4)
            port map(a(1),b(1),c0,sum_temp(1), carry_temp(1));

        x2: Fadd
            generic map(delay4)
            port map(a(2), b(2), carry_temp(1),sum_temp(2),carry_temp(2));

G0:     for i in 3 to 10 generate
        u1: and_2
             generic map(delay1)
            port map(a(i), b(i),g(i));
        z5: xor_2
            generic map(delay2)
            port map(a(i), b(i), p(i));
            end generate;

G1:     for i in 1 to 4 generate
         u2: and_2
            generic map(delay1)
            port map(p(2*i+2), g(2*i+1),temp1(i));
```

```
        y1: or_2
              generic map(delay3)
              port map(g(2*i+2),temp1(i),gg(2*i+2));
        u3: and_2
              generic map(delay1)
              port map(p(2*i+2), p(2*i+1), gp(2*i+2));
              end generate;

G3:     for i in 1 to 4 generate
        u4: and_2
              generic map(delay1)
              port map(gp(2*i+2), carry_temp(2*i),carry_temp1(i));
        y2: or_2
              generic map(delay3)
              port map(carry_temp1(i),gg(2*i+2),carry_temp(2*i+2));
              end generate;

G4:     for i in 1 to 3 generate
        z3: xor_2
              generic map(delay2)
              port map(p(2*i+1), carry_temp(2*i), sum_temp(2*i+1));
        u5: and_2
              generic map(delay1)
              port map(p(2*i+1), carry_temp(2*i), temp2(i));
        y3: or_2
              generic map(delay3)
              port map(temp2(i), g(2*i+1),temp3(i));
        z2: xor_2
              generic map(delay2)
              port map(p(2*i+2), temp3(i),sum_temp(2*i+2));
              end generate;
        x3: Fadd
              generic map(delay4)
              port map(a(9),b(9),carry_temp(8),sum_temp(9), carry_temp(9));
```

```
            u6: and_2
                    generic map(delay1)
                    port map(p(9), carry_temp(8), temp4);
            y3: or_2
                    generic map(delay3)
                    port map(temp4, g(9),Carry_temp(9));
            x4: Fadd
                    generic map(delay4)
                    port map(a(10),b(10),Carry_temp(9),sum_temp(10),
                            carry_temp(10));

--          process
--          variable k: integer;
--      begin
--          loop6:
--                  for i in 1 to 5 loop
--                  k:=2*i;
--
--                  carry(i) <= carry_temp(2*i) after delay1;
--                  end loop loop6;
--          end process;

            sum <= carry_temp(10)&sum_temp ;

end structure;

-------------------------------------------------------------------------------------------

                                    tb_adder1.vhd

-------------------------------------------------------------------------------------------


library XL;
use XL.XL_STD.all, STD.TEXTIO.all, XL.XL_IO.all;
entity tb_adder is
```

```
end tb_adder;
architecture structure of tb_adder is

        component CPA_adder
                generic(delay1, delay2,delay3,delay4: in time);
                port(a,b: bit_vector(10 downto 1); c0: in bit; sum: out bit_vector(11 downto
                        1));
        end component;

signal input_a,input_b: bit_vector(10 downto 1);
signal input_carry: bit;
signal sum_output: bit_vector(11 downto 1);
signal Even_carry_output: bit_vector(5 downto 1);

for x1: CPA_adder use entity work.tenbit_adder(structure);

        begin

                x1: CPA_adder
                        generic map(0.3218 ns,1.0216 ns, 0.5893 ns, 1.486 ns )
                         port map(input_a,input_b,input_carry,sum_output);


  input_a <= "0000000000" after 0 ns, "0000000001" after 7 ns,"1001000010" after 14
                ns, "1111111111" after 21 ns, "1010000000" after 28 ns, "1010101010"
                after 35 ns,"0000000000" after 42 ns;

  input_b <= "0000000000" after 0 ns, "0000000001" after 7 ns,"1010101010" after 14
                ns, "1111111111" after 21 ns, "1010000000" after 28 ns, "1110111101"
                after 35 ns, "1010111111" after 42 ns;

  input_carry <= '0' after 0 ns, '1' after 7 ns, '0' after 14 ns, '1' after 21 ns, '0' after 28 ns,
                '1' after 35 ns, '0' after 42 ns;

 --waves_monitor(a,b,c);
```

```
-- display result

        process
                variable L:line;
        begin
                write (L, "  TIME input_a  input_b  input_carry  sum_output   ");
                writeline (output, L);
                write (L, "   ---   ---   ---   ---");
                writeline (output, L);
                write (L, " ");
                Writeline (output, L);
                wait;
        end process;
monitor_process: process(input_a, input_b,input_carry )
                variable dline: line;

        begin
                write (dline, NOW, right, 10);
                write (dline, input_a, right, 15);
                write (dline, input_b, right, 15);
                write (dline, input_carry, right, 7);
                write (dline, sum_output, right, 15);
                writeline (output, dline);

        end process;

end structure;
```

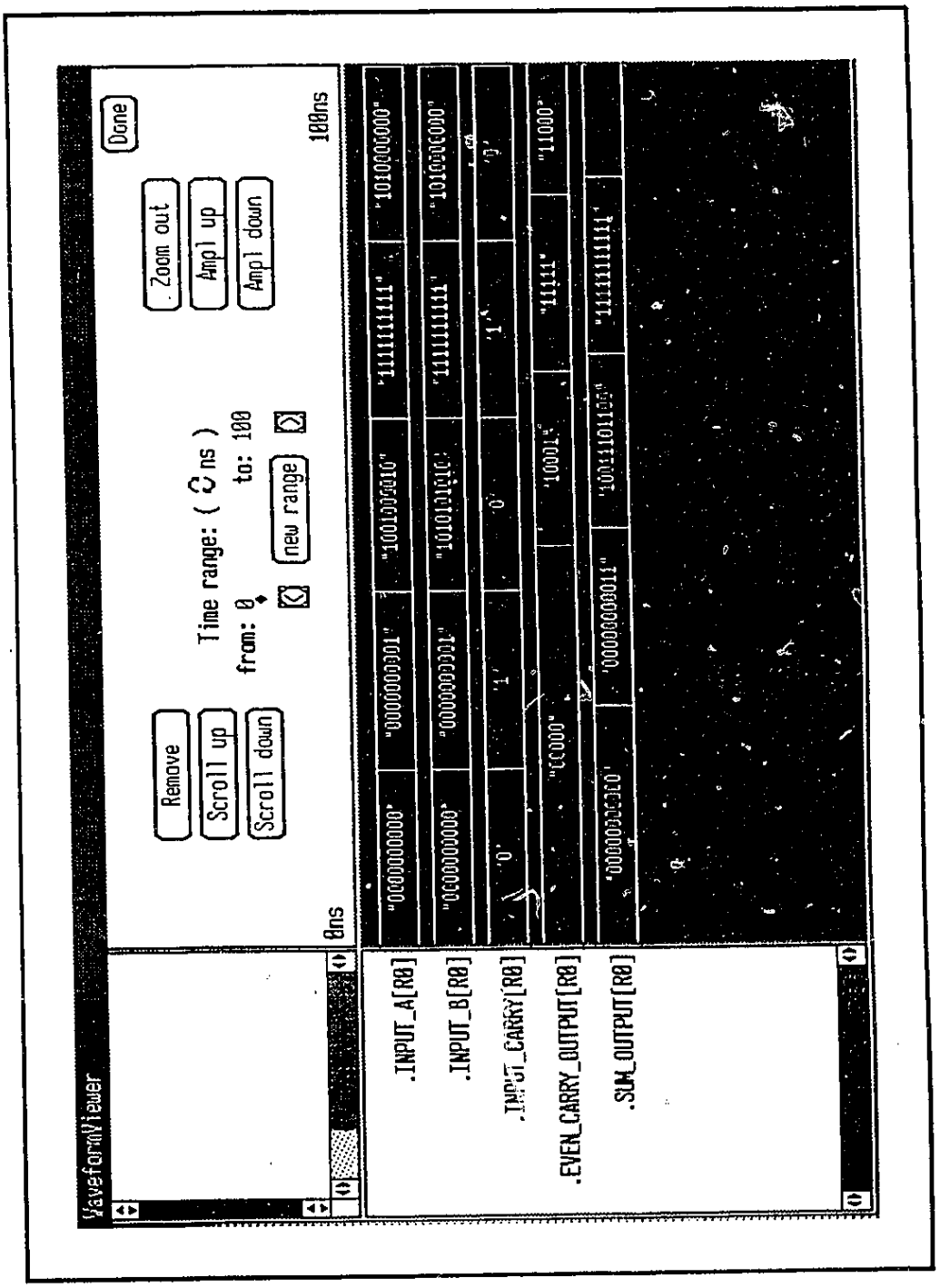


Figure 2B.1: VHDL Waveforms for the New Adder

APPENDIX
1C

Architectural Model for Type_1 Two-Bit Full_Adder Multiplier

```
--------------------------------------------------------------------------------
                                five_bit.vhd
--------------------------------------------------------------------------------
library XL;
use XL.XL_STD.all,STD.TEXTIO.all, XL.XL_IO.all;

entity five_bit is
        generic (delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8: in time);
        port(a1,b1: bit_vector(7 downto 0);clk1: in bit;Product: inout bit_vector(15 downto
                0); ain1,bin1,product1: out integer);
end five_bit;

architecture structure of five_bit is
        component and_2
                generic(delay : in time);
                port(a,b: in bit; c: out bit);
        end component;
        component xor_2
                generic (delay: in time);
                port(a,b: bit; c: out bit);
        end component;
        component or_2
                generic (delay: in time);
                port(a,b:bit; c: out bit);
        end component;
        component inverter
                generic (delay: in time);
                port (a: bit; b: out bit);
        end component;
        component xnor_2
                generic (delay: in time);
```

```
            port(a,b: bit; c: out bit);
        end component;
        component or_3
            generic (delay: in time);
            port(a,b,c: bit; d: out bit);
        end component;
        component twobit_FA
            generic (delay1,delay2,delay3,delay4,delay5,delay6: in time);
            port (a,b,c,d,e: in bit; s0,s1,c1: out bit);
        end component;
        component FADD
            generic(delay: in time);
            port(a,b,c: bit; sum:out bit; carry: inout bit);
        end component;
            component d_latchN
            generic(delay:in time);
            port(clk1,x: in bit; y:out bit);
        end component;
        component d_latchP
                    generic(delay: in time);
                    port(clk1,x:in bit; y: out bit);
            end component;

signal tempa1,tempa2,tempa3,tempa4,tempa5,tempa6,tempa7: bit_vector(2 downto
        0):="000";
signal tempb2,tempb3,tempb4,tempb5,tempb6,tempb7: bit_vector(2 downto 0):="000";
signal tempc3,tempc4,tempc5,tempc6,tempc7: bit_vector(2 downto 0):="000";
signal tempd4,tempd5,tempd6,tempd7: bit_vector(2 downto 0):="000";
signal tempe5,tempe6,tempe7: bit_vector(2 downto 0):="000";
signal tempf6,tempf7: bit_vector(2 downto 0):="000";
signal tempg7: bit_vector(2 downto 0):="000";
signal c0,c1,tempb1,tempc,tempc2,tempd2,tempd3,tempe3,tempe4,tempf4,tempf5,
        tempg5,tempg6,temph6,temph7:bit:='0';
signal GG1,GG2,GG3,GG4,GG5,GG6,GG7,GG8:bit_vector(7 downto
        0):="00000000";
```

```
signal a11,b11: bit_vector(7 downto 0):="00000000";
signal P: bit_vector(15 downto 0):="0000000000000000";
for all: and_2 use entity work.and_2(behavior);
for all: xor_2 use entity work.xor_2(behavior);
for all: or_2 use entity work.or_2(behavior);
for all: d_latchN use entity work.d_latchN(behavior);
for all: d_latchP use entity work.d_latchP(behavior);
for all: FADD use entity work.FADD(structure);
for all: twobit_FA use entity work.twobit_FA(structure);
for all: or_3 use entity work.or_3(behavior);
for all: xnor_2 use entity work.xnor_2(behavior);
for all: inverter use entity work.inverter(behavior);

  begin

      G0: for i in 0 to 7 generate
            d0: d_latchN
                  generic map(delay8)
                  port map(clk1, a1(i), a11(i));
            end generate;
      G141: for i in 0 to 7 generate
            d1: d_latchN
                  generic map(delay8)
                  port map(clk1, b1(i), b11(i));
            end generate;
      G1: for i in 0 to 7 generate
            u0: and_2
                  generic map(delay1)
                  port map(a11(i),b11(0),GG1(i));
            end generate;
      G2: for i in 0 to 7 generate
            u1: and_2
                  generic map(delay1)
                  port map(a11(i),b11(1),GG2(i));
            end generate;
```

```
G3: for i in 0 to 7 generate
        u2: and_2
                generic map(delay1)
                port map(a11(i),b11(2),GG3(i));
        end generate;
G4: for i in 0 to 7 generate
        u3: and_2
                generic map(delay1)
                port map(a11(i),b11(3),GG4(i));
        end generate;
G5: for i in 0 to 7 generate
        u4: and_2
                generic map(delay1)
                port map(a11(i),b11(4),GG5(i));
        end generate;
G6: for i in 0 to 7 generate
        u5: and_2
                generic map(delay1)
                port map(a11(i),b11(5),GG6(i));
        end generate;
G7: for i in 0 to 7 generate
        u6: and_2
                generic map(delay1)
                port map(a11(i),b11(6),GG7(i));
        end generate;
G8: for i in 0 to 7 generate
        u7: and_2
                generic map(delay1)
                port map(a11(i),b11(7),GG8(i));
        end generate;
        P(0) <= GG1(0);

f1:     twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG2(1),c0,GG1(1),GG2(0),c1, P(1),tempa1(1),tempa1(2));
```

```
f2:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG3(1),GG2(2),GG3(0),GG1(2),c0,tempa2(0),tempa2(1),tempa2(2));
f3:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG2(3),GG4(1),GG4(0),GG1(3),c0,tempa3(0),tempa3(1),tempa3(2));
f4:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG5(1),GG2(4),GG5(0),GG1(4),c0,tempa4(0),tempa4(1),tempa4(2));
f5:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG6(1),GG2(5),GG6(0),GG1(5),c0,tempa5(0),tempa5(1),tempa5(2));
f6:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG7(1),GG2(6),GG7(0),GG1(6),c0,tempa6(0),tempa6(1),tempa6(2));
f7:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG8(1),GG2(7),GG8(0),GG1(7),c0,tempa7(0),tempa7(1),tempa7(2));
FA1:  FADD
      generic map(delay7)
      port map(tempa1(1),tempa2(0),c0, P(2), tempb1);
f8:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG3(2),c0,tempa1(2),tempa2(1),tempa3(0),tempb2(0),
            tempb2(1),tempb2(2));
f9:   twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG4(2),GG3(3),tempa2(2),tempa3(1),tempa4(0),tempb3(0),
            tempb3(1),tempb3(2));
f10:  twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
      port map(GG5(2),GG3(4),tempa3(2),tempa4(1),tempa5(0),tempb4(0),t
      empb4(1),tempb4(2));
f11:  twobit_FA
      generic map(delay1,delay2,delay3,delay4,delay5,delay6)
```

```
        port map(GG3(5),GG6(2),tempa4(2),tempa5(1),tempa6(0),tempb5(0),
                tempb5(1),tempb5(2));
f12:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG7(2),GG3(6),tempa5(2),tempa6(1),tempa7(0),tempb6(0),
                tempb6(1),tempb6(2));
f13:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG8(2),GG3(7),tempa6(2),tempa7(1), c0,tempb7(0),
                tempb7(1),tempb7(2));

z1:     xor_2
        generic map(delay4)
        port map(tempb1, tempb2(0),P(3));
u8:     and_2
        generic map(delay1)
        port map(tempb1, tempb2(0), tempc);
FA2:    FADD
        generic map(delay7)
        port map(tempc, tempb2(1), tempb3(0), P(4), tempc2);
f14:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG4(3),c0,tempb2(2),tempb3(1), tempb4(0),tempc3(0)
                tempc3(1),tempc3(2));
f15:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG5(3),GG4(4),tempb3(2),tempb4(1), tempb5(0),tempc4(0)
                ,tempc4(1),tempc4(2));
f16:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG6(3),GG4(5),tempb4(2),tempb5(1), tempb6(0),tempc5(0),
                tempc5(1),tempc5(2));
f17:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG7(3),GG4(6),tempb5(2),tempb6(1), tempb7(0),tempc6(0),
```

```
            tempc6(1),tempc6(2));
f18:    twobit_FA

        generic map(delay1,delay2,delay3,delay4,delay5,delay6)

        port map(GG8(3),GG4(7),tempa7(2),tempb6(2), tempb7(1),tempc7(0),

            tempc7(1),tempc7(2));

z2:     xor_2
        generic map(delay4)
        port map(tempc2, tempc3(0),P(5));
u9:     and_2
        generic map(delay1)
        port map(tempc2, tempc3(0), tempd2);
FA3:    FADD
        generic map(delay7)
        port map(tempd2, tempc3(1), tempc4(0), P(6), tempd3);
f19:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG5(4),c0,tempc3(2),tempc4(1), tempc5(0),tempd4(0)
            ,tempd4(1),tempd4(2));
f20:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG6(4),GG5(5),tempc4(2),tempc5(1), tempc6(0),tempd5(0)
            ,tempd5(1),tempd5(2));
f21:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG7(4),GG5(6),tempc5(2),tempc6(1), tempc7(0),tempd6(0)
            ,tempd6(1),tempd6(2));
f22:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG8(4),GG5(7),tempc6(2),tempc7(1), tempb7(2),tempd7(0)
            ,tempd7(1),tempd7(2));
z3:     xor_2
        generic map(delay4)
```

```
        port map(tempd3, tempd4(0),P(7));
u10:    and_2
        generic map(delay1)
        port map(tempd3, tempd4(0), tempe3);
FA4:    FADD

        generic map(delay7)

        port map(tempe3, tempd4(1), tempd5(0), P(8), tempe4);
f23:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG6(5),c0,tempd4(2),tempd5(1), tempd6(0),tempe5(0)
                ,tempe5(1),tempe5(2));
f24:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG7(5),GG6(6),tempd5(2),tempd6(1), tempd7(0),tempe6(0)
                ,tempe6(1),tempe6(2));
f25:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG8(5),GG6(7),tempd6(2),tempd7(1), tempc7(2),tempe7(0)
                ,tempe7(1),tempe7(2));
z4:     xor_2
        generic map(delay4)
        port map(tempe4, tempe5(0),P(9));
u11:    and_2
        generic map(delay1)
        port map(tempe4, tempe5(0), tempf4);
FA5:    FADD
        generic map(delay7)
        port map(tempf4, tempe5(1), tempe6(0), P(10), tempf5);
f26:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG7(6),c0,tempe5(2),tempe6(1), tempe7(0),tempf6(0)
                ,tempf6(1),tempf6(2));
f27:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
```

```
        port map(GG7(7),GG8(6),tempe6(2),tempe7(1), tempd7(2),tempf7(0)
                ,tempf7(1),tempf7(2));
z5:     xor_2
        generic map(delay4)
        port map(tempf5, tempf6(0),P(11));
u12:    and_2
        generic map(delay1)
        port map(tempf5, tempf6(0), tempg5);
FA6:    FADD
        generic map(delay7)
        port map(tempg5, tempf6(1), tempf7(0), P(12), tempg6);
f28:    twobit_FA
        generic map(delay1,delay2,delay3,delay4,delay5,delay6)
        port map(GG8(7),c0,tempf6(2),tempf7(1), tempe7(2),tempg7(0)
                ,tempg7(1),tempg7(2));
z6:     xor_2
        generic map(delay4)
        port map(tempg6, tempg7(0),P(13));
u13:    and_2
        generic map(delay1)
        port map(tempg6, tempg7(0), temph6);
FA7:    FADD
        generic map(delay7)
        port map(temph6, tempg7(1), tempf7(2), P(14), temph7);
z7:     xor_2
        generic map(delay4)
        port map(tempg7(2), temph7,P(15));

G400:   for i in 0 to 15 generate
        d3: d_latchP
                generic map(delay8)
                port map(clk1, P(i), product(i));
        end generate;

                ain1<= to_integer(a11);
```

```
        bin1 <= to_integer(b11);
        product1 <= to_integer(product);
end structure;
```

Twobit_FA.vhd

```
entity twobit_FA is
    generic (delay1,delay2,delay3,delay4,delay5,delay6: in time);
    port(a,b,c,d,e: in bit; s0,s1,c1: out bit);
end twobit_FA;

architecture structure of twobit_FA is
    component and_2
        generic(delay : in time);
        port(a,b: in bit; c: out bit);
    end component;
    component xor_2
        generic (delay: in time);
        port(a,b: bit; c: out bit);
    end component;
    component or_2
        generic (delay: in time);
        port(a,b:bit, c: out bit);
    end component;
    component inverter
        generic (delay: in time);
        port (a: bit; b: out bit);
    end component;
    component xnor_2
        generic (delay: in time);
```

```
            port(a,b: bit; c: out bit);
        end component;
        component or_3
            generic (delay: in time);
            port(a,b,c: bit; d: out bit);
        end component;

signal temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,temp9:bit:='0';
signal temp10,temp11,temp12,temp13,temp14,temp15,temp16,temp17: bit:='0';
signal temp18,temp19:bit:='0';
signal c_bar,d_bar,e_bar: bit;
for all: and_2 use entity work.and_2(behavior);
for all: xor_2 use entity work.xor_2(behavior);
for all: or_2 use entity work.or_2(behavior);
for all: d_latchN use entity work.d_latchN(behavior);
for all: d_latchP use entity work.d_latchP(behavior);
for all: or_3 use entity work.or_3(behavior);
for all: xnor_2 use entity work.xnor_2(behavior);
for all: inverter use entity work.inverter(behavior);

begin

        in1:    inverter
                generic map(delay6)
                port map(c, c_bar);
        in2:    inverter
                generic map(delay6)
                port map(d, d_bar);
        in3:    inverter
                generic map(delay6)
                port map(e, e_bar);
        z1:     xor_2
                generic map(delay4)
                port map(d,e,temp1);
        x1:     xnor_2
```

```
        generic map(delay5)
        port map(d,e, temp2);
u1:     and_2
        generic map(delay1)
        port map(temp1, c_bar,temp3);
u2:     and_2
        generic map(delay1)
        port map(temp2, c,temp4);

y1:     or_2
        generic map(delay2)
        port map(temp3, temp4, s0);
u3:     and_2
        generic map(delay1)
        port map(c, d,temp5);
u4:     and_2
        generic map(delay1)
        port map(c, e,temp6);
u5:     and_2
        generic map(delay1)
        port map(d, e,temp7);
p1:     or_3
        generic map(delay3)
        port map(temp5, temp6,temp7,temp8);
y2:     or_2
        generic map(delay2)
        port map(a, b,temp9);
u6:     and_2
        generic map(delay1)
        port map(temp8,temp9,temp10);
u7:     and_2
        generic map(delay1)
        port map(a,b,temp11);
y3:     or_2
        generic map(delay2)
```

```
        port map(temp10,temp11,c1);
u8:     and_2
        generic map(delay1)
        port map(c_bar,d_bar,temp12);
u9:     and_2
        generic map(delay1)
        port map(d_bar,e_bar,temp13);
u10:    and_2
        generic map(delay1)
        port map(c_bar,e_bar,temp14);
p2:     or_3
        generic map(delay3)
        port map(temp12, temp13,temp14,temp15);
z2:     xor_2
        generic map(delay4)
        port map(a,b,temp16);
x2:     xnor_2
        generic map(delay5)
        port map(a,b, temp17);
u11:    and_2
        generic map(delay1)
        port map(temp15, temp16, temp18);
u12:    and_2
        generic map(delay1)
        port map(temp17, temp8, temp19);
y4:     or_2
        generic map(delay2)
        port map(temp18,temp19, s1);

end structure;

--------------------------------------------------------------------------------

                        tb_fivebit.vhd

--------------------------------------------------------------------------------

library XL;
```

```
use XL.XL_STD.all,STD.TEXTIO.all, XL.XL_IO.all;
entity tb_five is

end ;

architecture structure of tb_five is

signal a,b: bit_vector(7 downto 0);
signal ain,bin,product_I:integer:=0;
signal clk:bit:='0';
signal product_b:bit_vector(15 downto 0);

        component five_bit
                generic(delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8 :in time);
                port(a1,b1:bit_vector(7 downto 0); clk1: in bit; Product: inout bit_vector(15
                        downto 0); ain1,bin1,product1: out integer);

        end component;

for xx1: five_bitc use entity work.five_bit(structure);

begin
        xx1: five_bit
                generic map (0.3218 ns,0.5893 ns,0.6844 ns, 1.026 ns, 1.1042 ns.0.15
                                ns, 1.486 ns, 1.2 ns)
                port map (a,b, clk, product_b, ain, bin, product_I);

a <= "00000000","11000000" after 20 ns, "10000101" after 40 ns,"00000101" after
        60 ns, "11111111" after 80 ns, "00001010" after 100 ns, "00000001" after 120
        ns,"00000011" after 140 ns;

b <= "00000000","11000111" after 20 ns, "11000000" after 40 ns, "000001000" after 60
        ns, "11111111" after 80 ns, "00000100" after 100 ns, "00000001" after 120
        ns,"00000011" after 140 ns;

clk <='0', '1' after 18 ns,'0' after 20 ns, '1' after 38 ns, '0' after 40 ns, '1' after 58 ns,'0'
        after 60 ns, '1' after 78 ns, '0' after 80 ns, '1' after 98 ns, '0' after 100 ns, '1'
        after 118 ns, '0' after 120 ns, '1' after 138 ns, '0' after 140 ns;
```

```
process
        variable L:line;
    begin
        write (L, "  TIME   ain   bin   Product_I     Product_B");
        writeline (output, L);
        write (L, "  ---   ---   ---   ---");
        writeline (output, L);
        write (L, " ");
        writeline (output, L);
        wait;
end process;

monitor_process: process(a,b)
    variable dline: line;

    begin
        write (dline, NOW, right, 7);
        write (dline, ain, right, 7);
        write (dline, bin, right, 7);
        write (dline, Product_I, right, 10);

        write (dline, Product_b, right, 20);
--      write (dline, e1, right, 7);
--      write (dline, s01, right, 7);
--      write (dline, s11, right, 7);
--      write (dline, c11, right, 7);
        writeline (output, dline);

    end process;
waves_monitor (to_integer(a),to_integer(b),to_integer(product_b));
end structure ;
```

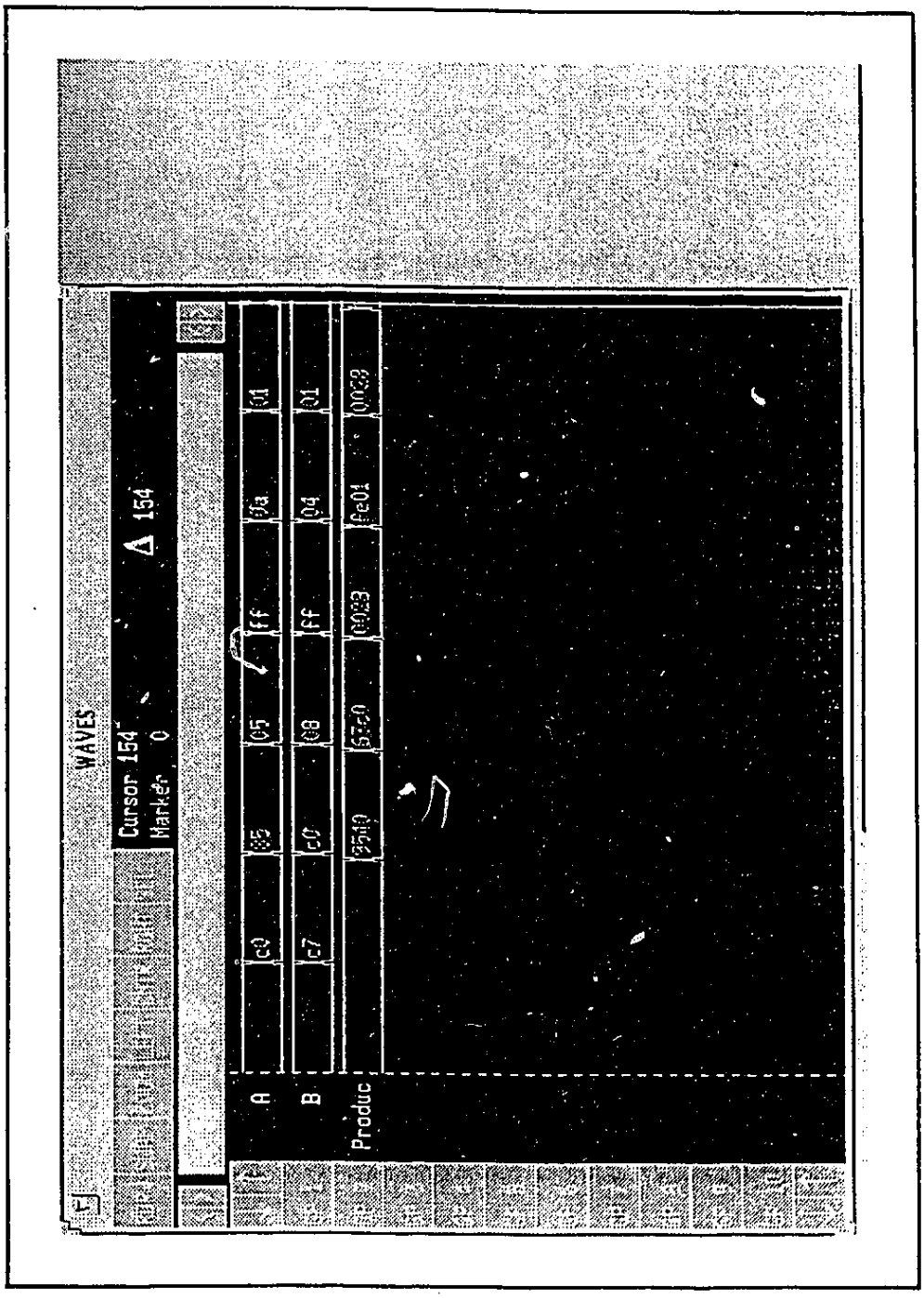


Figure 1C.1: VHDL Waveforms for Type_1 Two-Bit Full-Adder Multiplier

APPENDIX
2C

Architectural Model for Type_2 Two-Bit Full-Adder Multiplier

---

fivebit1.vhd

```
library XL;
use XL.XL_STD.all,STD.TEXTIO.all, XL.XL_IO.all;
entity five_bit1 is
        generic (delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8: in time);
        port(a1,b1: bit_vector(7 downto 0);clk1: in bit;Product: inout bit_vector(15 downto
        0); ain1,bin1,product1: out integer);
end five_bit1;

architecture structure of five_bit1 is
        component and_2
                generic(delay : in time);
                port(a,b: in bit; c: out bit);
        end component;
        component xor_2
                generic (delay: in time);
                port(a,b: bit; c: out bit);
        end component;
        component or_2
                generic (delay: in time);
                port(a,b:bit; c: out bit);
        end component;
        component inverter
                generic (delay: in time);
                port (a: bit; b: out bit);
        end component;
        component xnor_2
                generic (delay: in time);
                port(a,b: bit; c: out bit);
        end component;
        component or_3
                generic (delay: in time);
```

```
            port(a,b,c: bit; d: out bit);
        end component;
        component twobit_FA1
            generic (delay1: in time);
            port (a,b,c,d,e: in bit; s0,s1: out bit; c1: inout bit);
        end component;
        component FADD
            generic(delay: in time);
            port(a,b,c: bit; sum:out bit; carry: inout bit);
        end component;
        component d_latchN
            generic(delay:in time);
            port(clk1,x: in bit; y:out bit);
        end component;
        component d_latchP
            generic(delay: in time);
            port(clk1,x:in bit; y: out bit);
        end component;
signal tempa1,tempa2,tempa3,tempa4,tempa5,tempa6,tempa7: bit_vector(2 downto
        0):="000";
signal tempb2,tempb3,tempb4,tempb5,tempb6,tempb7: bit_vector(2 downto 0):="000";
signal tempc3,tempc4,tempc5,tempc6,tempc7: bit_vector(2 downto 0):="000";
signal tempd4,tempd5,tempd6,tempd7: bit_vector(2 downto 0):="000";
signal tempe5,tempe6,tempe7: bit_vector(2 downto 0):="000";
signal tempf6,tempf7: bit_vector(2 downto 0):="000";
signal tempg7: bit_vector(2 downto 0):="000";
signal c0,c1,tempb1,tempc,tempc2,tempd2,tempd3,tempe3,tempe4,tempf4
        tempf5,tempg5,tempg6,temph6,temph7:bit:='0';
signal GG1,GG2,GG3,GG4,GG5,GG6,GG7,GG8:bit_vector(7 downto
        0):="00000000";
signal a11,b11: bit_vector(7 downto 0):="00000000";
signal P: bit_vector(15 downto 0):="0000000000000000";
for all: and_2 use entity work.and_2(behavior);
for all: xor_2 use entity work.xor_2(behavior);
for all: or_2 use entity work.or_2(behavior);
```

```
for all: d_latchN use entity work.d_latchN(behavior);
for all: d_latchP use entity work.d_latchP(behavior);
for all: FADD use entity work.FADD(structure);
for all: twobit_FA1 use entity work.twobit_FA1(structure);
for all: or_3 use entity work.or_3(behavior);
for all: xnor_2 use entity work.xnor_2(behavior);
for all: inverter use entity work.inverter(behavior);

    begin

    G0: for i in 0 to 7 generate

        d0: d_latchN
            generic map(delay8)
            port map(clk1, a1(i), a11(i));
        end generate;
    G141: for i in 0 to 7 generate
        d1: d_latchN
            generic map(delay8)
            port map(clk1, b1(i), b11(i));
        end generate;
    G1: for i in 0 to 7 generate
       u0: and_2
            generic map(delay1)
            port map(a11(i),b11(0),GG1(i));
        end generate;
    G2: for i in 0 to 7 generate
        u1: and_2
            generic map(delay1)
            port map(a11(i),b11(1),GG2(i));
        end generate;

    G3: for i in 0 to 7 generate
        u2: and_2
            generic map(delay1)
```

```
                port map(a11(i),b11(2),GG3(i));
        end generate;
G4: for i in 0 to 7 generate
        u3: and_2
                generic map(delay1)
                port map(a11(i),b11(3),GG4(i));
        end generate;
G5: for i in 0 to 7 generate
        u4: and_2
                generic map(delay1)
                port map(a11(i),b11(4),GG5(i));
        end generate;
G6: for i in 0 to 7 generate
        u5: and_2
                generic map(delay1)
                port map(a11(i),b11(5),GG6(i));
        end generate;
G7: for i in 0 to 7 generate
        u6: and_2
                generic map(delay1)
                port map(a11(i),b11(6),GG7(i));
        end generate;
G8: for i in 0 to 7 generate
        u7: and_2
                generic map(delay1)
                port map(a11(i),b11(7),GG8(i));
        end generate;
    P(0) <= GG1(0);
f1: twobit_FA1
    generic map(delay7)
    port map(GG2(1),c0,GG1(1),GG2(0),c1, P(1),tempa1(1),tempa1(2));
f2: twobit_FA1
    generic map(delay7)
    port map(GG3(1),GG2(2),GG3(0),GG1(2),c0,tempa2(0),tempa2(1),tempa2(2));
f3: twobit_FA1
```

```
        generic map(delay7)
        port map(GG2(3),GG4(1),GG4(0),GG1(3),c0,tempa3(0),tempa3(1),tempa3(2));
f4:     twobit_FA1
        generic map(delay7)
        port map(GG5(1),GG2(4),GG5(0),GG1(4),c0,tempa4(0),tempa4(1),tempa4(2));
f5:     twobit_FA1
        generic map(delay7)
        port map(GG6(1),GG2(5),GG6(0),GG1(5),c0,tempa5(0),tempa5(1),tempa5(2));
f6:     twobit_FA1
        generic map(delay7)
        port map(GG7(1),GG2(6),GG7(0),GG1(6),c0,tempa6(0),tempa6(1),tempa6(2));
f7:     twobit_FA1
        generic map(delay7)
        port map(GG8(1),GG2(7),GG8(0),GG1(7),c0,tempa7(0),tempa7(1),tempa7(2));
FA1:    FADD
                generic map(delay7)
                port map(tempa1(1),tempa2(0),c0, P(2), tempb1);
f8:     twobit_FA1
        generic map(delay7)
        port map(GG3(2),c0,tempa1(2),tempa2(1),tempa3(0),tempb2(0)
                tempb2(1),tempb2(2));
f9:     twobit_FA1
        generic map(delay7)
        port map(GG4(2),GG3(3),tempa2(2),tempa3(1),tempa4(0),tempb3(0)
                tempb3(1),tempb3(2));
f10:    twobit_FA1
        generic map(delay7)
        port map(GG5(2),GG3(4),tempa3(2),tempa4(1),tempa5(0),tempb4(0)
                tempb4(1),tempb4(2));
f11:    twobit_FA1
        generic map(delay7)
        port map(GG3(5),GG6(2),tempa4(2),tempa5(1),tempa6(0),tempb5(0),
                tempb5(1),tempb5(2));
```

```
f12:  twobit_FA1
      generic map(delay7)
      port map(GG7(2),GG3(6),tempa5(2),tempa6(1),tempa7(0),tempb6(0),
            tempb6(1),tempb6(2));
f13:  twobit_FA1
      generic map(delay7)
      port map(GG8(2),GG3(7),tempa6(2),tempa7(1), c0,tempb7(0)
            tempb7(1),tempb7(2));
z1:   xor_2
      generic map(delay4)
      port map(tempb1, tempb2(0),P(3));
u8:   and_2
      generic map(delay1)
      port map(tempb1, tempb2(0), tempc);
FA2:  FADD
      generic map(delay7)
      port map(tempc, tempb2(1), tempb3(0), P(4), tempc2);
f14:  twobit_FA1
      generic map(delay7)
      port map(GG4(3),c0,tempb2(2),tempb3(1), tempb4(0),tempc3(0)
            tempc3(1),tempc3(2));
f15:  twobit_FA1
      generic map(delay7)
      port map(GG5(3),GG4(4),tempb3(2),tempb4(1), tempb5(0),tempc4(0)
            tempc4(1),tempc4(2));
f16:  twobit_FA1
      generic map(delay7)
      port map(GG6(3),GG4(5),tempb4(2),tempb5(1), tempb6(0),tempc5(0)
            tempc5(1),tempc5(2));
f17:  twobit_FA1
      generic map(delay7)
      port map(GG7(3),GG4(6),tempb5(2),tempb6(1), tempb7(0),tempc6(0)
            tempc6(1),tempc6(2));
f18:  twobit_FA1
      generic map(delay7)
```

```
        port map(GG8(3),GG4(7),tempa7(2),tempb6(2), tempb7(1),tempc7(0)
                ,tempc7(1),tempc7(2));
z2:     xor_2
        generic map(delay4)
        port map(tempc2, tempc3(0),P(5));
u9:     and_2
        generic map(delay1)
        port map(tempc2, tempc3(0), tempd2);
FA3:    FADD
        generic map(delay7)
        port map(tempd2, tempc3(1), tempc4(0), P(6), tempd3);
f19:    twobit_FA1
        generic map(delay7)
        port map(GG5(4),c0,tempc3(2),tempc4(1), tempc5(0),tempd4(0)
                tempd4(1),tempd4(2));
f20:    twobit_FA1
        generic map(delay7)
        port map(GG6(4),GG5(5),tempc4(2),tempc5(1), tempc6(0),tempd5(0),
                tempd5(1),tempd5(2));
f21:    twobit_FA1
        generic map(delay7)
        port map(GG7(4),GG5(6),tempc5(2),tempc6(1), tempc7(0),tempd6(0),
                tempd6(1),tempd6(2));
f22:    twobit_FA1
        generic map(delay7)
        port map(GG8(4),GG5(7),tempc6(2),tempc7(1), tempb7(2),tempd7(0)
                tempd7(1),tempd7(2));
z3:     xor_2
        generic map(delay4)
        port map(tempd3, tempd4(0),P(7));
u10:    and_2
        generic map(delay1)
        port map(tempd3, tempd4(0), tempe3);

FA4:    FADD
```

```
        generic map(delay7)
        port map(tempe3, tempd4(1), tempd5(0), P(8), tempe4);
f23:    twobit_FA1
        generic map(delay7)
        port map(GG6(5),c0,tempd4(2),tempd5(1), tempd6(0),tempe5(0)
              ,tempe5(1),tempe5(2));
f24:    twobit_FA1
        generic map(delay7)
        port map(GG7(5),GG6(6),tempd5(2),tempd6(1), tempd7(0),tempe6(0)
              tempe6(1),tempe6(2));
f25:    twobit_FA1
        generic map(delay7)
        port map(GG8(5),GG6(7),tempd6(2),tempd7(1), tempc7(2),tempe7(0),
              tempe7(1),tempe7(2));
z4:     xor_2
        generic map(delay4)
        port map(tempe4, tempe5(0),P(9));
u11:    and_2
        generic map(delay1)
        port map(tempe4, tempe5(0), tempf4);
FA5:    FADD
        generic map(delay7)
        port map(tempf4, tempe5(1), tempe6(0), P(10), tempf5);
f26:    twobit_FA1
        generic map(delay7)
        port map(GG7(6),c0,tempe5(2),tempe6(1), tempe7(0),tempf6(0)
              tempf6(1),tempf6(2));
f27:    twobit_FA1
        generic map(delay7)
        port map(GG7(7),GG8(6),tempe6(2),tempe7(1), tempd7(2),tempf7(0),
              tempf7(1),tempf7(2));
z5:     xor_2
        generic map(delay4)
        port map(tempf5, tempf6(0),P(11));
```

```
u12:  and_2
      generic map(delay1)
      port map(tempf5, tempf6(0), tempg5);
FA6:  FADD
      generic map(delay7)
      port map(tempg5, tempf6(1), tempf7(0), P(12), tempg6);
f28:  twobit_FA1
      generic map(delay7)
      port map(GG8(7),c0,tempf6(2),tempf7(1), tempe7(2),tempg7(0),
      tempg7(1),tempg7(2));
z6:   xor_2
      generic map(delay4)
      port map(tempg6, tempg7(0),P(13));
u13:  and_2
      generic map(delay1)
      port map(tempg6, tempg7(0), temph6);
FA7:  FADD
      generic map(delay7)
      port map(temph6, tempg7(1), tempf7(2), P(14), temph7);
z7:   xor_2
      generic map(delay4)
      port map(tempg7(2), temph7,P(15));

G400: for i in 0 to 15 generate
      d3: d_latchP
            generic map(delay8)
            port map(clk1, P(i), product(i));
      end generate;

      ain1<= to_integer(a11);
      bin1 <= to_integer(b11);
      product1 <= to_integer(product);

end structure;
```

Twobit_FA1.vhd

```
entity twobit_FA1 is
        generic (delay1: in time);
        port(a,b,c,d,e: in bit; s0,s1: out bit; c1: inout bit);
end twobit_FA1;

architecture structure of twobit_FA1 is
        component FADD
                generic (delay: in time);
                port(a,b,c: bit; sum:out bit;carry: inout bit);
        end component;
signal temp_out:bit:='0';
for all: FADD use entity work.FADD(structure);

        begin
                FA1:    FADD
                        generic map(delay1)
                        port map(c,d,e,s0,temp_out);
                FA2:    FADD
                        generic map(delay1)
                        port map(a,b,temp_out,s1,c1);

end structure;
```

---

tb_five1.vhd

```
library XL;
use XL.XL_STD.all,STD.TEXTIO.all, XL.XL_IO.all;

entity tb_five1 is

end ;
```

```
architecture structure of tb_five1 is
        signal a,b: bit_vector(7 downto 0);
        signal ain,bin,product_I:integer:=0;
        signal clk:bit:='0';
        signal product_b:bit_vector(15 downto 0);

        component five_bit1
                generic(delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8 :in time);
                port(a1,b1:bit_vector(7 downto 0); clk1: in bit; Product: inout bit_vector(15
                        downto 0); ain1,bin1,product1: out integer);
        end component;

for xx1: five_bit1 use entity work.five_bit1(structure);

begin
        xx1: five_bit1
                generic map (0.3218 ns,0.5893 ns,0.6844 ns, 1.026 ns, 1.1042 ns,0.15
                                ns, 1.486 ns, 1.2 ns)
                port map (a,b, clk, product_b, ain, bin, product_I);

a <= "00000000","10000000" after 24 ns, "10000011" after 48 ns,"00000101" after 72
        ns, "11000000" after 96 ns, "00001100" after 120 ns, "00000001" after 144
        ns,"00000011" after 168 ns;
b <= "00000000","10000000" after 24 ns, "11100001" after 48 ns, "00000101" after 72
        ns, "11000110" after 96 ns, "00001101" after 120 ns, "00000001" after 144
        ns,"00000011" after 168 ns;
clk <='0', '1' after 22 ns,'0' after 24 ns, '1' after 46 ns, '0' after 48 ns, '1' after 70 ns,'0'
        after 72 ns, '1' after 94 ns, '0' after 96 ns, '1' after 118 ns, '0' after 120 ns, '1'
        after 142 ns, '0' after 144 ns, '1' after 166 ns, '0' after 168 ns;

-process
                variable L:line;
        begin
                write (L, " TIME   ain   bin   Product_I      Product_B");
                writeline (output, L);
```

```
                write (L, "   ---   ---   ---   ---");
                writeline (output, L);
                write (L, " ");
                writeline (output, L);
                wait;
end process;

        monitor_process: process(a,b)
                variable dline: line;

        begin
                write (dline, NOW, right, 7);
                write (dline, ain, right, 7);
                write (dline, bin, right, 7);
                write (dline, Product_I, right, 10);

                write (dline, Product_b, right, 20);
                writeline (output, dline);

        end process;

waves_monitor (to_integer(a),to_integer(b),to_integer(product_b));
end structure ;
```

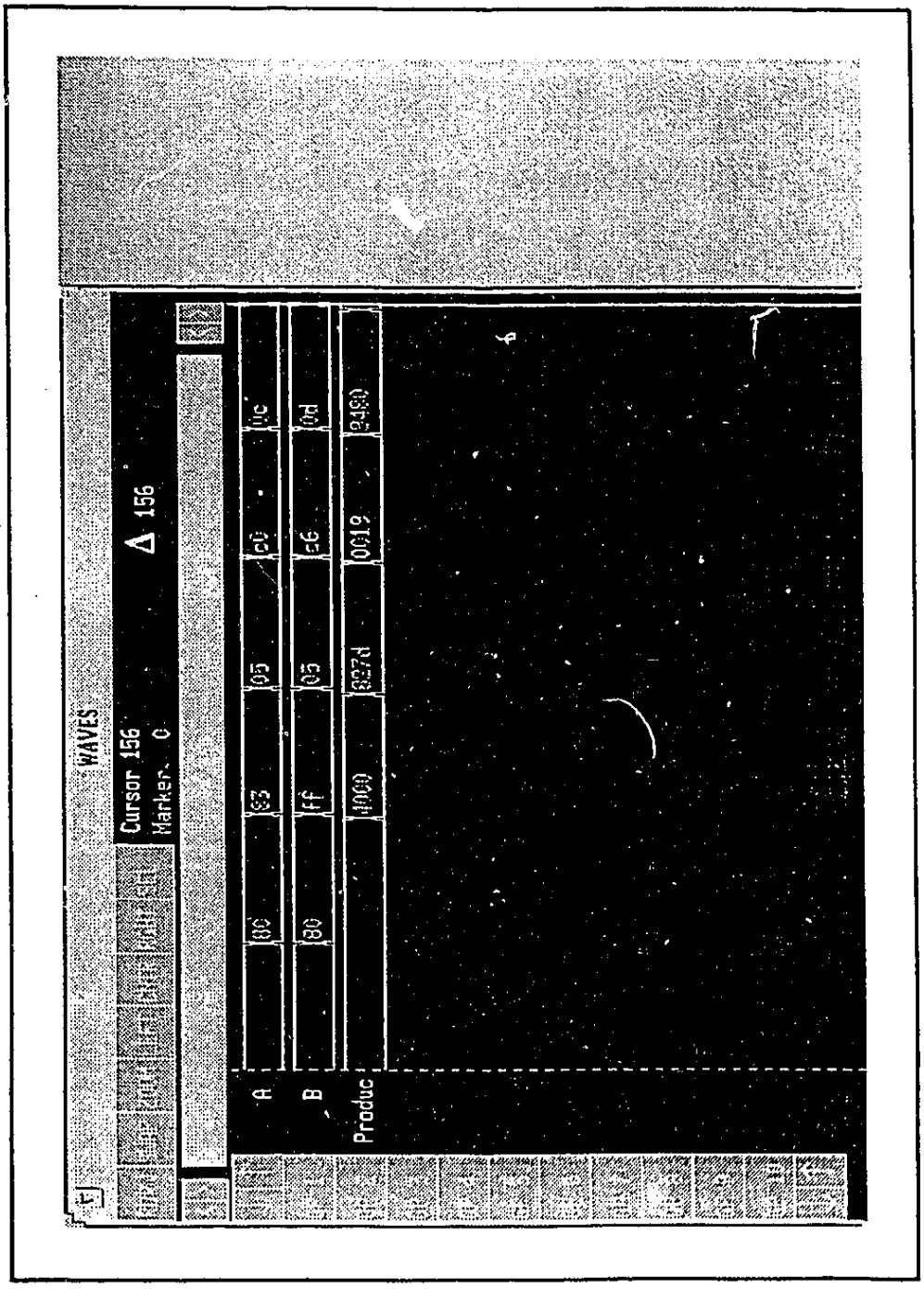


Figure 2C.1: VHDL Waveforms for Type_2 Two-Bit Full-Adder Multiplier

APPENDIX 1D

Architectural Model for Column Compression Multiplier

Ccmultp.vhd

```
Library XL;
use XL.XL_STD.all;
Library Mgates;
Use Mgates.all;
Library Madder;
Use Madder.all;

 entity dadda_multp2 is
        generic(delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8,delay9,delay10,
                delay11,delay12: in time);
        port(clk1: bit; a1,b1:in bit_vector(7 downto 0);  sum: inout bit_vector(14 downto
                0);T_sum,T_carry: inout bit_vector(10 downto 1); Product1:inout
                bit_vector(15 downto 0);Product: inout bit_vector(11 downto
                1);ain1,bin1,output:out integer);
end dadda_multp2;

architecture structure of dadda_multp2 is

        component and_2
                generic(delay : in time);
                port(a,b: in bit; c: out bit);
        end component;
        component xor_2
                generic (delay: in time);
                port(a,b: bit; c: out bit);
        end component;
        component or_2
                generic (delay: in time);
```

```
            port(a,b:bit; c: out bit);
        end component;
        component FADD
            generic(delay: in time);
            port(a,b,c: bit; sum,carry:out bit);
        end component;
        component Hadder
            generic(delay: in time);
            port(x,y:in bit; sum,carry:out bit);
        end component;
        component Tenbit_adder
            generic(delay1,delay2,delay3,delay4: in time);
            port(a,b: bit_vector(10 downto 1); c0: in bit;  sum: out bit_vector(11
            downto 1));
        end component;
        component canca
            port(producta: in bit_vector(11 downto 1); suma: in bit_vector(14 downto
                0); productb: inout bit_vector(15 downto 0));
        end component;
        component d_latchN
            generic(delay:in time);
            port(clk1,x: in bit; y:out bit);
        end component;
        component d_latchP
            generic(delay: in time);
            port(clk1,x:in bit; y: out bit);
        end component;

signal c01: bit:='0';
signal carry: bit_vector(14 downto 0);
signal GG1,GG2,GG3,GG4,GG5,GG6,GG7,GG8: bit_vector(7 downto
        0):="00000000";
signal tsa2,tsa3,tsa4,tsa5,tsa6,tsa9,tsa10: bit:='0';
signal tca1,tca2,tca3,tca4,tca5,tca6,tca9,tca10:bit:='0';
signal tsa7,tsa8,tca7,tca8: bit_vector(1 downto 0):="00";
```

```
signal tsb3,tcb3,tcb2,tsb4,tsb5,tsb7,tsb9,tsb10,tsb11:bit:='0';
signal tcb4,tcb5,tcb7,tcb9,tcb10,tcb11:bit:='0';
signal tsb6,tcb6,tsb8,tcb8:bit_vector(1 downto 0):="00";
signal tsc4,tsc5,tsc6,tsc8,tsc10,tsc11,tsc12:bit:='0';
signal tcc3,tcc4,tcc5,tcc6,tcc8,tcc10,tcc11,tcc12:bit:='0';
signal tsc9,tcc9,tsc7,tcc7: bit_vector(1 downto 0):="00";
signal tsc14,tsc15:bit :='0';
signal a1l,b1L: bit_vector(7 downto 0):="00000000";
signal suml: bit_vector(14 downto 0):="000000000000000";
signal carry_templ, sum_templ: bit_vector(10 downto 1):="0000000000";
signal g,gg,p,gp: bit_vector(10 downto 1):="0000000000";
signal sum_temp,carry_temp: bit_vector(10 downto 1):="0000000000";
signal temp1,temp2,temp3,carry_temp1: bit_vector(4 downto 1):="0000";
signal temp4:bit:='0';

for all: and_2 use entity work.and_2(behavior);
for all: xor_2 use entity work.xor_2(behavior);
for all: or_2 use entity work.or_2(behavior);
for all: d_latchN use entity work.d_latchN(behavior);
for all: d_latchP use entity work.d_latchP(behavior);
for all: FADD use entity work.FADD(structure);
for all: hadder use entity work.hadder(structure);
for cc1: canca use entity work.cancation(behavior);

begin

G40: for i in 0 to 7 generate
        d0: d_latchN
                generic map(delay2)
                port map(clk1, a1(i), a1(i));
        end generate;

G41: for i in 0 to 7 generate
        d1: d_latchN
                generic map(delay2)
```

```
            port map(clk1, b1(i), b11(i));
        end generate;

G0: for i in 0 to 7 generate
        u0: and_2
            generic map(delay1)
            port map(a11(i),b11(0),GG1(i));
        end generate;

G1: for i in 0 to 7 generate
        u1: and_2
            generic map(delay1)
            port map(a11(i),b11(1),GG2(i));
        end generate;

G2: for i in 0 to 7 generate
        u2: and_2
            generic map(delay1)
            port map(a11(i),b11(2),GG3(i));
        end generate;

G3: for i in 0 to 7 generate
        u3: and_2
            generic map(delay1)
            port map(a11(i),b11(3),GG4(i));
        end generate;
G4: for i in 0 to 7 generate
        u4: and_2
            generic map(delay1)
            port map(a11(i),b11(4),GG5(i));
        end generate;
G5: for i in 0 to 7 generate
        u5: and_2
            generic map(delay1)
            port map(a11(i),b11(5),GG6(i));
```

```
        end generate;
G6: for i in 0 to 7 generate
        u6: and_2
                generic map(delay1)
                port map(a1l(i),b1l(6),GG7(i));
        end generate;

G7: for i in 0 to 7 generate
        u7: and_2
                generic map(delay1)
                port map(a1l(i),b1l(7),GG8(i));
        end generate;

HA1a: hadder
                generic map(delay3)
                port map(GG1(1), GG2(0), sum(1),tca1);
FA2a: FADD
                generic map(delay3)
                port map(GG2(1),GG1(2),GG3(0),tsa2,tca2);
FA3a: FADD
                generic map(delay3)
                port map(GG3(1),GG1(3),GG4(0),tsa3,tca3);
FA4a: FADD
                generic map(delay3)
                port map(GG4(1),GG1(4),GG5(0),tsa4,tca4);
FA5a: FADD
                generic map(delay3)
                port map(GG5(1),GG6(0),GG1(5),tsa5,tca5);
FA6a: FADD
                generic map(delay3)
                port map(GG4(3),GG1(6),GG7(0),tsa6,tca6);
FA7a: FADD
                generic map(delay3)
                port map(GG3(5),GG6(2),GG2(6),tsa7(0),tca7(0));
FA7a1: FADD
```

```
                generic map(delay3)
                port map(GG7(1),GG8(0),GG1(7),tsa7(1),tca7(1));
FA8a: FADD
                generic map(delay3)
                port map(GG7(2),GG2(7),GG8(1),tsa8(0),tca8(0));
FA8a1: FADD
                generic map(delay3)
                port map(GG4(5),GG6(3),GG3(6),tsa8(1),tca8(1));
FA9a: FADD
                generic map(delay3)
                port map(GG7(3),GG3(7),GG8(2),tsa9,tca9);
FA10a: FADD
                generic map(delay3)
                port map(GG8(3),GG4(7),GG7(4),tsa10,tca10);
HA2b: hadder
                generic map(delay3)
                port map(tca1,tsa2,sum(2),tcb2);
FA3b: FADD
                generic map(delay3)
                port map(GG2(2),tca2,tsa3,tsb3,tcb3);
FA4b: FADD
                generic map(delay3)
                port map(GG2(3),tca3,tsa4,tsb4,tcb4);
FA5b: FADD
                generic map(delay3)
                port map(GG2(4),tca4,tsa5,tsb5,tcb5);
FA6b: FADD
                generic map(delay3)
                port map(GG3(4),tca5,GG5(2),tsb6(0),tcb6(0));
FA6b1: FADD
                generic map(delay3)
                port map(GG6(1),GG2(5),tsa6,tsb6(1),tcb6(1));
FA7b: FADD
                generic map(delay3)
                port map(tca6,tsa7(0),tsa7(1),tsb7,tcb7);
```

```
HA8b: hadder
        generic map(delay3)
        port map(tca7(0),tca7(1),tsb8(0),tcb8(0));
FA8b: FADD
        generic map(delay3)
        port map(tsa8(0),tsa8(1),GG5(4),tsb8(1),tcb8(1));
FA9b: FADD
        generic map(delay3)
        port map(tca8(0),tca8(1),tsa9,tsb9,tcb9);
FA10b: FADD
        generic map(delay3)
        port map(tca9,tsa10,GG5(6),tsb10,tcb10);
FA11b: FADD
        generic map(delay3)
        port map(tca10,GG5(7),GG8(4),tsb11,tcb11);
HA3c: hadder
        generic map(delay3)
        port map(tcb2,tsb3,sum(3),tcc3);
HA4c: hadder
        generic map(delay3)
        port map(tcb3,tsb4,tsc4,tcc4);
FA5c: FADD
        generic map(delay3)
        port map(tcb4,tsb5,GG4(2),tsc5,tcc5);
FA6c: FADD
        generic map(delay3)
        port map(tcb5,tsb6(0),tsb6(1),tsc6,tcc6);
FA7c: FADD
        generic map(delay3)
        port map(tcb6(0),GG5(3),GG4(4),tsc7(0),tcc7(0));
HA7c: hadder
        generic map(delay3)
        port map(tcb6(1),tsb7,tsc7(1),tcc7(1));
FA8c: FADD
        generic map(delay3)
```

```
                port map(tcb7,tsb8(0),tsb8(1),tsc8,tcc8);
FA9c: FADD
                generic map(delay3)
                port map(tcb8(0),GG5(5),GG6(4),tsc9(0),tcc9(0));
FA9c1: FADD
                generic map(delay3)
                port map(tcb8(1),tsb9,GG4(6),tsc9(1),tcc9(1));
FA10c: FADD
                generic map(delay3)
                port map(tcb9,tsb10,GG6(5),tsc10,tcc10);
FA11c: FADD
                generic map(delay3)
                port map(tcb10,tsb11,GG6(6),tsc11,tcc11);
FA12c: FADD
                generic map(delay3)
                port map(tcb11,GG6(7),GG8(5),tsc12,tcc12);
FA4d: FADD
                generic map(delay3)
                port map(tcc3,tsc4,GG3(2),sum(4),carry(4));
FA5d: FADD
                generic map(delay3)
                port map(tcc4,tsc5,GG3(3),sum(5),carry(5));
HA6d: hadder
                generic map(delay3)
                port map(tcc5,tsc6,sum(6),carry(6));
FA7d: FADD
                generic map(delay3)
                port map(tcc6,tsc7(0),tsc7(1),sum(7),carry(7));
FA8d: FADD
                generic map(delay3)
                port map(tcc7(0),tcc7(1),tsc8,sum(8),carry(8));
FA9d: FADD
                generic map(delay3)
                port map(tcc8,tsc9(0),tsc9(1),sum(9),carry(9));
FA10d: FADD
```

```
            generic map(delay3)
            port map(tcc9(0),tcc9(1),tsc10,sum(10),carry(10));
FA11d: FADD
            generic map(delay3)
            port map(tcc10,tsc11,GG7(5),sum(11),carry(11));
FA12d: FADD
            generic map(delay3)
            port map(tcc11,tsc12,GG7(6),sum(12),carry(12));
FA13d: FADD
            generic map(delay3)
            port map(tcc12,GG7(7),GG8(6),sum(13),carry(13));

FA14d: FADD
            generic map(delay3)
            port map(tsc14,tsc15,GG8(7),sum(14),carry(14));

            sum(0) <= GG1(0);
            T_sum <= sum(14 downto 5);
            T_carry <= carry(13 downto 4);
x1: FADD
            generic map(delay3)
            port map(T_sum(1),T_carry(1),c01,sum_temp(1), carry_temp(1));
x2: FADD
            generic map(delay3)
            portmap(T_sum(2),T_carry(2),carry_temp(1),sum_temp(2),
                  carry_temp(2));

G30:for i in 3 to 10 generate

      u11: and_2
            generic map(delay1)
            port map(T_sum(i), T_carry(i),g(i));
      z5: xor_2
            generic map(delay5)
            port map(T_sum(i), T_carry(i), p(i));
```

```
        end generate;

G31:    for i in 1 to 4 generate

         u12: and_2
                generic map(delay1)
                port map(p(2*i+2), g(2*i+1),temp1(i));
         y1: or_2
                generic map(delay4)
                port map(g(2*i+2),temp1(i),gg(2*i+2));

         u13: and_2
                generic map(delay1)
                port map(p(2*i+2), p(2*i+1), gp(2*i+2));
        end generate;

G33:    for i in 1 to 4 generate
         u14: and_2
                generic map(delay1)
                port map(gp(2*i+2), carry_temp(2*i),carry_temp1(i));
          y2: or_2
                generic map(delay4)
                port map(carry_temp1(i),gg(2*i+2),carry_temp(2*i+2));
        end generate;

G34:    for i in 1 to 3 generate
        z3: xor_2
                generic map(delay5)
                port map(p(2*i+1), carry_temp(2*i), sum_temp(2*i+1));
         u15: and_2
                generic map(delay1)
                port map(p(2*i+1), carry_temp(2*i), temp2(i));
        y3: or_2
                generic map(delay4)
                port map(temp2(i), g(2*i+1),temp3(i));
```

```
        z2: xor_2
                generic map(delay5)
                port map(p(2*i+2), temp3(i),sum_temp(2*i+2));
        end generate;
        x3: FADD
                generic map(delay3)
                port map(T_sum(9),T_carry(9),carry_temp(8),sum_temp(9),
                        carry_temp(9));
        u16: and_2
                generic map(delay1)
                port map(p(9), carry_temp(8), temp4);

        y4: or_2
                generic map(delay4)
                port map(temp4, g(9),Carry_temp(9));
        x4: FADD
                generic map(delay3)
                port map(T_sum(10),T_carry(10),Carry_temp(9),sum_temp(10),
                        carry_temp(10));

G42:for i in 1 to 10 generate
        d3: d_latchP
                generic map(delay2)
                port map(clk1, carry_temp(i), carry_templ(i));
        end generate;

G43: for i in 1 to 10 generate
        d4: d_latchP
                generic map(delay2)
                port map(clk1, sum_temp(i),sum_templ(i));
        end generate;

G44:    for i in 0 to 14 generate
        d5: d_latchP
                generic map(delay2)
```

```
            port map(clk1, sum(i), suml(i));
        end generate;

        Product<= carry_templ(10)&sum_templ ;

            ain1<= to_integer(a11);
            bin1 <= to_integer(b11);
cc1: canca
            port map(product,suml,product1);

            output <= to_integer(product1);
end structure;
```

---

tb_Ccmultp.vhd

```
library XL;
use XL.XL_STD.all,STD.TEXTIO.all, XL.XL_IO.all;
entity tb_dadda is
end tb_dadda;

architecture structure of tb_dadda is

    component dadda_multp2
        generic(delay1,delay2,delay3,delay4,delay5,delay6,delay7,delay8
            ,delay9,delay10,delay11,delay12: in time);
        port(clk1: bit;a1,b1: in bit_vector(7 downto 0); sum: inout bit_vector(14
            downto 0);T_sum,T_carry : inout bit_vector(10 downto 1);
            product1:inout bit_vector(15 downto 0);Product: inout bit_vector(11
            downto 1); ain1,bin1,output: out integer);
    end component;

signal ain,bin,Product_I:integer:=0;
```

```
signal product_b: bit_vector(15 downto 0):="0000000000000000";
signal a,b: bit_vector(7 downto 0):="00000000";
signal carry1,sum1: bit_vector(14 downto 0):="000000000000000";
signal cin,clk: bit:='0';
signal T_sum1,T_carry1:bit_vector(10 downto 1):="0000000000";
signal even_carry1: bit_vector(5 downto 1);
signal Producta: bit_vector(11 downto 1):="00000000000";

for x1: dadda_multp2 use entity work.dadda_multp2(structure);

        begin

        x1: dadda_multp2
                generic map(0.3218 ns, 1.2 ns, 1.486 ns, 0.5893 ns, 1.026 ns, 2.972 ns,
                                1.486 ns, 11.405 ns, 9.919 ns, 8.433 ns, 6.947 ns, 5.416
                                ns)
                port map(clk,a,b,sum1,T_sum1,T_carry1,product_b,Producta
                                ,ain,bin,Product_I);

a<= "10000000","11111111" after 16 ns, "10000011" after 32 ns, "10000011" after 48
ns,"10000111" after 64 ns,"11000011" after 80 ns, "11111111" after 96 ns, "11100001"
after 112 ns, "10000000" after 128 ns;

b <= "10000000","11111111" after 16 ns, "11000011" after 32 ns, "11100001" after 48
ns,"10000001" after 64 ns,"11001000" after 80 ns, "10000011" after 96 ns, "11100001"
after 112 ns, "10000000" after 128 ns;

clk <= '0' , '1' after 14 ns, '0' after 16 ns, '1' after 30 ns, '0' after 32 ns, '1' after 46 ns,
'0' after 48 ns, '1' after 62 ns, '0' after 64 ns, '1' after 78 ns, '0' after 80 ns, '1' after 94
ns, '0' after 96 ns, '1' after 110 ns, '0' after 112 ns;


process
                variable L:line;
        begin
```

```
                write (L, "  TIME    ain    bin     product_b      product_I ");
                writeline (output, L);
                write (L, "  ---     ---    ---         ---           -----  ");
                writeline (output, L);
                write (L, " ");
                writeline (output, L);
                wait;
end process;

        monitor_process: process(ain,bin)
                variable dline: line;
        begin
                write (dline, NOW, right, 7);
                write (dline, ain, right, 7);
                write (dline, bin, right, 10);
                write (dline, product_b, right, 20);
                write (dline, product_I, right, 15);
                writeline (output, dline);

        end process;

waves_monitor (to_integer(a),to_integer(b),to_integer(product_b));

end structure ;
```

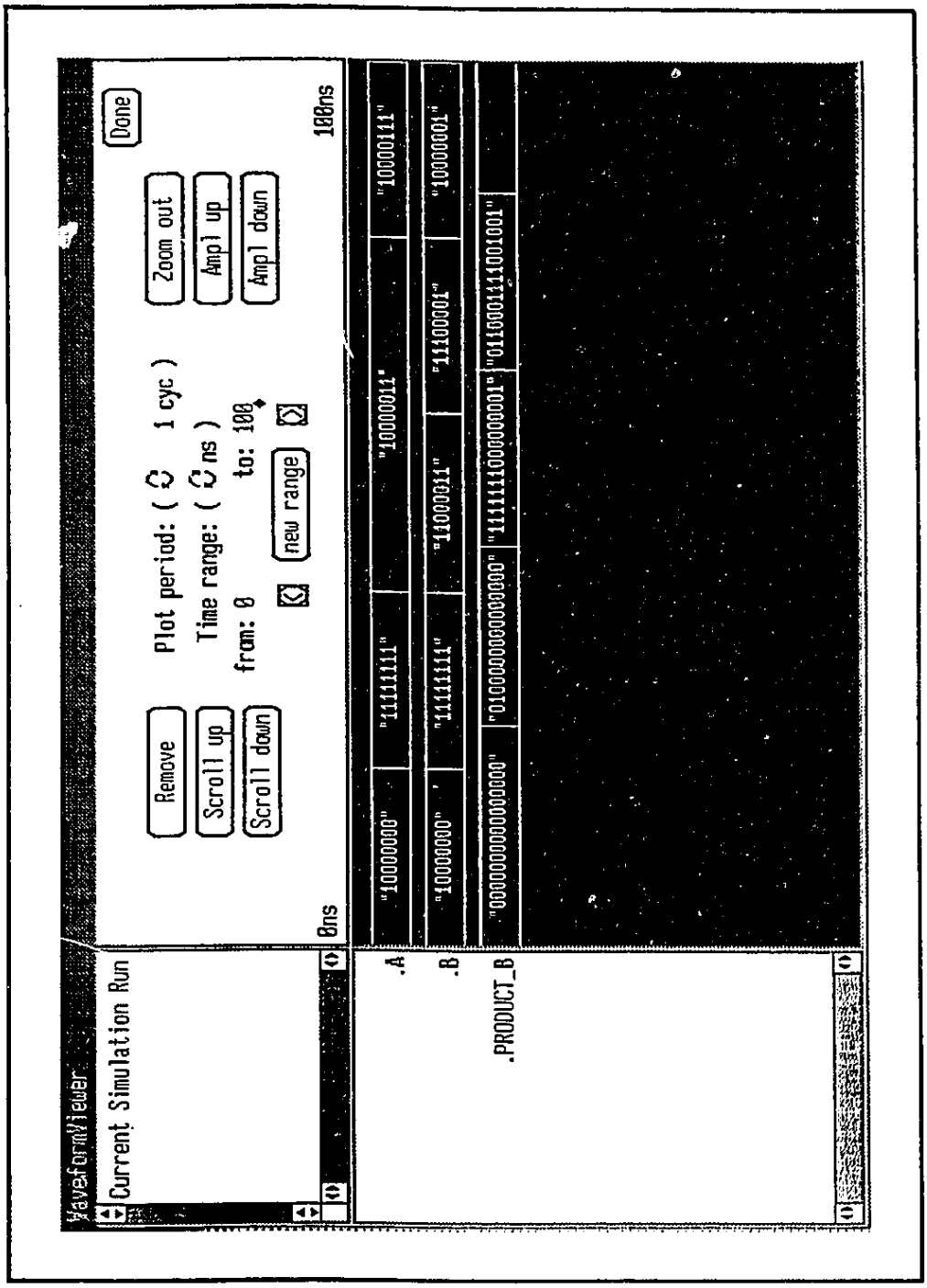


Figure D1.1: VHDL Waveforms for a Column Compression Multiplier

APPENDIX 1E

HIGH SPEED COMPACT 10-BIT CMOS ADDER IN MULTIPLE OUTPUT DOMINO LOGIC

This appendix provides the simulation results and the layout of the subcomponents for the new adder described in chapter 4. In addtion, the clocking scheme and distribution used in the design of the adder will be described.

E.1 INTRODUCTION

Performing addition rapidly and with a minimum investment in equipment has been a continuing challenge to computer designers. The essential nature of carry propagation is the most difficult problem in speeding up addition, and a variety of ways have been devised for coping with it such as Carry Completion Sensing, Conditional Sum adder, Carry Minimization and Carry Look Ahead(CLA).

The CLA technique has been widely used in most of the arithmetic units. The principle of the CLA is that instead of propagating carries through the adder stages sequentially, one provides supplemental circuitry that forms a carry signal into each adder stage if its predecessor generated a carry or if its predecessor propagated a carry generated two stages previously. All of these actions go on simultaneously and carry propagation thus becomes concurrent instead of sequential. In order to further enhance the performance of the CPA a new carry propagate adder has been designed in this work using MODL gates.

E.2 MULTIPLE OUTPUT DOMINO LOGIC (MODL)

In order to improve the area and speed of CMOS logic while retaining its lower power characteristic, many of the recent CMOS logic styles exploit non-complementary circuit structures and dynamic circuit operations. In particular the domino technique is very valuable for arithmetic and other circuit involving complex gates with high fan-in and high fan-out. This is so in spite of the fact that it can provide only non-inverting gates. In addition domino circuits are more stable than other unbuffered dynamic circuits.

E.3 Description Of MODL

In domino CMOS logic as well as other noncomplementary MOS logic styles, there is only one output available from a given logic gate. This is in spite of the fact that multiple functions are often implemented in the logic tree with one being the subfunction of the another. Therefore if one or more of these subfunctions are needed as separate output signals they have to be implemented in several additional gates resulting in replication of circuitry. For example in the domino circuit shown in Figure E.1 the function f and f2 require the implementation of two different gates with logic tree for f2 duplicated.

The main concept behind MODL is the utilization of subfunction available in the logic tree of domino gates thus saving replication of circuitry. The additional outputs are obtained by adding precharge devices and static inverter at the corresponding intermediate nodes of the logic tree. The MODL circuit for f and f2 shown in Figure E.2 illustrates the concept and construction, producing both functions from single gate without duplicating the logic tree for f2.

In addition, since nodes internal to the logic tree are being precharged for functional purposes, MODL is by construction considerably less susceptive to charge sharing than standard domino. Overall use of MODL can reduce silicon area, increase circuit performance and decrease power, because of reduction of device count, wire length, and consequently output loading.

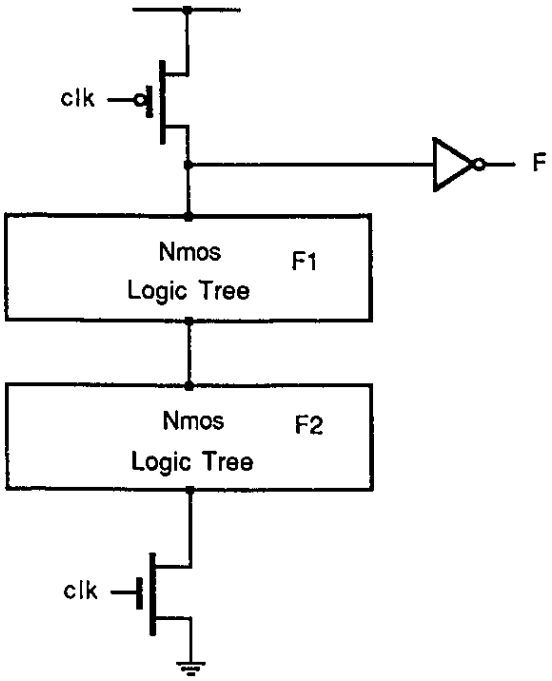


Figure E.1: Domino Implementation of F with F=F1F2

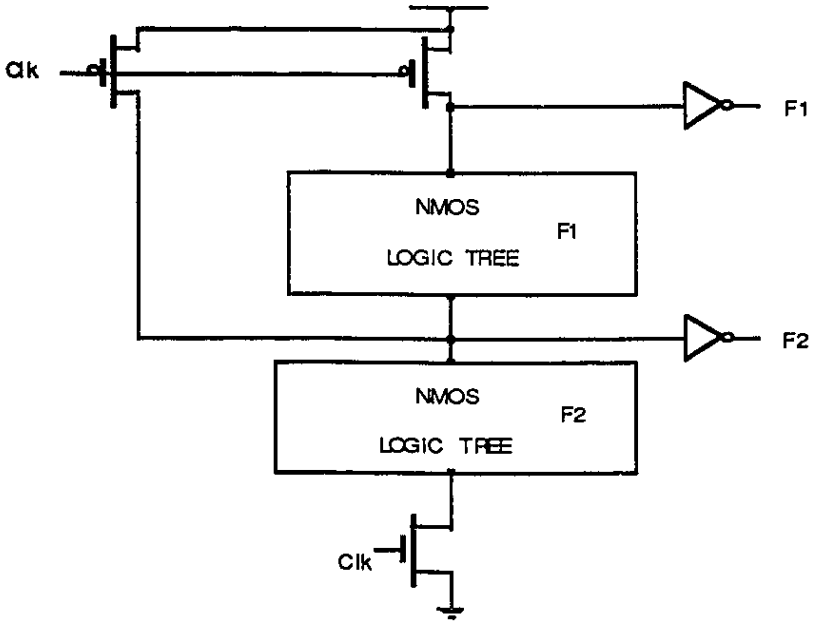


Figure E.2: MODL Implementation of the same function

E.4 10-Bit MODL Adder

A. Circuit Description

The organization of a 10 bit adder using both the original structure designed by I.Hwang[E1] and the modified structure are shown in Figure E.3 and Figure E.4 respectively.

As shown in Figure E.3 and Figure E.4, unlike the original architecture, the modified architecture only generates even carries and has less stages. Therefore the new or modified adder architecture has a better performance than its counterpart.

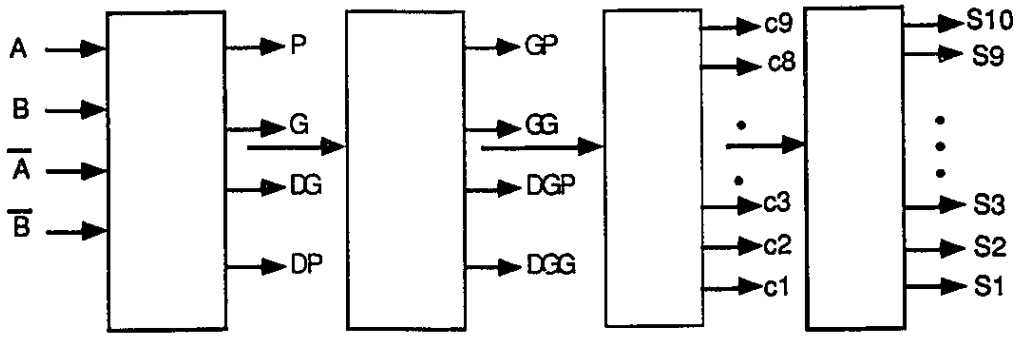


Figure E.3: Structure of the Original Adder

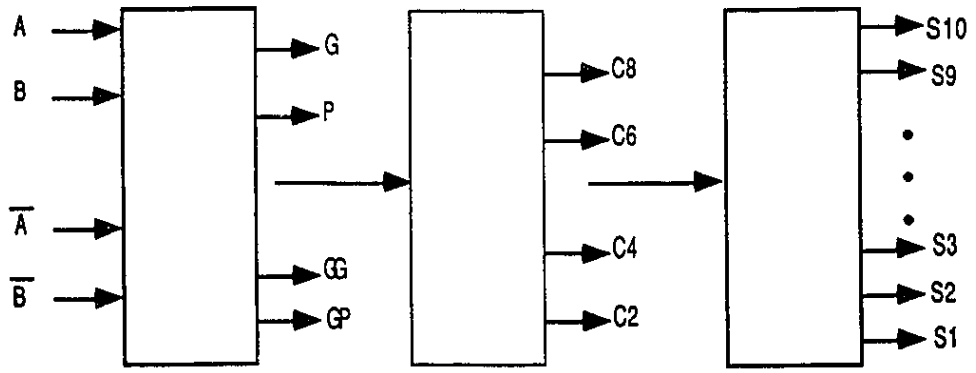


Figure E.4: Structure of the Modified Adder

The first stage of the new adder stage is a 4-bit wide unit which produces 1 and 2 bit generate (g_i and gg_i) and propagate terms(p_i and gp_i).

where

$g_i = a_i$ **and** b_i

$gg_{2i} = [(a_{2i}$ **xor** $b_{2i})$ **and** $g_i]$ **or** $(a_{2i}$ and $b_{2i})$

$pi = a_i$ **xor** b_i

$gp_{2i} = p_i$ **and** p_{2i}

The second stage takes its inputs from the first stage and produces only even carries. The final stage accepts its inputs from the first stage and second stage and gives the final sum of the input operands.

where

$C_{2i+2} = [gp_{2i+2}$ **and** $c_{2i}\ ]$ **or** gg_{2i+2}

$S_{2i+1} = p_{2i+1}$ **xor** c_{2i}

$S_{2i+2} = p_{2i+2}$ **xor** $((p_{2i}+1$ and $c_{2i})$ **or** $g_{2i+1})$

The recurrent nature of the above logic functions makes the carry propagate adder suitable for multiple output domino logic. One of the drawbacks of the domino logic as mentioned in the previous section is that it can provide only non-inverting gates. Therefore modified Differential Cascode Voltage Switch(DCVS) logic was used to realize the above Boolean

functions to accommodate both true and complement signals. The circuit equivalent of the functions are illustrated in chapter 4.

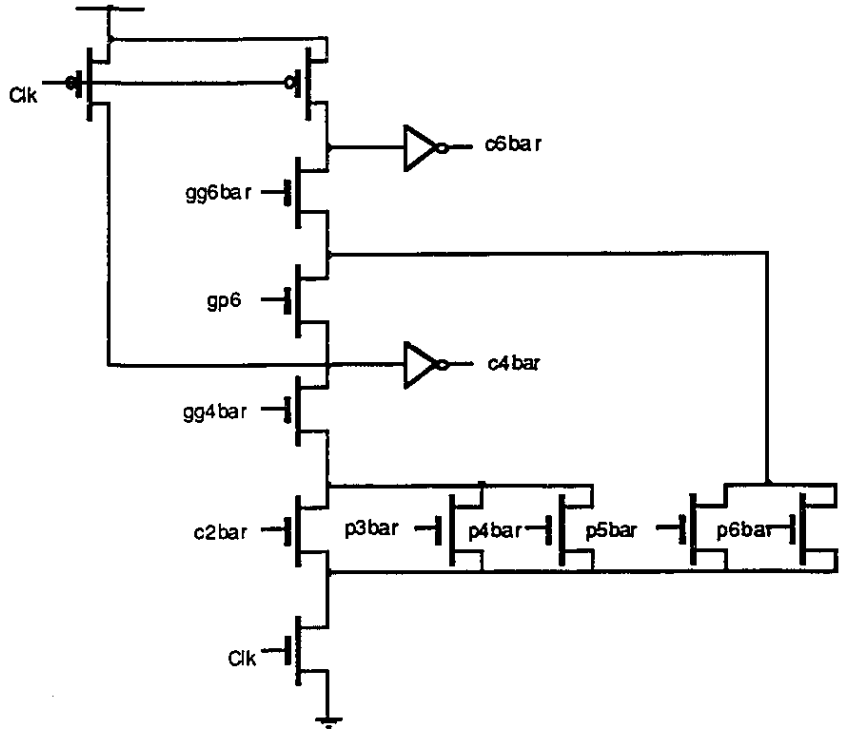


Figure E.5: MODL Gate for Carry_bar Generator

In constructing and cascading OR-AND (Sum of Products) forms of the MODL gates some care is required to prevent false discharge at a lower AND dynamic output node when a higher OR dynamic node is pulled down. This occurs because a reverse current path can be established from the lower node through the higher node to ground depending on inputs to the logic tree. In the design of the circuit shown in Figure E.5 Boolean Simplification Theorm[E2] was used to avoid the above mentioned problem.

E.5 Clocking Scheme and Distribution

In previous sections the architecture and design of the carry propagate adder using MODL gates was discussed. To achieve optimal performance a true single phase clocking scheme [E3] and true single phase latch shown in Figure E.6 were used. This clocking scheme minimizes clock skew problem which is a major problem in two and single phase clocking schemes. In order to further reduce the clock skew problem the clock distribution

shown in Figure E.7 was used in this design. Distributing the load across several buffers serves to shorten clock lines and reduce the maximum RC delay, both of which help keep the clock edges sharp.

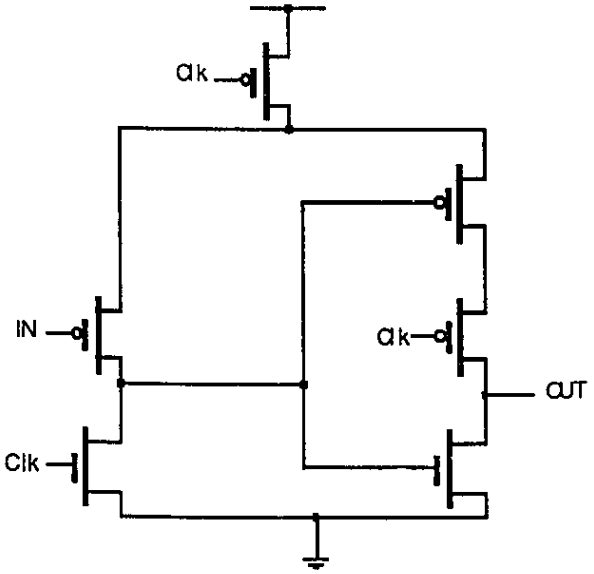


Figure E.6: True Single Phase P_Latch

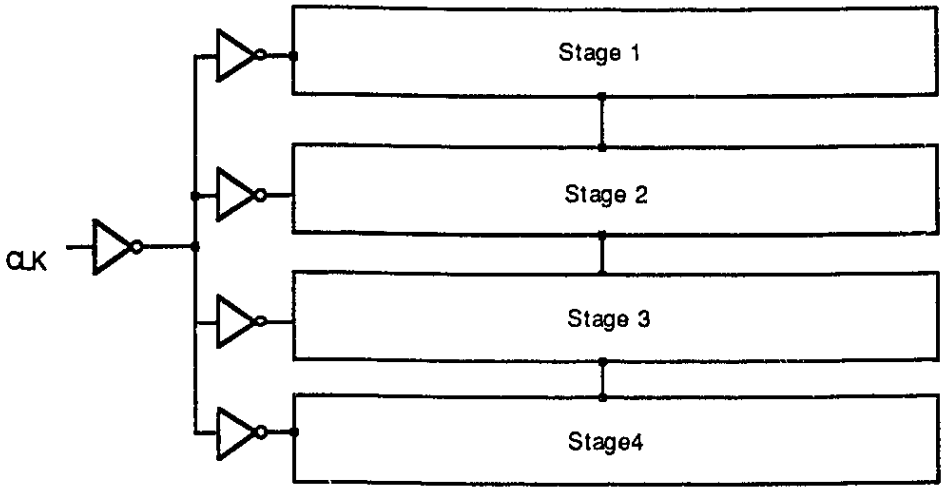


Figure E.7: Clock Distribution

E.6 Simulation Results

The functionality of the new adder was fully verified using VHDL as shown in Figure 2B.1. The adder was also simulated successfully using HSPICE in both 1.2μm and 3μm CMOS technology. The simulation results of the adder is shown in Figure E.8. The circuit was found to be operating at a maximum frequency of 70Mhz by applying the device sizing method developed by Sammy Bizzan.

E.7 CHIP IMPLEMENTATION

The new adder was implemented in a standard 3μm CMOS two level metal CMOS technology. All the cells are fully custom design. The adder circuit has 458 transistors in an active area 5.8mm x 4.1mm. The layout for the subcomponents of the chip are shown in Figure E.9 to E.14.

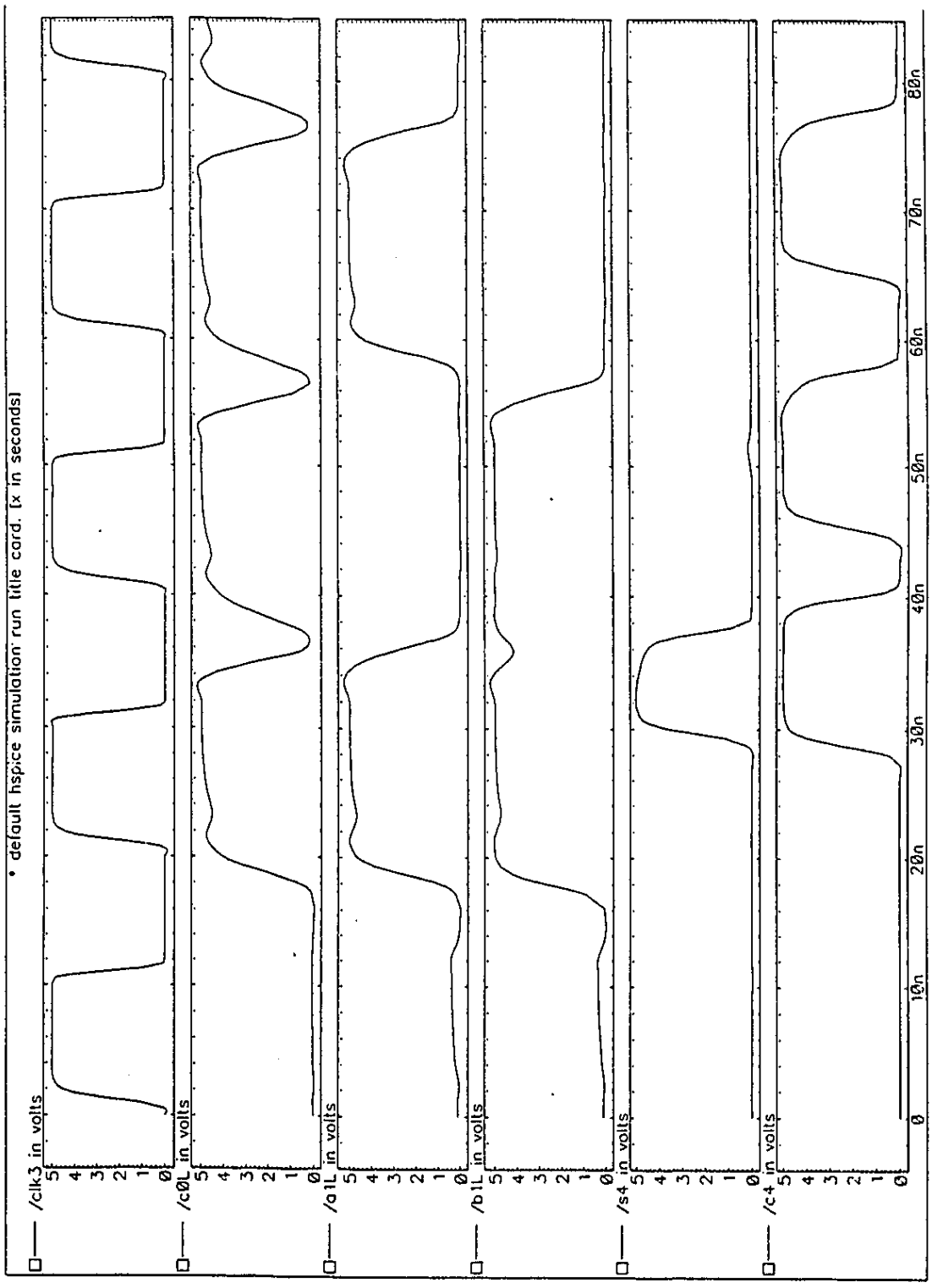


Figure E.8: Simulation Result for the 10-Bit Adder

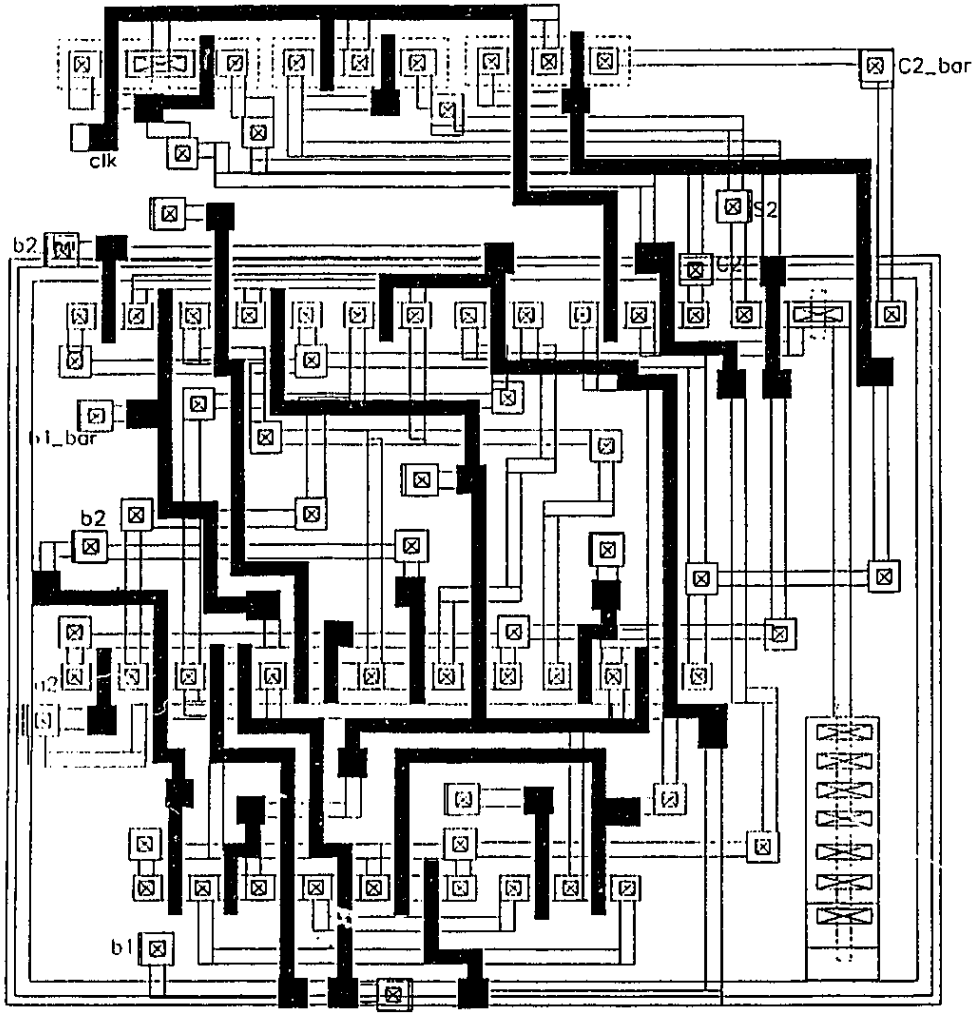


Figure E.9: Layout for S2 and C2 Block

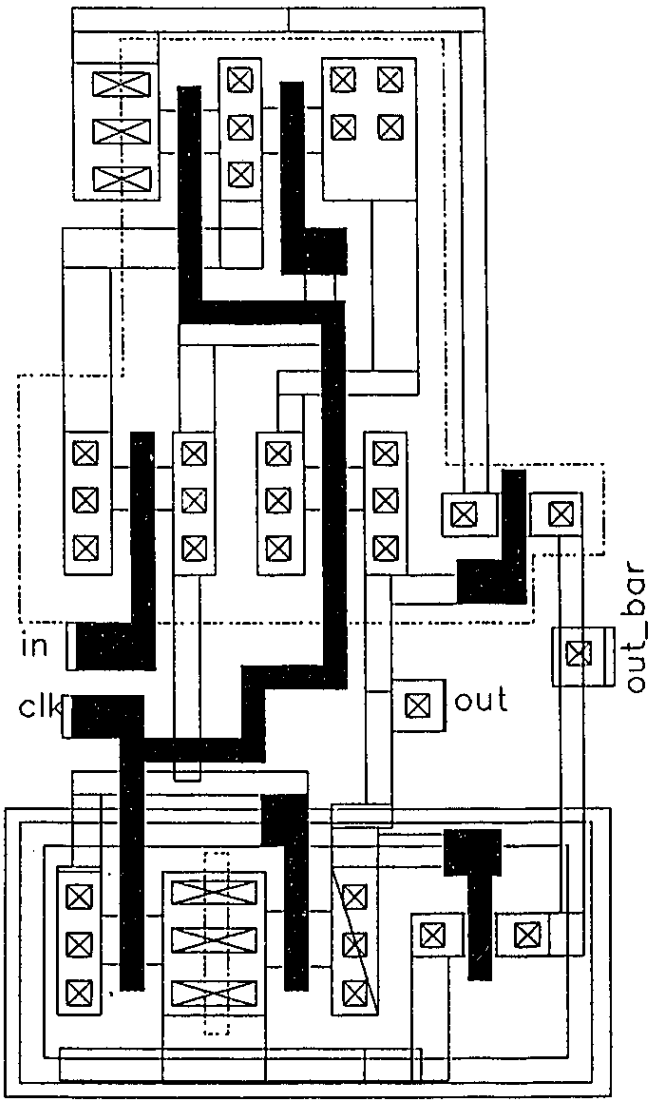


Figure E.10: Layout for True Single Phase Latch

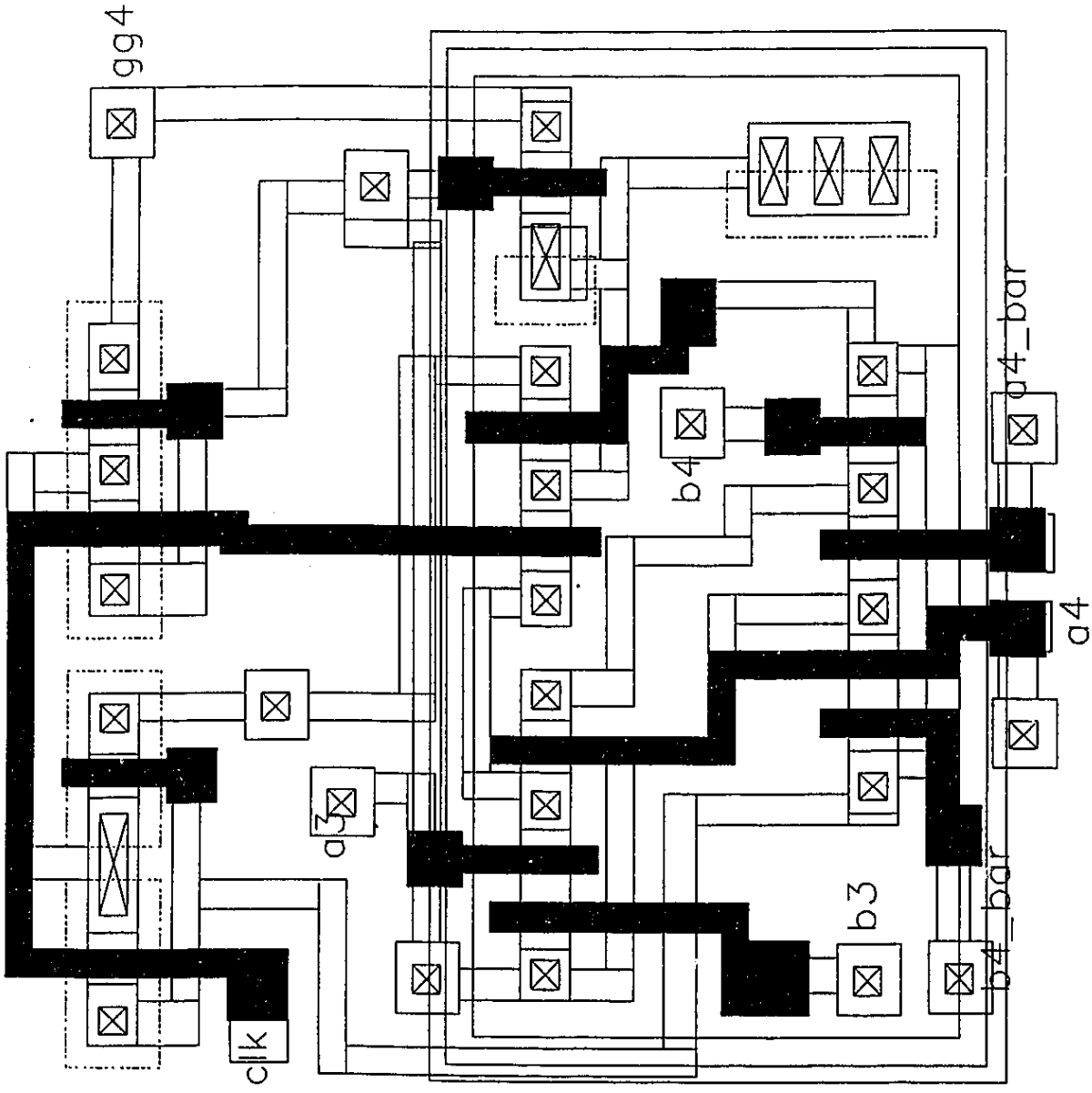


Figure E.11: Layout for Generate and Group Generate Block

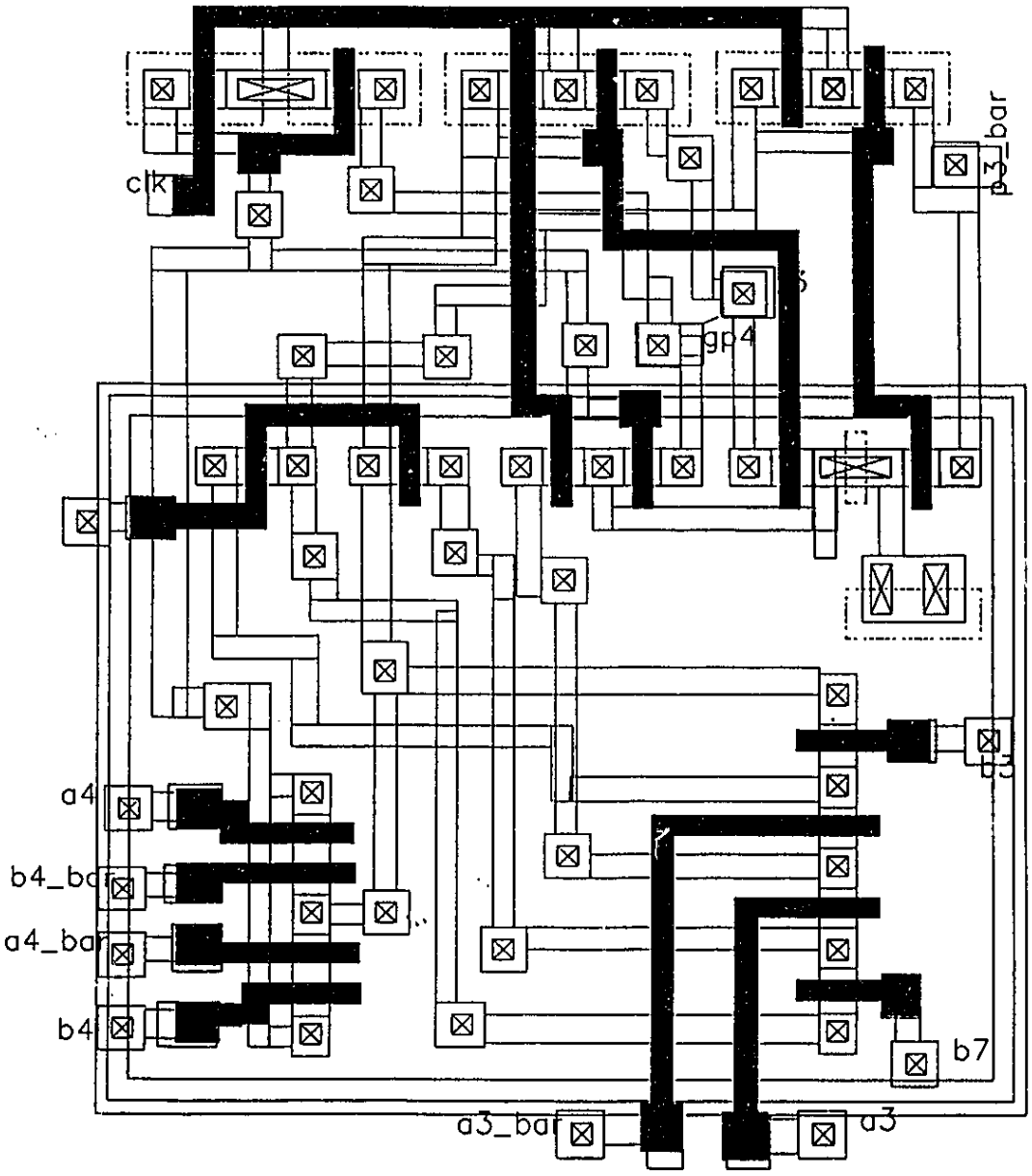


Figure E.12: Layout for Propagate and Group Propagate Block

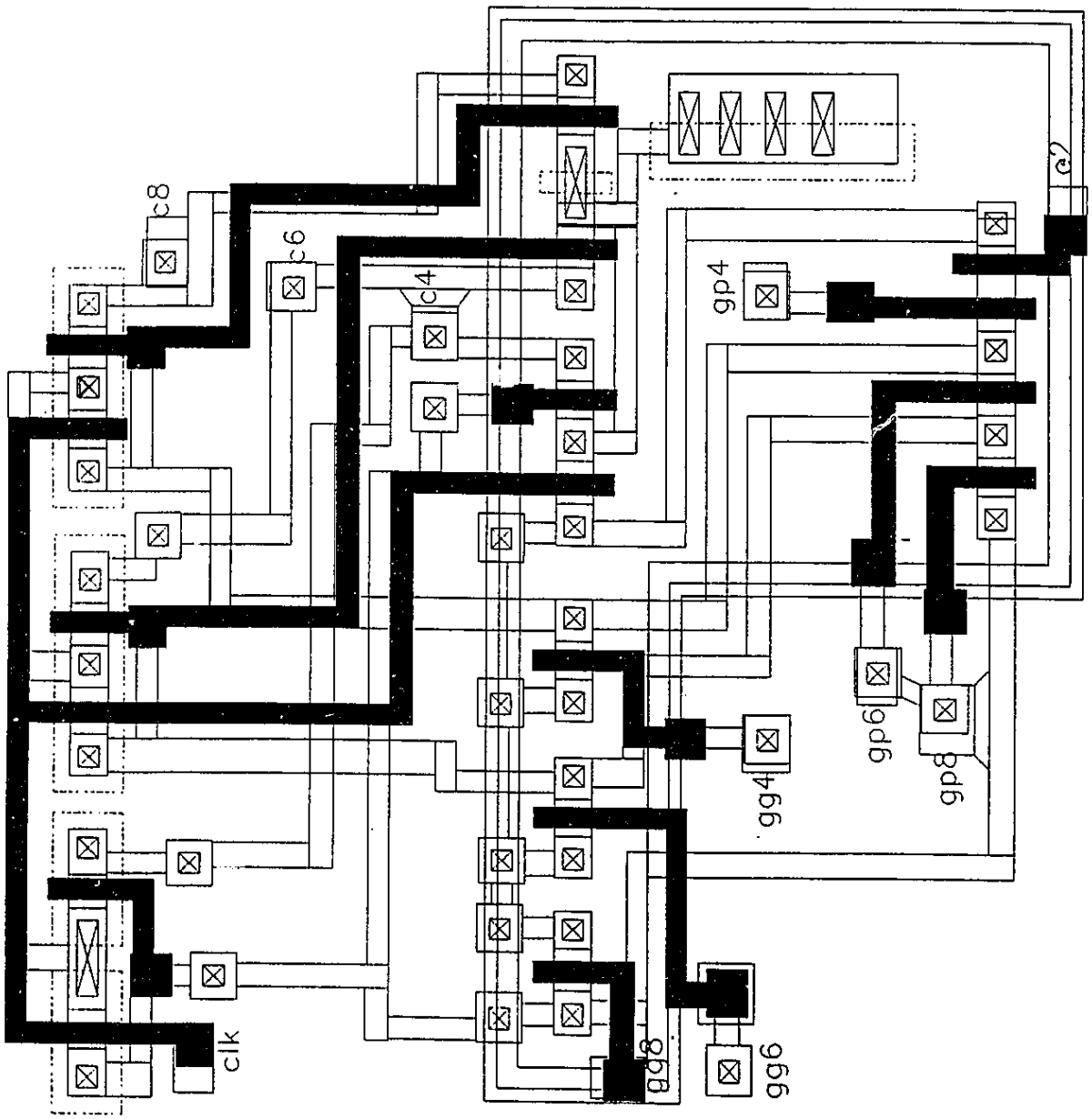


Figure E.13: Layout for Carry Generator Block

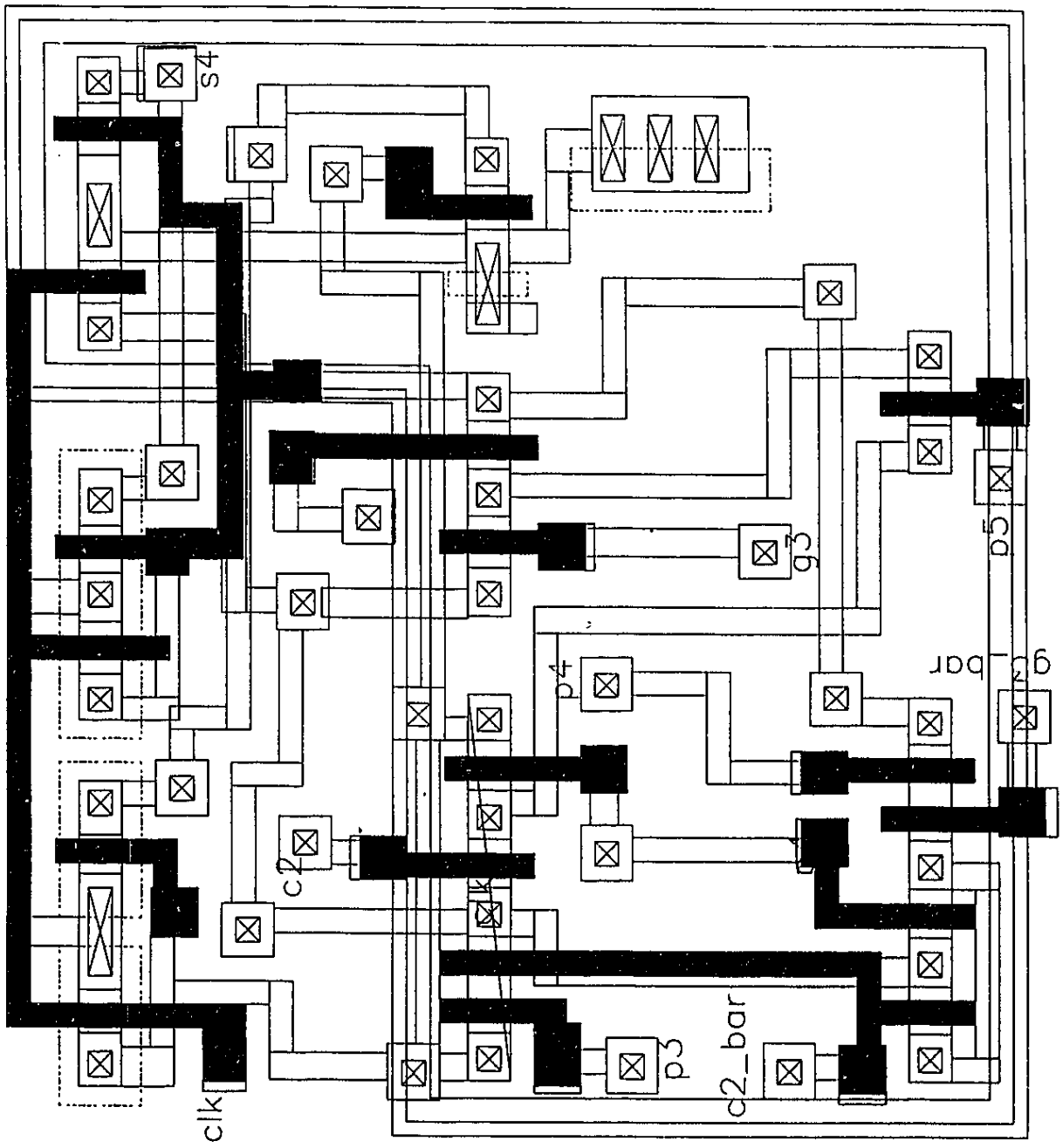


Figure E.14: Layout for Sum Block

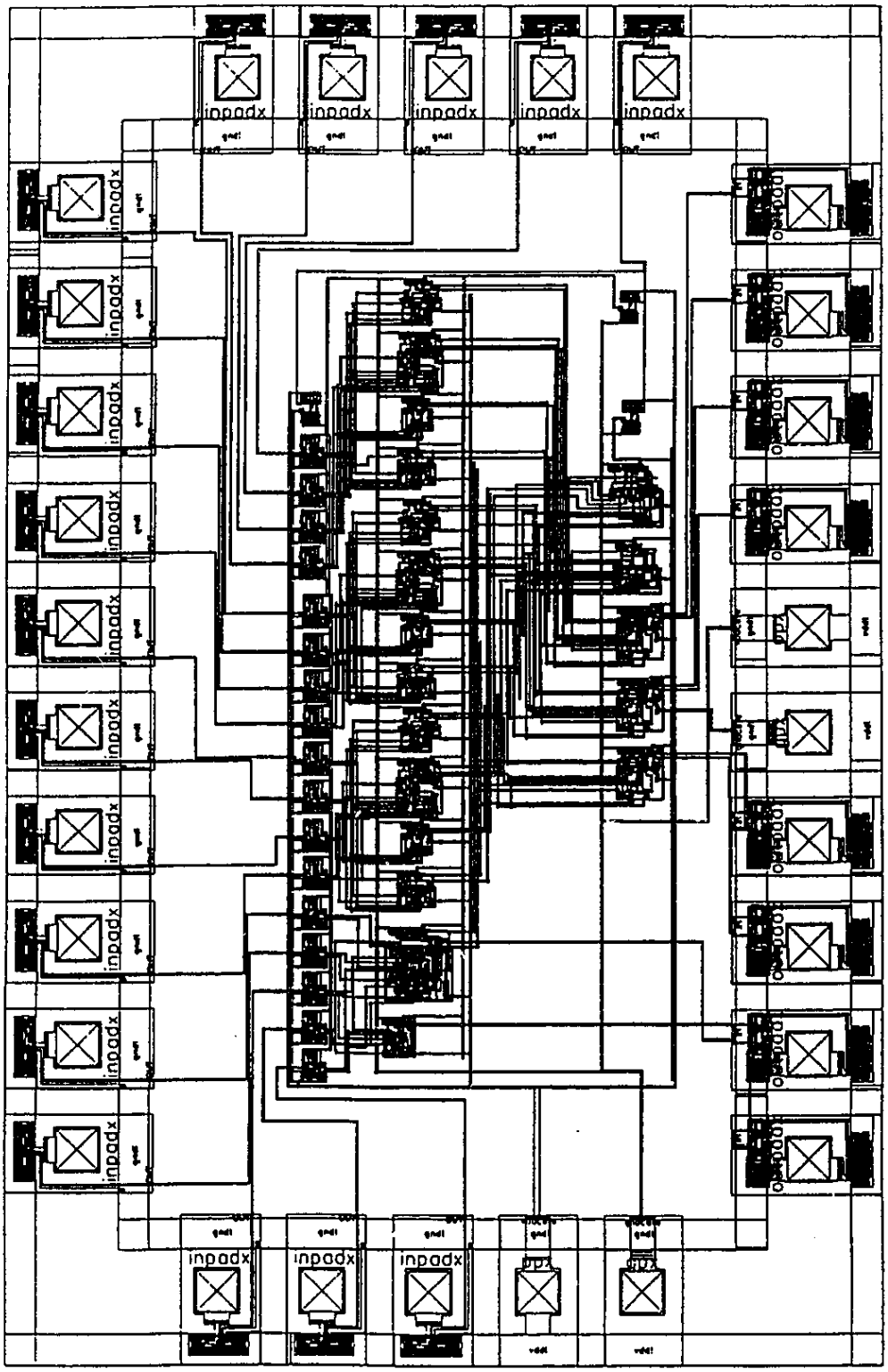


Figure E.15: Layout for the 10-Bit Adder

E.8 CONCLUSIONS

The 10 bit adder described in this section demonstrates extremely high speed and low silicon area requirement. The adder achieves 10 bit addition at a maximum frequency of 70Mhz. These achievements are due to the combination of MODL, clocking scheme and device sizing. MODL allows for the area and speed efficient designs.

REFERENCES

[E1] I. Hwang ,"Ultrafast Compact 32-bit CMOS Adders in Multiple Output Domino Logic," IEEE Journal of Solid State Circuits, Vol. 24, April 1989.

[E2] D. L. Dietmever, "Large Design of Digital Systems," Boston and Bacon, 1979, pp. 72-79.

[E3] J. Yuan and C. Svensson, "High Speed CMOS Circuit Technique," IEEE Journal of Solid State Circuits, Vol. 24, February 1989.

VITA AUCTORIS

Biniam Mesfin was born on the 19th of February, 1964 in Asmara Eritrea. He comes from a family of 9 children. He completed his high school education at Asmara comprehensive School in Eritrea. He joined the University of Windsor in 1985 where he received several academic scholarships during his undergraduate study. He graduated with a Bachelor of Applied Science degree in 1990. He finished his Master of Applied Science in Electrical Engineering at the University of Windsor, Canada, in June 1992.