

**AN FPGA BASED INDIAN ARITHMETIC CO-PROCESSOR**

**A PROJECT REPORT**

*Submitted in partial fulfillment of the requirement  
for the award of the degree of*

**BACHELOR OF ENGINEERING**  
in  
**COMPUTER SCIENCE AND ENGINEERING**

*By*

**EASWARAN. R**  
**KARTHIK. S**  
**LAKSHMI NARASIMHAN. C**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**  
**COLLEGE OF ENGINEERING, GUINDY**  
**ANNA UNIVERSITY**  
**CHENNAI-600025**  
MAY 2000

## **BONAFIDE CERTIFICATE**

Certified that this thesis titled "An FPGA based Indian arithmetic co-processor" is the bonafide work of

**EASWARAN. R**

**KARTHIK. S**

**LAKSHMI NARASIMHAN. C**

who carried out the work under my supervision.

Certified further, that to the best of my knowledge, the work reported herein, does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on these or other candidates.

**Dr. K.M. MEHATA**  
Director  
School of Computer Science & Engg.  
College of Engineering  
Anna University  
Guindy  
Chennai - 25

**Dr. RANJANI PARTHASARATHI**  
Asst. Professor  
School of Computer Science & Engg.  
College of Engineering  
Anna University  
Guindy  
Chennai - 25

## **ACKNOWLEDGEMENT**

We would like to express our deepest gratitude to our guide, Dr. Ranjani Parthasarathi, for creating a deep interest in the study of Indian mathematics and reconfigurable computing. Her constant guidance, motivation and support inspired us to complete this project. We would like to thank all our professors, for their support. We thank the Xilinx corporation, Santa Clara, USA, for providing us with the Xilinx FPGA boards and the Foundation series software. Finally, we would like to thank the non-teaching staff of the department, especially Mr. Kingsley and Mr. Ramesh Kumar, for helping us to use the laboratory facilities.

**KARTHIK S  
EASWARAN R  
LAKSHMI NARASIMHAN C**

## **ABSTRACT**

Ancient Indian algorithms have been found to be well suited for arbitrary precision computer arithmetic. This project aims at evaluating a hardware implementation of these algorithms. Five such algorithms - Straight Division, Dwandwa Square Root, Urdhva Tiryak Multiplication, Dwandwa Squaring, and Ekadhika Divisibility Testing - have been considered. Since these algorithms are based on the place-value system of numbers, they can be applied to any arbitrary radix. A high degree of parallelism is present in these algorithms, the exploitation of which is limited only by the available hardware resources. Further, the implementations share the same basic units, thereby reducing development time and complexity. Hence, an FPGA based reconfigurable architecture has been favored for implementation. The algorithms have been implemented on a 'Host + Reconfigurable Coprocessor' architecture. The results of the implementation show the efficiency of these algorithms. Applications involving very large numbers can benefit from this implementation.

## TABLE OF CONTENTS

	<i>Page No.</i>
Abstract (English)	i
List of figures	ii
List of tables	iii
List of Abbreviations	iv
Chapter 1	
Introduction	
1.1 Project overview	1
1.2 Thesis organization	2
Chapter 2	
VLPA and Indian algorithms	
2.1 Introduction	3
2.2 The algorithms	
2.2.1 Multiplication	4
2.2.2 Squaring	5
2.2.3 Division	6
2.2.4 Square root	7
2.2.5 Divisibility testing	9
2.3 Suitability of hardware implementation	10
Chapter 3	
The design of the co-processor	
3.1 Overview of the co-processor	11
3.2 The Multiplication unit	12
3.3 The Squaring unit	15
3.4 The Division unit	18
3.5 The Square root unit	23
3.6 The Divisibility testing unit	27

## Chapter 4

### Implementation details

4.1 Implementation overview	30
4.2 Implementation of the units	31
4.2.1 The Multiplication unit	32
4.2.2 The Squaring unit	33
4.2.3 The Division unit	35
4.2.4 The Square root unit	36
4.2.5 The Divisibility testing unit	38

## Chapter 5

### Conclusion

5.1 Summary	42
5.2 Suggested future work	43

## A. Appendix

A.1 Straight division	44
A.1.1 The Straight division algorithm	44
A.1.2 Straight division - examples	46
A.2 Square rooting	50
A.2.1 The square rooting algorithm	50
A.2.2 Square rooting - examples	53
A.3 Divisibility testing	56
A.3.1 Divisibility testing - example	56

References	58
------------	----

## LIST OF FIGURES

FIGURE NO.	TITLE	PAGE NO.
3.1	The Host + Reconfigurable architecture	12
3.2	Multiplication unit	13
3.3	State machine for multiplication	14
3.4	Cross product unit	15
3.5	Squaring unit	16
3.6	State machine for squaring	17
3.7	Dwandwa unit	17
3.8	Straight division unit	19
3.9	State machine for straight division	20
3.10	Straight division correction unit	22
3.11	State machine for straight division correction	23
3.12	Square root unit	24
3.13	State machine for square root	25
3.14	Square root correction unit	26
3.15	State machine for square root correction	27
3.16	Divisibility testing unit	28
3.17	Multiply and Add unit of divisibility testing	28
3.18	State machine for divisibility testing	29
4.1	Implementation overview	31
4.2	Timing simulation of the multiplication unit	33
4.3	Timing simulation of the squaring unit	34
4.4	Timing simulation of the division unit	36
4.5	Timing simulation of the square root unit	38
4.6	Timing simulation of the divisibility unit	40

## LIST OF TABLES

TABLE NO.	TITLE	PAGE NO.
4.1	Estimated performance of the multiplication unit	32
4.2	Estimated performance of the squaring unit	34
4.3	Estimated performance of the division unit	36
4.4	Estimated performance of the square root unit	37
4.5	Estimated performance of the divisibility unit	39
4.6	Time and space complexity of the units	41

## LIST OF ABBREVIATIONS

CLB	Configurable Logic Block
CP	Cross Product
CU	Correction Unit
DSR	Dwandwa Square Rooting
FPGA	Field Programmable Gate Array
HRAC	Host + Reconfigurable Architecture Computing
LA	Look Ahead
RNS	Redundant Number System
SDNS	Signed Digit Number System
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLP	Variable Long Precision
VLPA	Variable Long Precision Arithmetic

## CHAPTER 1 - INTRODUCTION

India has a rich mathematical heritage. Over the time, many methods for performing fast mental calculations have evolved, which are better suited than other common techniques. This is because of the fact that, in the Indian methods, only few numbers have to be remembered at any instant. This is analogous to performing calculations on large numbers with a limited memory on a computer. Some attempts have been made to implement these algorithms on computers [1-3]. These studies show that they are well suited for computer arithmetic. Many of these earlier works have used existing hardware and implemented these algorithms in software. They suggest that existing hardware pose restrictions in exploiting many of the features of these algorithms. A custom hardware can remove many of these restrictions and provide better performance. This project aims at exploring and evaluating such an implementation. To shorten development time and to further exploit certain features of these algorithms, an FPGA based reconfigurable architecture has been used.

### 1.1 Project Overview:

The Indian algorithms that have been considered for hardware implementation are:

1. Multiplication - Urdhva Tiryak technique [4]
2. Squaring - Dwandwa technique [4]
3. Straight Division - Urdhva Tiryak technique [1,4]
4. Square Root - Dwandwa technique [2,4]
5. Divisibility testing - Ekadhika technique [4]

These have been implemented on a 'Host + Reconfigurable coprocessor' architecture. The coprocessor is a Xilinx FPGA [5], which can be configured as custom hardware. The host configures the coprocessor with the configuration to implement the desired arithmetic algorithm. At any time, the coprocessor consists of only one of the above five algorithms. The host sends operands to the coprocessor and reads back the results of its computation.

The design of hardware for each of the above five algorithms has been done using VHDL. The designs have been tested by simulating them on Xilinx foundation series software and have been implemented on the Xilinx XC4000XL series FPGA [5]. The results of the implementation demonstrate the efficiency of these algorithms and their suitability for computers. Since these algorithms perform better for larger numbers, applications involving 'very large numbers' (e.g. cryptography) can benefit from these algorithms.

## **1.2 Thesis Organization**

This thesis is organized as follows:

Chapter 2 gives the basic algorithms for multiplication, squaring, division, square root and divisibility testing. These algorithms, with certain modifications to improve efficiency, are implemented in hardware.

Chapter 3 details the design of the co-processor. The detailed design of the various units implementing the algorithms mentioned in section 2 is given here.

Chapter 4 deals with the implementation details of the various units of the co-processor and gives the results of performance analysis.

Chapter 5 summarizes the project and suggests avenues for future work.

References and Appendices giving examples for Divisibility, Square Root and Division algorithms are given at the end of the thesis.

## CHAPTER 2 - VLPA AND INDIAN ALGORITHMS

### 2.1 Introduction

Floating point representation is the usual notation for representing large numbers. This representation reduces the space required to store large numbers but results in a loss of accuracy. For applications that require a high degree of accuracy, a fixed point representation with variable precision is preferred. That is, each processor word is treated as a high radix digit (e.g. : base 256 base 65536 etc.). The long precision operands are treated as arrays of processor words. Arithmetic on such a representation is called as Variable Long Precision Arithmetic (VLPA). VLPA algorithms for multiplication, division and square root have been proposed [1-3,6]. These algorithms satisfy the 'online' property. Online algorithms are those, where, after an initial delay of ' $\delta$ ' digits, for every digit of input operand, one result digit is obtained [7]. Hence, online algorithms have a constant order space complexity.

Ancient Indian algorithms used for mental arithmetic computation, have all these characteristics of VLP algorithms. These algorithms are in-place, i.e., no extra space other than the input VLP numbers is required. The fundamental operations of these algorithms are single digit by single digit multiplication, double digit by single digit division, and two digit addition. The space requirement is only for the storage of the intermediate results of these fundamental operations. Since these algorithms are based on the place value system of numbers, they are applicable to any radix. Moreover, these algorithms perform better as the radix becomes higher, and these algorithms are online. Software implementation of these algorithms has been found to be better than certain existing long precision algorithms [1,2].

This chapter formally presents five of these Indian algorithms viz. Multiplication, Squaring, Division, Square root and Divisibility, modified for computer implementation.

## 2.2 The Algorithms

### 2.2.1 Multiplication

The input to this algorithm is 2 numbers of arbitrary length and base. The output is their product.

The multiplication algorithm makes use of an operator called the cross-product (CP). The CP of an n digit number  $a_n a_{n-1} a_{n-2} \dots a_1$  with another n digit number  $b_n b_{n-1} b_{n-2} \dots b_1$  is defined as :

$$CP(a_n a_{n-1} a_{n-2} \dots a_1, b_n b_{n-1} b_{n-2} \dots b_1) = a_1 * b_n + a_2 * b_{n-1} + \dots + a_{n-1} * b_2 + a_n * b_1$$

The result when an n digit number is multiplied by an m digit number ( $n > m$ ), where x is the base is given by:

$$(a_n a_{n-1} a_{n-2} \dots a_1 * b_m b_{m-1} b_{m-2} \dots b_1) = x^0 * CP(b_1, a_1) + x^1 * CP(b_2 b_1, a_2 a_1) + \dots + \\ x^{m-1} * CP(b_m \dots b_1, a_m \dots a_1) + \dots + \\ x^{n-1} * CP(b_m \dots b_1, a_n \dots a_{n-m+1}) + \\ x^n * CP(b_m \dots b_2, a_m \dots a_2) + \dots + x^{n+m-2} * CP(b_m, a_m)$$

It may be noted that this is an online algorithm. Digits of the product can be generated even when the multiplicand and the multiplier are not fully available. An example illustrating this algorithm is given below.

$$\begin{aligned} \text{Example: } 4567 * 123 &= 10^0 * CP(3,7) + 10^1 * CP(23,67) + 10^2 * CP(123,567) + \\ &10^3 * CP(123,456) + 10^4 * CP(12,45) + 10^5 * CP(1,4) \\ &= 1*21 + 10*32 + 100*34 + 1000*28 + 10000*13 + 100000*4 \\ &= 561741 \end{aligned}$$

## 2.2.2 Squaring

The input to this algorithm is a number of arbitrary length and base. The output is the square of the number. Although squaring is a special case of multiplication, it is advantageous to use a different algorithm for squaring as it eliminates redundant multiplications.

The squaring algorithm makes use of an operator called Dwandwa. The Dwandwa of any number  $a_n a_{n-1} a_{n-2} \dots a_1$  is defined as:

$$D(a_n a_{n-1} a_{n-2} \dots a_1) = 2 * a_1 * a_n + 2 * a_2 * a_{n-1} + \dots + 2 * a_{n/2} * a_{(n/2)+1} \quad \text{when } n \text{ is even.}$$

$$D(a_n a_{n-1} a_{n-2} \dots a_1) = 2 * a_1 * a_n + 2 * a_2 * a_{n-1} + \dots + 2 * a_{(n-1)/2} * a_{(n+3)/2} + a_{(n+1)/2} * a_{(n+1)/2} \quad \text{when } n \text{ is odd.}$$

If  $x$  is the base in which digits are represented, the square of a number  $a_n a_{n-1} a_{n-2} \dots a_1$  is

$$\begin{aligned} (a_n a_{n-1} a_{n-2} \dots a_1)^2 = & x^0 * D(a_1) + x^1 * D(a_2 a_1) + \dots + x^{n-1} * D(a_n a_{n-1} a_{n-2} \dots a_1) + \\ & x^n * D(a_n a_{n-1} a_{n-2} \dots a_2) + x^{n+1} * D(a_n a_{n-1} a_{n-2} \dots a_3) + \dots + \\ & x^{2n-3} * D(a_n a_{n-1}) + x^{2n-2} * D(a_n). \end{aligned}$$

(e.g.)

$$\begin{aligned} (234)^2 &= 1 * D(4) + 10 * D(34) + 100 * D(234) + 1000 * D(23) + 10000 * D(2) \\ &= 1 * 16 + 10 * 24 + 100 * 25 + 1000 * 12 + 10000 * 4 \\ &= 54756 \end{aligned}$$

This is also an online algorithm and the squaring operation can be initiated even when all the digits of the input are not available.

### 2.2.3 Division

The inputs to this algorithm are two numbers (the divisor A and the dividend B) of arbitrary length and base. The output is the arbitrary precision quotient of A/B

Let the base of the dividend and the divisor be x. Let the divisor be  $A = a_n a_{n-1} a_{n-2} \dots a_1$ , Let the dividend be  $B = b_m b_{m-1} b_{m-2} \dots b_1$ . The division algorithm involves the division of the partial dividend  $D_i$  by the most significant digit of the divisor,  $a_n$ .

In the first step,  $D_m = b_m$ ,  $S_m = 0$

For  $i < m$ , the partial dividend  $D_i$  and the quotient  $q_i$  in step i are given by the following.

$$D_i = b_i + S_{i+1}x - \sum_{j=1}^k q_{i+j} a_{n-j} \quad (1)$$

Where  $k = \min(n, m-i)$

$$S_i = D_i - q_i a_n. \quad (2)$$

Equations (1) and (2) represent the basic steps of the division algorithm. The division of the partial dividend should yield a quotient between 0 and x-1. Since the algorithm involves the estimation of quotient digits using only the first digit of the divisor, there is likely to be an error. The error is indicated by  $D_i$  becoming negative, in which case, it is corrected as follows :

- (i) Reduce the previous quotient digit  $q_{i+1}$  by 1. If the previous quotient digit(s) are zero, and the first non-zero quotient digit is  $q_{i+j}$ , the borrow is propagated.
- (ii)  $D_i$  is set to  $D_i + a_n x + a_{n-1} + \dots + a_{n-j}$

Steps (i) and (ii) are repeated till  $D_i$  becomes positive.

**Normalization:** It is noted that small values of  $a_n$  leads to large number of corrections as there is a larger probability of overestimating the quotient digits. Under such circumstances, the divisor and the dividend are multiplied by a constant such that the first digit of the divisor becomes greater than or equal to  $x/2$ .

**Use of Look ahead:** At any stage, corrections are generated when the cross product is subtracted from the gross dividend ( $b_i + S_{i+1}x$ ) and  $D_i$  becomes negative. This is caused by too high an estimation of  $q_{i+1}$  and hence too low  $S_i$ . All the quotients needed for this cross product (except one) would have already been available in the previous iteration. This information can be used to get a better estimate of the quotient. This is done by calculating a term called as the look ahead, which is given by  $LA = \sum_{j=1}^L q_{i+j}a_{n-j-1}$  where

$L = \min(n-1, m-1)$ . The revised algorithm is presented as:-

$$D'_i = b_i + S_{i+1}x - \sum_{j=1}^k q_{i+j}a_{n-j} - \text{int}\left(\left[\sum_{j=1}^L q_{i+j}a_{n-j-1}\right]\right) / x \quad (3)$$

Where  $K = \min(n, m-i)$  and  $L = \min(n-1, m-i)$

The quotient is estimated as:

$$q_i = D'_i / a_n \text{ and } S_i = D_i - q_i a_n \text{ when } (0 \leq D'_i < a_n x)$$

$$q_i = x-1 \text{ and } D_i - q_i a_n \text{ when } (D'_i \geq a_n x)$$

else correction.

**Use of SDNS:** Signed Digit Number System (SDNS) is a redundant number system where each digit can take values from  $-(x-1)$  to  $(x-1)$ . When SDNS is used, each digit has its own sign. This gives the flexibility to generate negative quotients for negative partial dividends minimizing corrections. Examples illustrating the modifications are present in the appendix.

#### 2.2.4 Square root

The input to this algorithm is an SDNS number of arbitrary length and base. The output is its square root, calculated to the desired precision.

Let the number whose square root is to be determined be represented as  $b_n x^n + b_{n-1} x^{n-1} + \dots + b_m x^{-m}$ , where  $x$  is the base and the  $b_i$ s are the coefficients. Since the number is in SDNS,  $b_i$ s can be negative. The number of digits of the number,  $n+1$ , is even. The square root of the number may be denoted as  $a_{n-1} x^p + a_{n-2} x^{p-1} + \dots + a_{n-p-1} + a_{n-p} x^{-1} + \dots + a_{n-p-q-1} x^{-q}$

where  $p = (n-1)/2$  and  $q$  is the desired precision. Using  $D_i$  to denote the partial radicand,  $s_i$  to denote the remainder at each step, and defining  $2a_{n-1}$  as  $T$ , the first two steps involved in the calculation are:

*Step 1:*

$$D_{n-1} = b_n x + b_{n-1},$$

$$a_{n-1} = \text{int}(\sqrt{D_{n-1}}),$$

$$s_{n-1} = D_{n-1} - a_{n-1}^2,$$

*Step 2:*

$$D_{n-2} = s_{n-1} x + b_{n-2},$$

$$a_{n-2} = \text{int}[D_{n-2}/T],$$

$$s_{n-2} = D_{n-2} - a_{n-2} * T$$

From the third step, the Dwandwa of  $(a_{n-2}, a_{n-1}, \dots, a_{i+1})$ , where  $a_{i+1}$  is the last obtained digit of square root, has to be subtracted to obtain the partial radicand  $D_i$ . The Look ahead (LA) is calculated as Dwandwa of  $(a_{n-1}, \dots, a_{i+1})$ . From  $D_i$ ,  $LA/x$  is subtracted to get  $D_i'$ . If the magnitude of  $D_i'$  exceeds base times  $T$ , a correction has to be applied. If there is no correction,  $D_i'$  is divided by  $T$  to get the next quotient digit  $a_i$ .

This can be stated as follows:

for  $(i = n-3)$  to  $(i = n-p-q-1)$

$$D_i = s_{i+1} x + b_i - \text{Dwandwa}(a_{n-2}, \dots, a_{i+1})$$

$$LA = \text{Dwandwa}(a_{n-1}, \dots, a_{i+1})$$

$$D_i' = D_i - \text{int}[LA/x]$$

if  $\{ -T*(x-1) < D_i' < T*(x-1) \}$

$$a_i = \text{int}[D_i'/T]$$

$$s_i = D_i - T * a_i$$

else correction until  $-T*(x-1) < D_i' < T*(x-1)$

correction

$$a_{i+j} = a_{i+j} \pm 1$$

$$D_i = D_i \bar{+} (T^*x + 2a_{n-2} + \dots + 2a_{n-2-j})$$

$$\text{if } (2*j = n-i)$$

$$D_i = D_i \pm 1$$

$$LA = LA \bar{+} 2*((x-1)*(a_{n-1} + a_{n-2} + \dots + a_{n-2-j+1}) - a_{n-2-j})$$

$$\text{If } (2*j = n-i+1)$$

$$LA = LA \pm 1$$

$$D_i' = D_i - LA/x;$$

In places where  $\pm$  is used, the type of operation depends on the correction type. If  $D_i \geq T^*x$ , then the operation is addition, otherwise it is subtraction. The operator  $\bar{+}$  is used to indicate the inverse of  $\pm$  i.e., when  $D_i \geq T^*x$ , then the operation  $\bar{+}$  is subtraction otherwise it is addition.

Examples illustrating this algorithm are given in the appendix.

### 2.2.5 Divisibility testing

The inputs to this algorithm are 2 numbers of arbitrary length and base-the dividend and the divisor. The divisibility of the dividend by the divisor is checked and reported without carrying out the actual process of division.

This algorithm makes use of an operator called Ekadhika. Ekadhika of a number N of base x,  $E(N)_x$  is calculated by multiplying N with a suitable number such that the last digit is reduced to x-1. Then 1 is added to the result and the trailing zero is dropped.

For example to determine  $E(17)_{10}$

Multiply 17 with 7 to get 10-1=9 in the last digit:  $17*7 = 119$

Add 1:  $119+1= 120$

Drop the trailing Zero of 120 to get 12.

The actual divisibility-testing algorithm is given below.

Let  $N = a_n a_{n-1} a_{n-2} \dots a_1$  be the dividend.

Let  $M = b_m b_{m-1} b_{m-2} \dots b_1$  be the divisor.

Calculate the Ekadhika of the divisor as  $E(M) = e_m e_{m-1} e_{m-2} \dots e_1$ .

*Step 1:* Assign  $P=N$

*Step 2:* Multiply the Ekadhika with the last digit of P. Add the resultant number to P after dropping the last digit of P.(i.e. ) In the first iteration, multiply  $e_m e_{m-1} e_{m-2} \dots e_1$  by  $a_1$  and add it to  $a_n a_{n-1} a_{n-2} \dots a_2$ .

*Step 3:* Store the result in P.

Note: The number of digits in P will decrease by 1 or remain the same depending on whether there is a carry that propagates beyond the first digit. On the whole, repeated application of steps 2 and 3 ensure that the digits of P monotonically decrease. This process of multiply-and-add is called osculation.

*Step 4:* Repeat steps 2 and 3 till the number of digits of P reduce to  $(m+1)$ . At this point if P is divisible by M, the number N is divisible by M.

What this algorithm does, is to reduce the problem to checking the divisibility of an  $n+1$  digit number by an  $n$  digit number. Standard techniques like Knuth's guessing technique[8] exist to check  $n+1$  by  $n$  digit divisibility through a single vector multiplication.

### **2.3 Suitability of hardware implementation**

As seen above, the efficiency of these algorithms improves with modifications like the use of Signed Digit Number System (SDNS). If the underlying hardware were to support such representations, these algorithms could perform even better [1,2]. Moreover, there is a high degree of parallelism in these algorithms that can be exploited. So, a hardware implementation that exploits these properties can be explored.

## CHAPTER 3 - THE DESIGN OF THE CO-PROCESSOR

A reconfigurable architecture has been considered for the implementation of the co-processor. This is based on the following reasons:

- These algorithms share the same basic units. So, the development time and complexity is reduced.
- Since these algorithms operate on VLP numbers, they typically take many cycles of execution. In comparison to this time, the reconfiguration overhead is minimal.
- Certain portions of the hardware for some of these algorithms are rarely used, that too on a mutually exclusive basis. Configuration of the required portion alone saves on hardware resources.
- Further, the similarity between the various designs can be exploited by the use of partial reconfiguration features.

The detailed design of the coprocessor is presented below.

### 3.1 Overview of the co-processor

The co-processor comprises of five major units:

1. The Multiplication unit
2. The Squaring unit
3. The Division unit
4. The Square Root unit
5. The Divisibility Testing unit

The above units are the hardware implementations of the Indian arithmetic algorithms of the corresponding arithmetic operations (the Multiplication unit implements the Urdhva Tiryak method, the Squaring unit implements the Dwandwa method; the Division unit implements the Urdhva Tiryak method;). The co-processor has been designed for implementation on an XS40 board, which has a Xilinx FPGA, RAM, micro controller and a parallel port interface.

The co-processor and its interface with the host is as shown in fig 3.1.

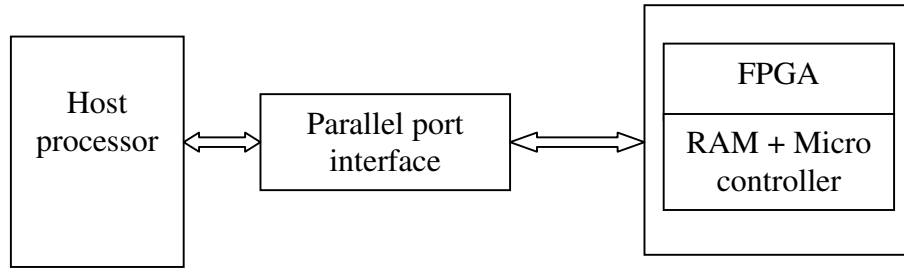


Fig. 3.1 The Host + Reconfigurable architecture

At any single moment, the co-processor has only one of the above five units. Based on the operation required, the host processor decides on what unit goes into the Xilinx FPGA. It configures the desired unit on the FPGA through the parallel port interface.

The host performs some preprocessing of the operands like normalization, conversion into SDNS etc. This processed data from the host processor is brought into the memory inside the FPGA, through the parallel port. The units get their operands from this memory. The units operate on the data and write the outputs back to the memory. These outputs are brought back to the host, again through the parallel port, where some post processing (like converting back into normal representation) is done by the host.

### 3.2 The Multiplication unit

The multiplication unit, as shown in fig. 3.2, operates on 8-bit, unsigned, base-256 numbers. The unit implements the Urdhva Tiryak method of multiplication. The detailed structure of the unit is given below in fig.3.2.

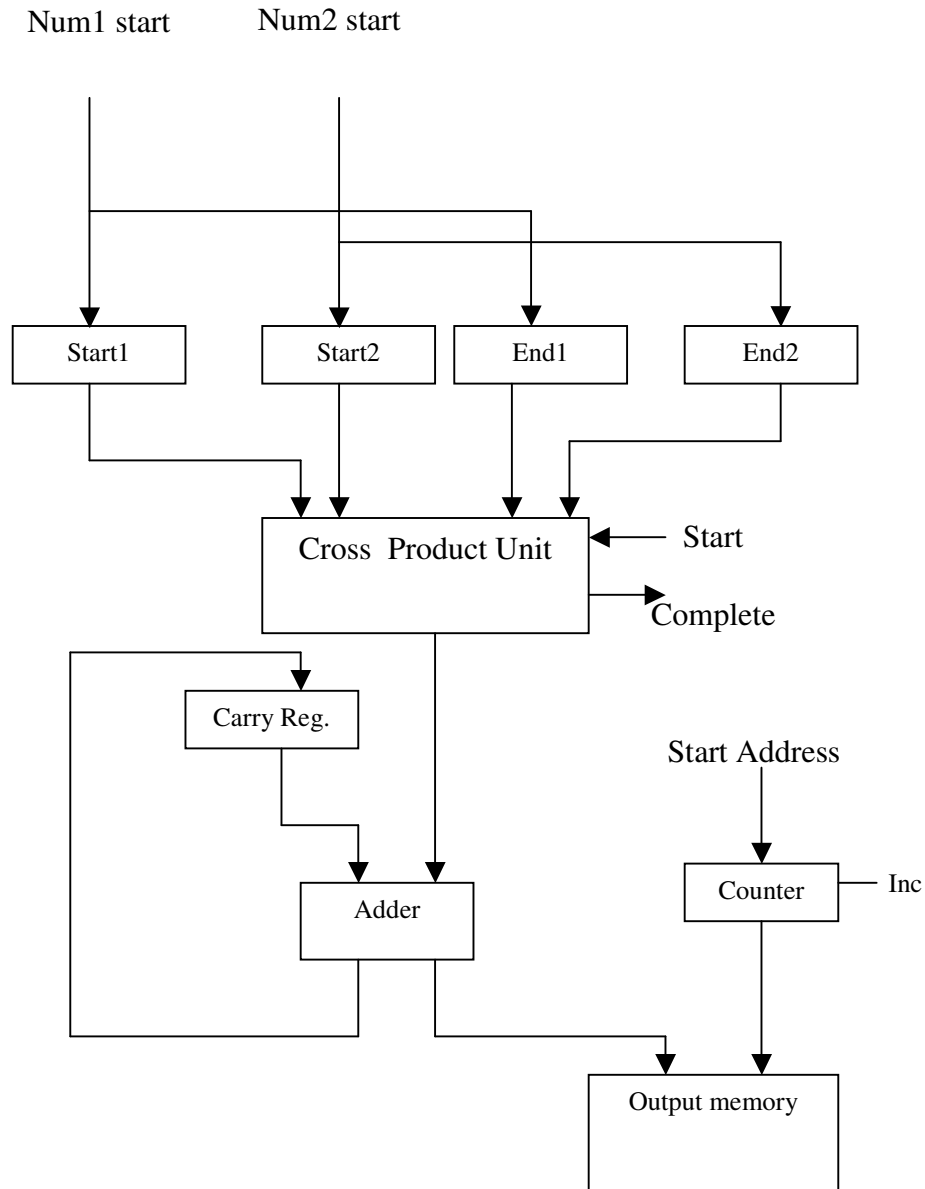


Fig. 3.2 Multiplication unit

'Num1' and 'Num2' are the multiplicand and multiplier respectively. Given proper start and end pointers, the 'Cross Product' unit, shown in fig. 3.4, computes the cross product of one set of given digits, with another set. Each time, after the cross product is over, the result is added with the previous carry to produce the product digit. This is the function of the carry register and the associated adder shown. After each cross product computation, the start and end pointers have to be changed according to the Urdhva Tiryak method.

This pointer manipulation is shown in the following state diagram, which is the control for the multiplication unit. Apart from generating increment/decrement signals for the pointers, the control unit also generates controls to load the carry register, start the cross product etc. In the states S3 and S4, the size of the vectors used in cross product calculation increases in every iteration. In S5 and S6, the vector size is constant. In S7 and S8, the vector size decreases. When it reaches 0, the state machine goes to S9, where the final carry is added to get the cross product.

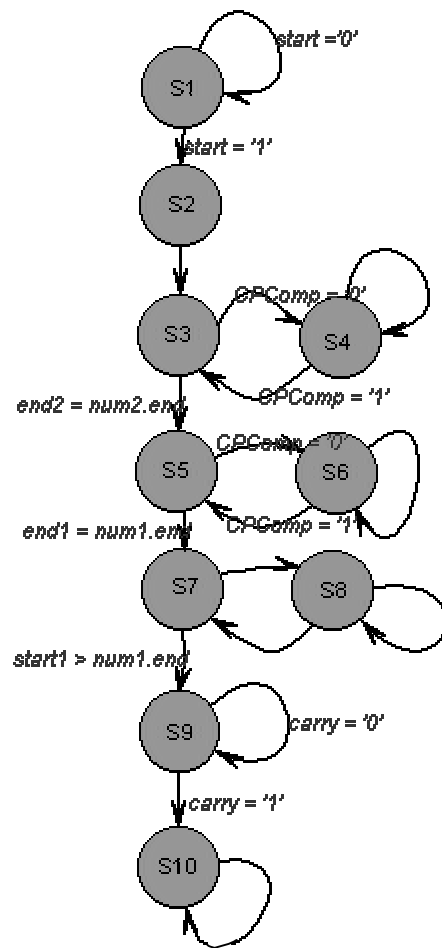


Fig 3.3 State machine for multiplication

The detailed design of the 'Cross Product' unit is shown in fig. 3.4. Here, in each iteration, two single-digit by single-digit products are computed (there are two multipliers and the memory is dual port). After every iteration, the 'start' pointer is incremented, the 'end '

pointer is decremented, and the product is added up (accumulated) in the cross product accumulator.

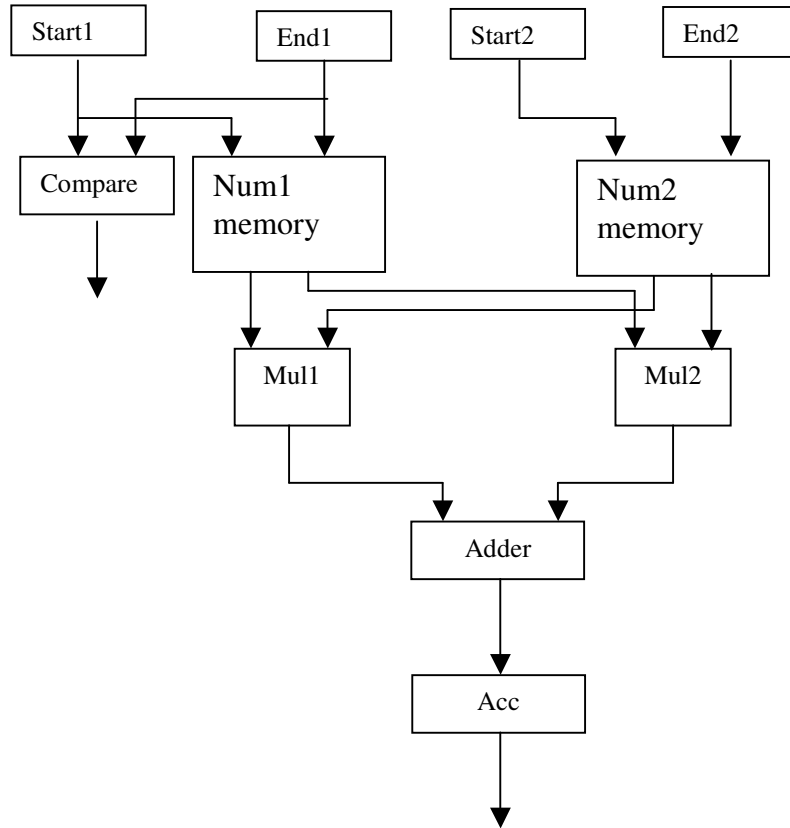


Fig. 3.4 Cross Product Unit

### 3.3 The Squaring unit

The squaring unit implements the Dwandwa method of squaring for 8-bit, unsigned, base-256 numbers. The detailed structure of the unit is given in fig. 3.5

This unit is very similar to the multiplication unit. Instead of the cross product unit, the dwandwa unit is used here. 'Num' is the number to be squared. After each Dwandwa computation, the previous carry is added to it producing one digit of the dwandwa. The detailed structure of the unit is given below in fig. 3.5.

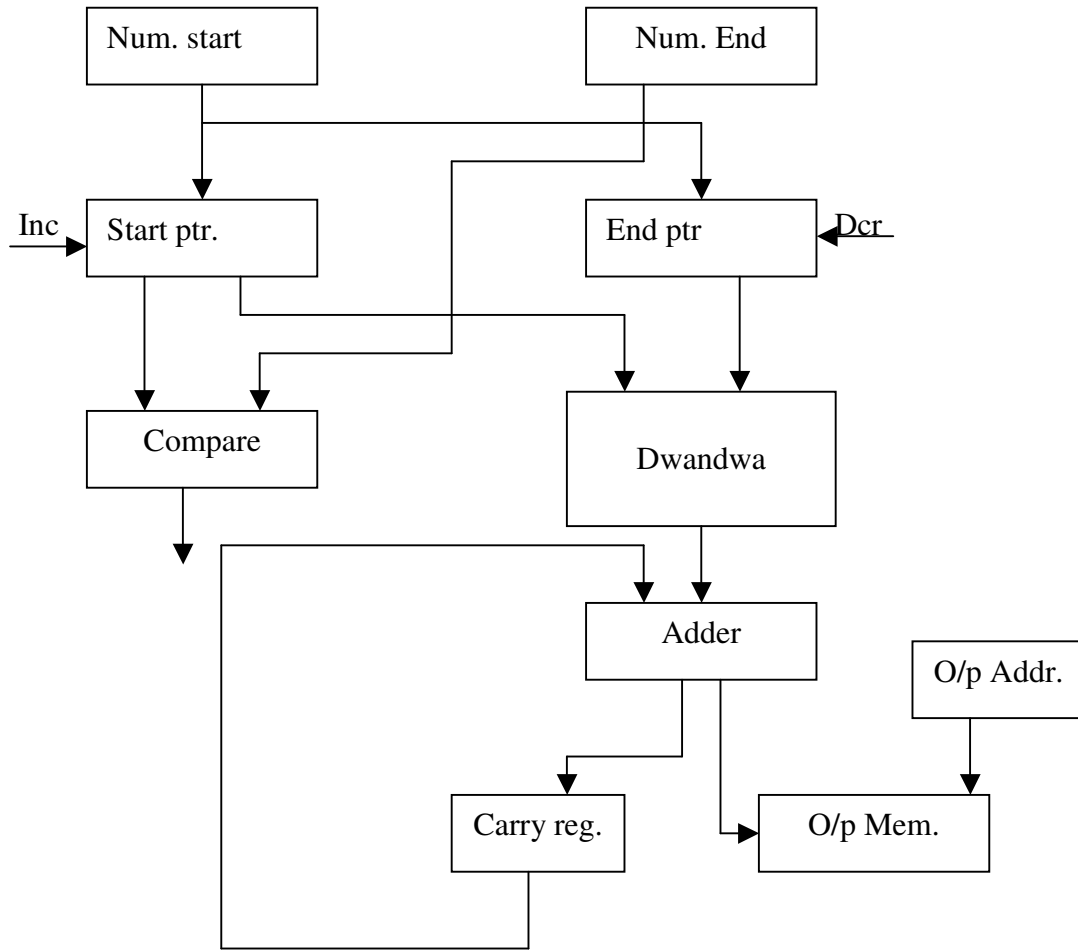


Fig. 3.5 Squaring unit

The state diagram of the control unit, which manipulates pointers and generates other control signals, is given in fig. 3.6. The Dwandwa calculation is performed in the states S4 and S6. At S9, the last carry digit is transferred to result memory.

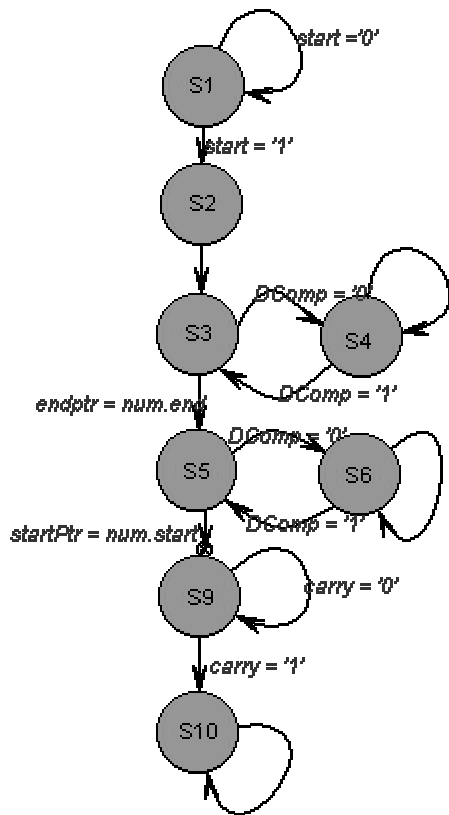


Fig. 3.6 State machine for squaring

The Dwandwa unit operates in a way very similar to the Cross Product unit of multiplication. The structure of the unit is given in fig. 3.7

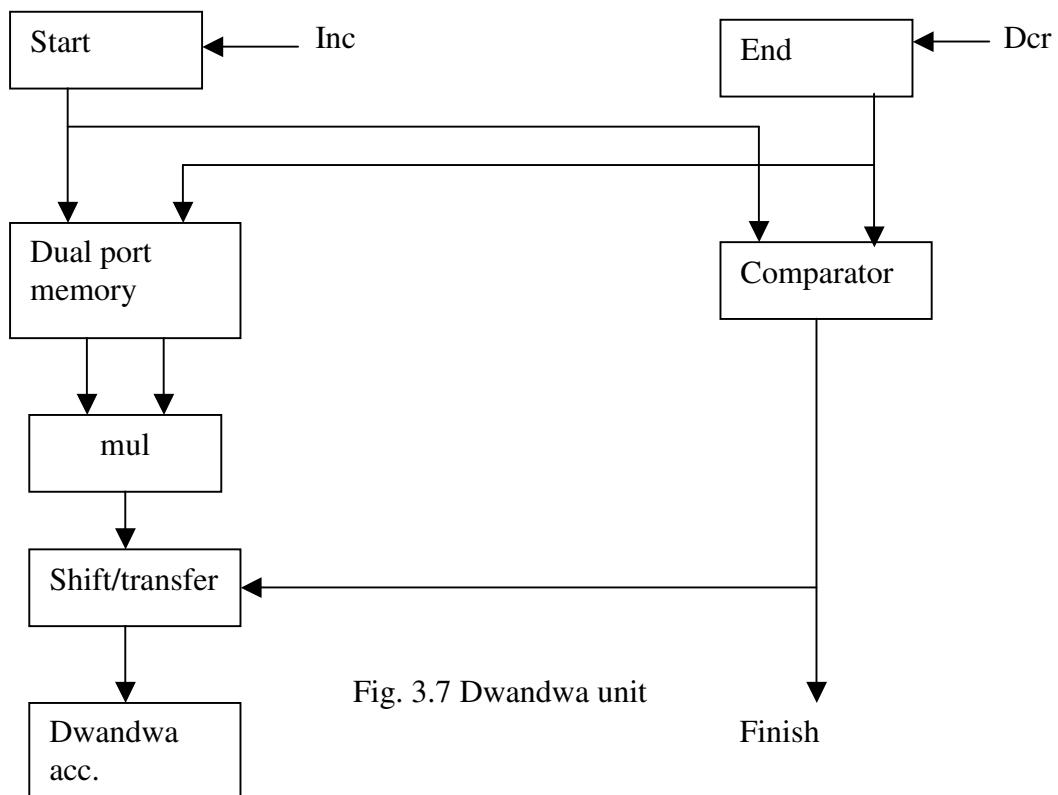


Fig. 3.7 Dwandwa unit

Note that only one multiplier is used and that multiplication by two is effected by shifting the product by one bit.

### **3.4 The Division unit**

The division unit, shown in fig. 3.8, performs VLPA division of two numbers that are in SDNS form. Each digit of the operands is stored as a 2's complement number. The division unit has three banks of memory to store the dividend, divisor and the quotient digits. The divisor and the quotients are stored in a dual port memory. This is to ensure that two digits of the divisor and the quotient are obtained every cycle during CP calculation. During one CP iteration, the addressing of the divisor memory and the quotient memory is done by the Divisor Counter and the Quotient Counter respectively. The parallelism in the cross product operation is exploited by having two multipliers. The look ahead is stored in the look-ahead accumulator. The partial dividend is obtained using the previous remainder, the next dividend digit and the cross product. The look-ahead component is subtracted from the partial dividend and stored in the MPDR. The correction determiner checks the value of MPDR to detect the presence of correction. If there is no correction, the content of MPDR is divided by  $a_0$  to obtain the quotient and the remainder. For this purpose, a combinational non-restoring divider[9] is used. Since this divider performs unsigned division, pre-complementing and post-complementing is necessary. The quotient is stored in the quotient memory. The remainder is added with the LA component and passed to the next iteration.



The control of the various registers, counters and the accumulators is done by a control unit. This unit is implemented as a state machine (fig. 3.9). This unit is activated by an external start signal and terminates when the quotient obtained is of the required precision. State S5 is the state where the cross product is computed. At S3, the quotient is obtained and it is written to memory. At S7, corrections are handled.

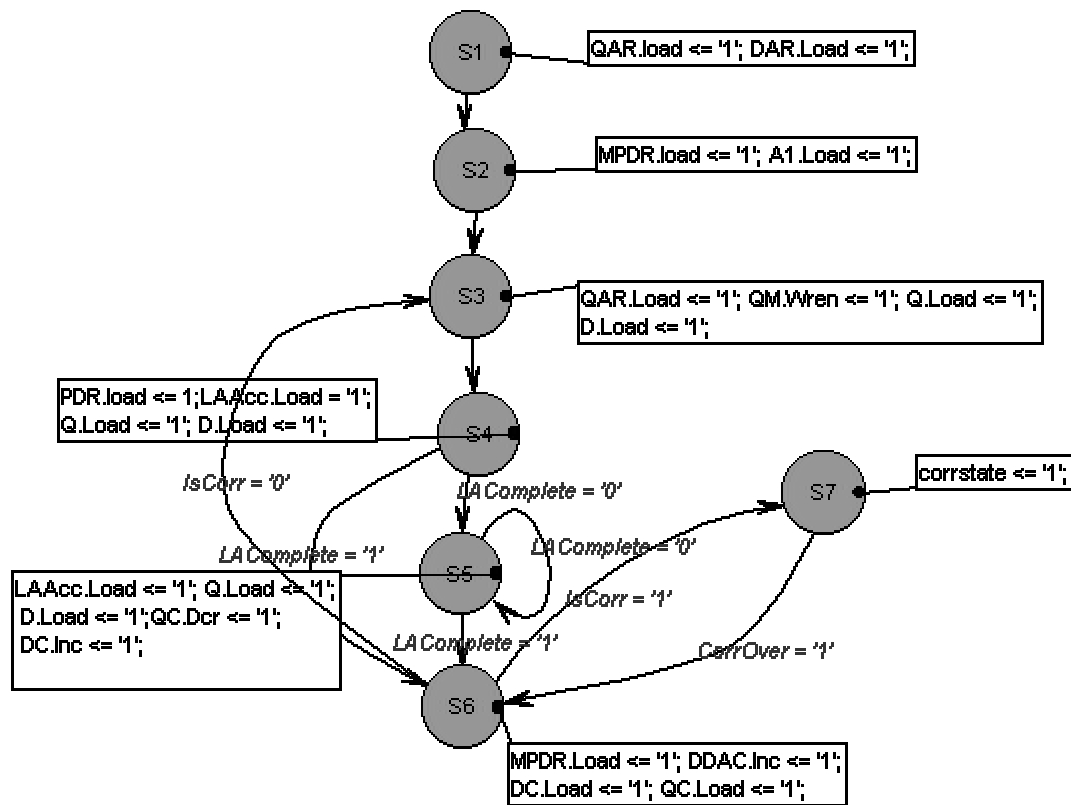


Fig. 3.9 State machine for Straight division

If the correction determiner detects a correction, the control is transferred to the Correction Unit (CU) (fig. 3.10). There can be two types of corrections. The last obtained quotient digit may either be decreased by one or increased by one due to correction. The type of correction is also identified by the correction determiner and this information is given to the CU. The CU suitably modifies the values of Quotient Memory, PDR and the LA. The CU performs corrections, till the correction determiner signals the absence of correction.



The components of the CU are controlled by a CU controller (fig. 3.11). This controller is also modelled as a state machine. The start signal to this controller is given by the Correction Determiner. Corrections are done at states S2 and S3. If the correction propagates, it is handles by repeatedly going through states S2 and S3. After performing the correction(s), this controller gives out a complete signal.

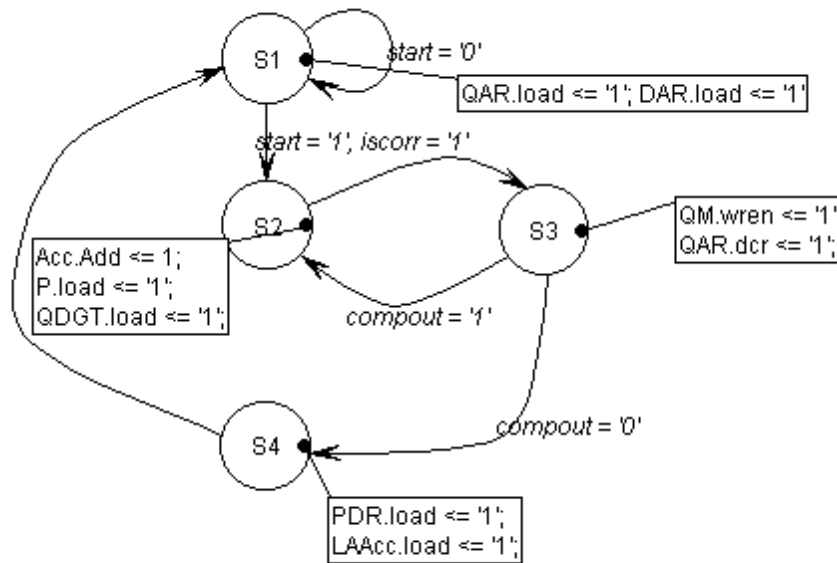


Fig. 3.11 State machine for straight division correction

### 3.5 The Square root unit

The square root unit, as shown in fig. 3.12, calculates the square root of a variable precision number in SDNS form. The number and the root are stored in two different memory banks. To calculate the Dwandwa, two digits of the root are needed in a single cycle. So a dual port memory is used to store the root. The initial root  $a_0$  and the initial remainder obtained by dividing the first two digits of the number with  $2a_0$  are given to the unit. The Dwandwa unit of squaring is modified to calculate the LA along with the Dwandwa. This modified unit, when given the start pointer and the end pointer,

calculates the corresponding Dwandwa and the Look Ahead. Using the Dwandwa, the partial product is calculated and stored in the PDR. From this, the LA component is subtracted and the result is stored in the MPDR. As in division, the correction determiner is used to detect the correction and transfer the control to the CU. In the absence of correction, the value of MPDR is divided by  $2a_0$  to get the next digit of the square root. The remainder is added with the LA component and given to the next iteration of the square root.

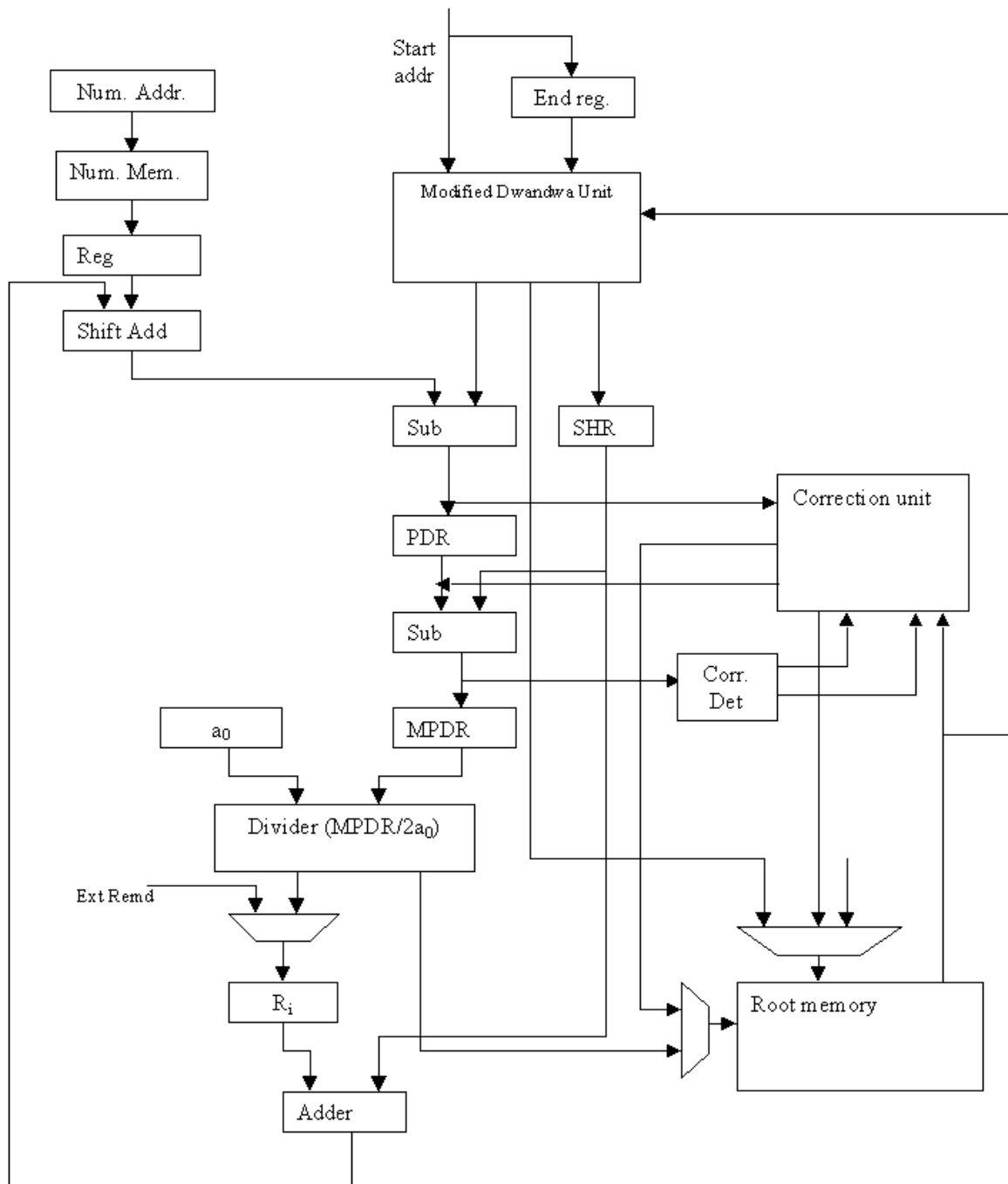


Fig. 3.12 Square root unit

The control circuitry for the square root unit is specified as a state machine, given in fig. 3.12. This is activated by an external start signal and terminates when the square root digits of required precision are obtained. S6 is the Dwandwa calculation state. At S5, the square root digit is written into the memory. If a correction is detected, control is transferred to state S8.

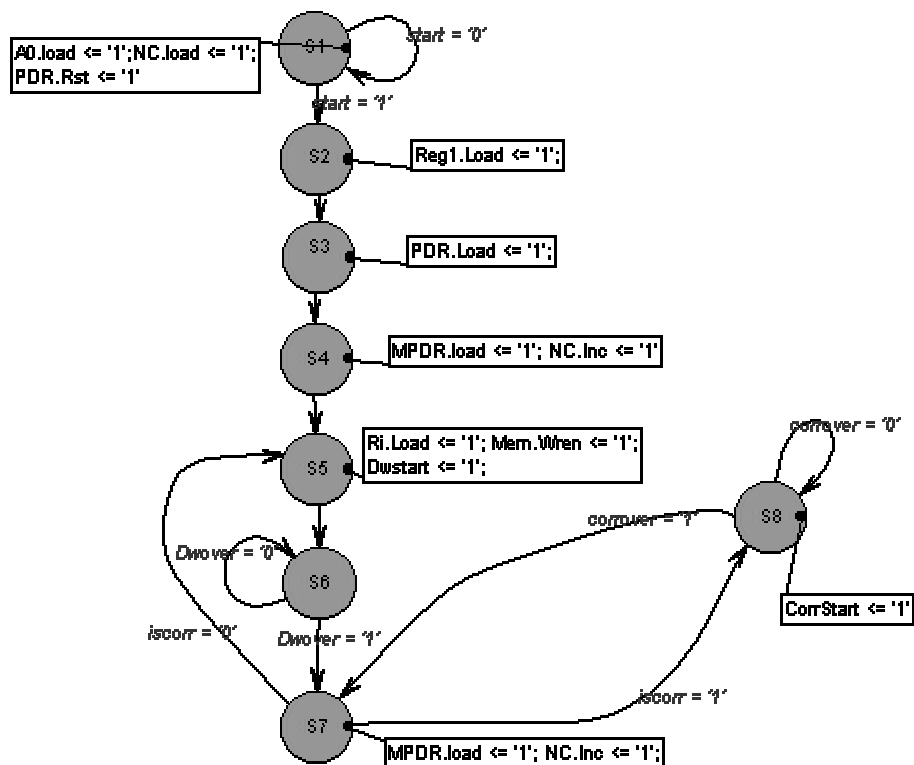


Fig. 3.13 State machine for square root

As in straight division, the CU of square root, shown in fig. 3.14 unit is started by the correction determiner and terminates when there are no more corrections. The CU modifies the Dwandwa, LA and the square root digits appropriately. This unit is also controlled by a state machine given in fig. 3.15. States S3 and S4, writes the corrected value of the root into the memory, and calculates the change in the PDR and LA. At S5 and S6, the new values of PDR and LA are written into the registers.

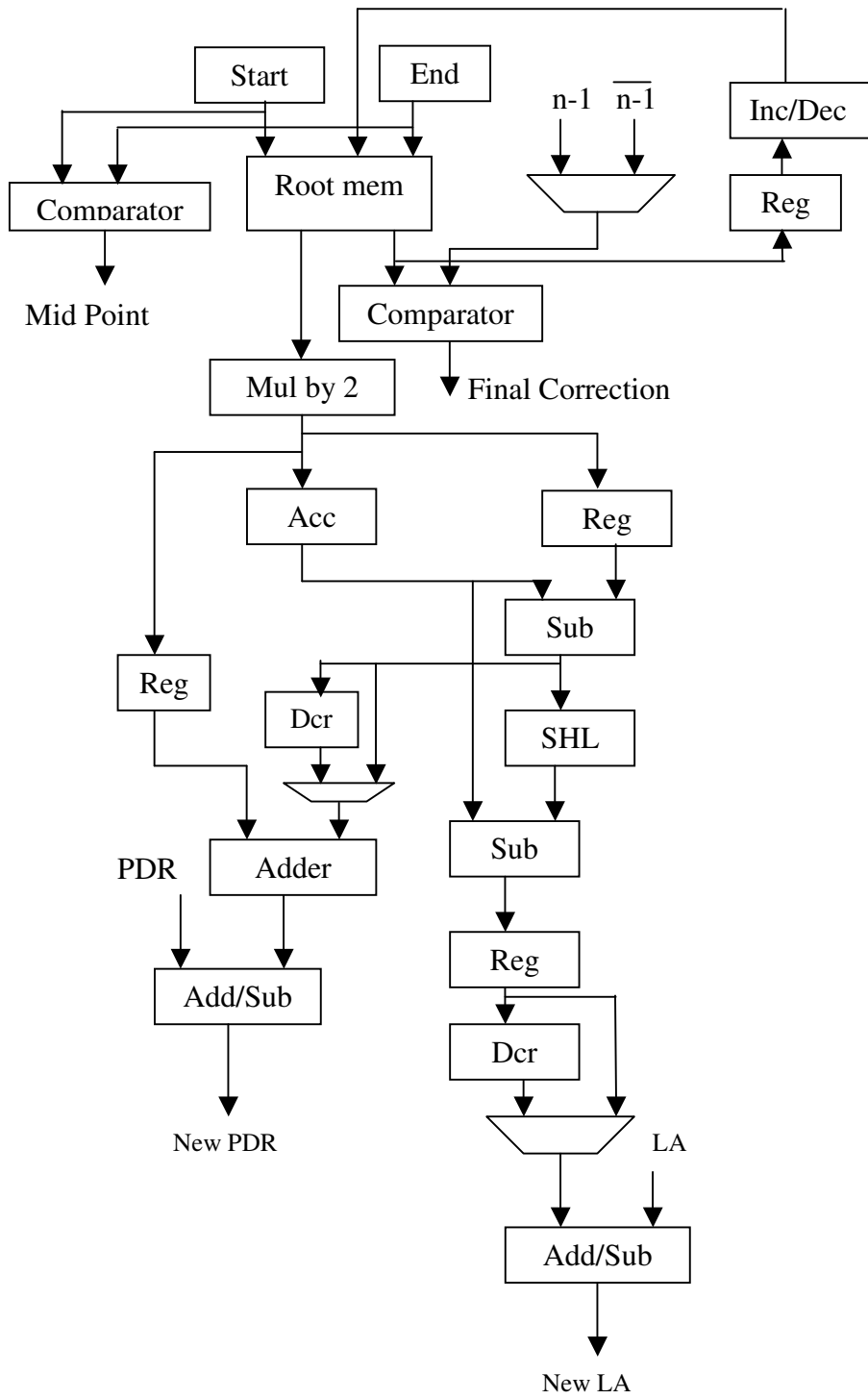


Fig. 3.14 Square root correction unit

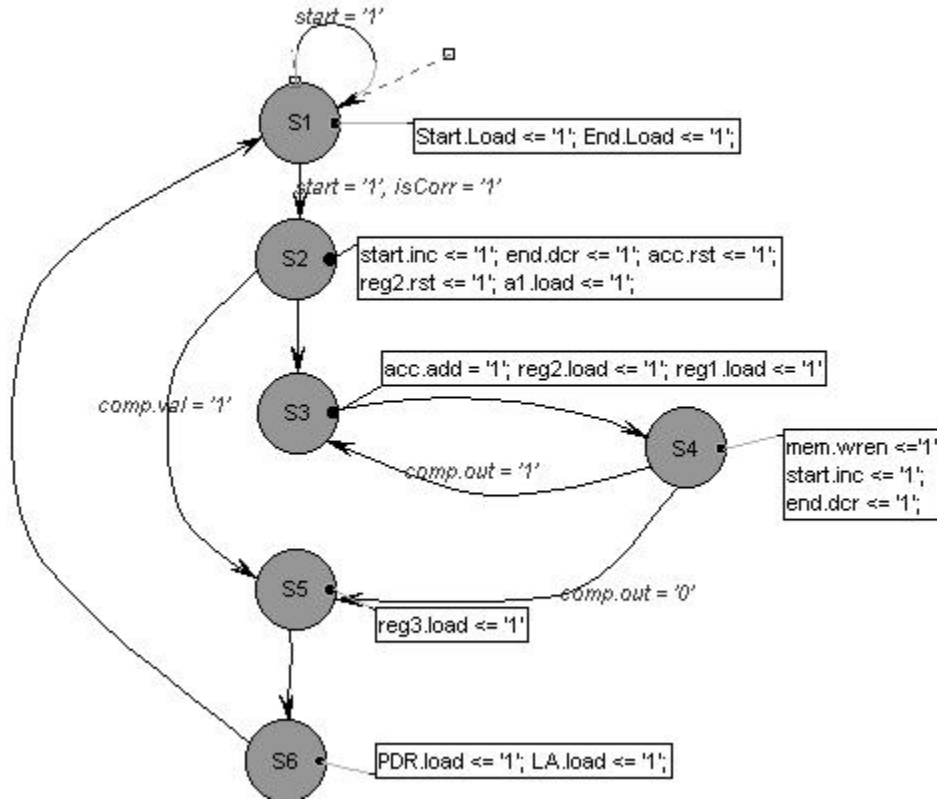


Fig. 3.15 State machine for Square root correction

### 3.6 The Divisibility testing unit

This unit is different from the other units in the sense that some additional operations, apart from the normal preprocessing and post-processing, are performed in the host machine. The host machine calculates the Ekadhika of the divisor with the help of a lookup table. The dividend and the Ekadhika are given to the hardware component. The hardware component performs a series of osculations of the dividend by the Ekadhika. The result is a much-reduced dividend that is zero or one digit more than the divisor. The host performs the divisibility check of the reduced number by the divisor. This is done on the host in a few operations.

The operations involved in the osculation process are multiplication of a vector by a single digit and addition of two vectors. These vector operations are chained in the 'Multiply and Add' unit. The structure of the unit is given in fig 3.16 and 3.17.

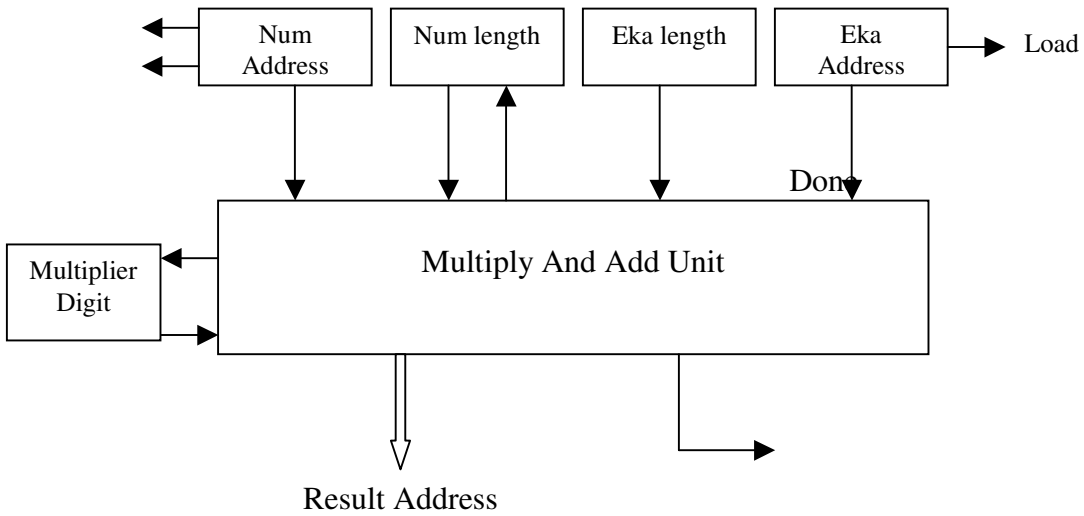


Fig 3.16 Divisibility testing unit

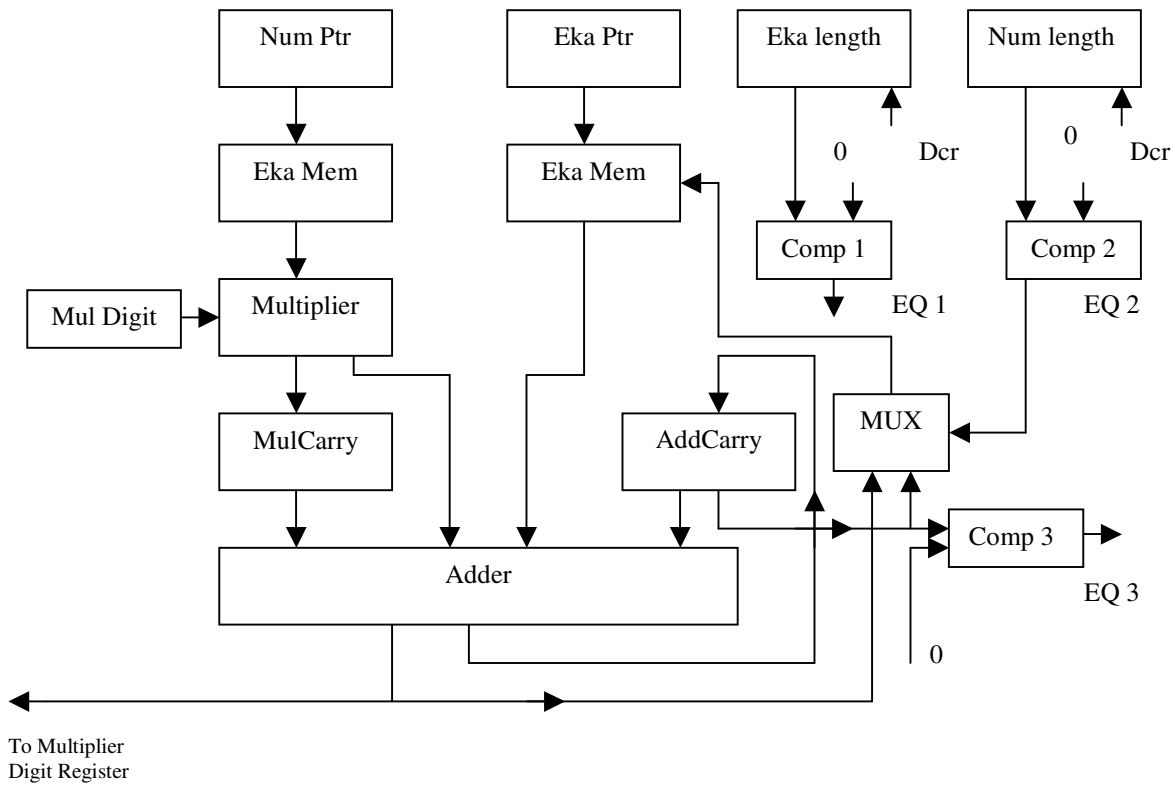


Fig 3.17 Multiply and Add Unit of Divisibility testing

The unit is controlled by a state machine shown in fig. 3.18.

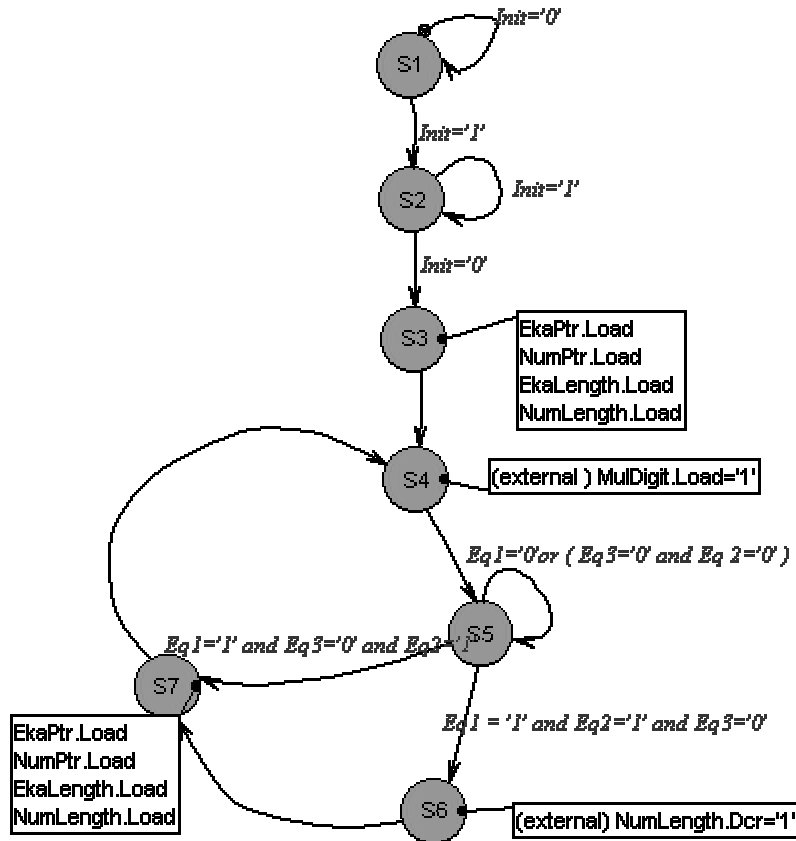


Fig 3.18 State machine for Divisibility Testing

States S<sub>3</sub> and S<sub>7</sub> denote the Start State of one osculation iteration. State S<sub>5</sub> denotes the state in which the Ekadhika digits are being consumed or the carry being propagated. State S<sub>6</sub> is visited when the carry does not propagate beyond the most significant digit. In this case, the length of P gets decremented.

The implementation details of the designs are present in the next chapter.

## CHAPTER 4 - IMPLEMENTATION DETAILS

### 4.1 Implementation overview

The designs of the individual units are specified using VHDL. This VHDL description is given as input to the synthesis tool of the Xilinx Foundation express. This tool generates a netlist for the design. In the net list file, the hardware is specified as a set of basic gates, sequential elements, and the interconnections between them. The functionality of the design is then verified by doing a simulation based on this netlist. Since the netlist does not have the timing information like the propagation delay and the interconnection delay, timing violations are not reported during this simulation. The netlists are then given to the implementation unit of the Xilinx Foundation Express. This maps the gates in the design into CLBs, places them at appropriate positions in the FPGA, performs the interconnections between them, analyses the exact timing of the design and generates the data needed for configuring the FPGA. Using the data obtained from the timing analysis, an exact timing simulation of the design is performed. This helps in finding any timing violations. If there are violations, it is analyzed and corrected and the process is repeated. After doing the timing analysis, the configuration data is downloaded into the device using the XSLOAD software. At the host processor, a C program has been written to interface with the FPGA board. This program clocks the hardware, gives the necessary inputs to the unit through the parallel port and gets back the result. This program also takes care of reconfiguring the device appropriately. The overall flow is given in fig. 4.1.

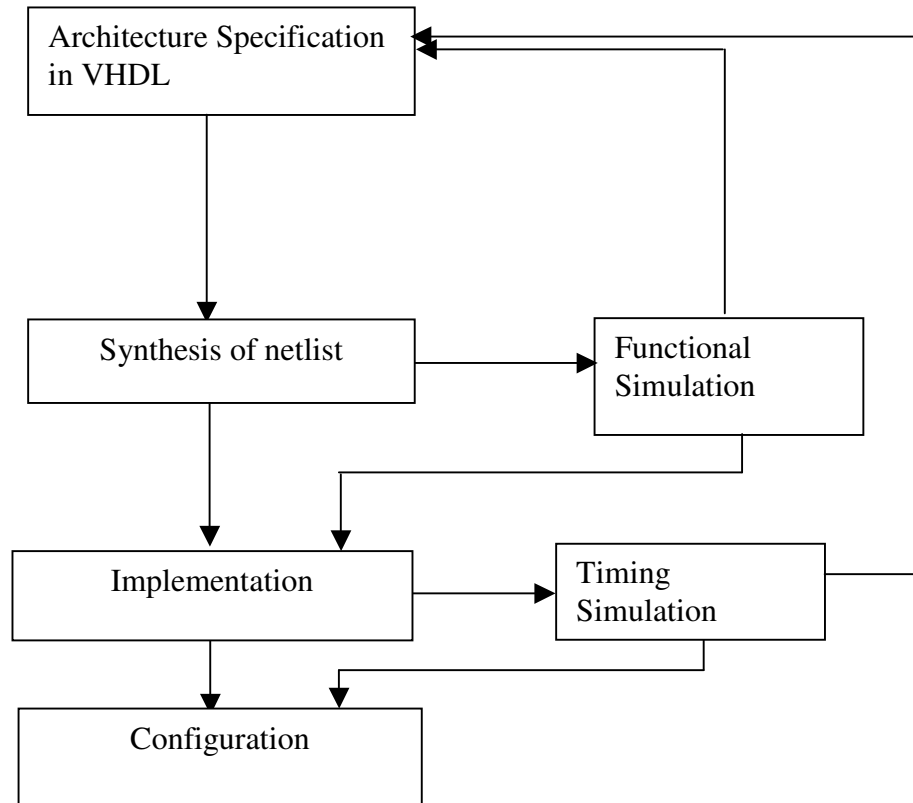


Fig. 4.1 Implementation overview

## 4.2 Implementation of the units

The implementation of the various units are given in this section. Note that, although the designs are applicable to any radix, particular base have been chosen, so that the design can be fit into the available FPGA resources. (The XC4010XL device has 400 CLBs and a maximum operating frequency of 50 MHz)

#### 4.2.1 The Multiplication unit:

The multiplication unit has been implemented with base 256 unsigned numbers. The multiplier, multiplicand and their lengths form the inputs to the unit. The assembling unit, which resides in the FPGA, receives these inputs as a set of four bit numbers, assembles them to form 8 bit values and writes them into the RAM and registers inside the FPGA. When the multiplication is over, it converts them into 4 bit values and sends them to the parallel port. At the host side, the interfacing software takes care of sending and receiving 4 bit values through the parallel port.

The main operation in this unit is calculating the cross product. A cross product of two vectors of size  $n$  has  $n$  multiplications and  $n-1$  additions. In the implementation, two multipliers are used to reduce the time. The number of cycles needed to perform a  $n * n$  cross product is  $n+1$  cycles. To multiply two numbers of length  $m$  and  $n$ ,  $m+n-1$  iterations are needed. The cycles needed for a particular iteration depends on the length of the vector whose cross product is to be calculated during that iteration. The total time needed to perform a  $m * n$  multiplication is  $(mn + m + n - 1)$  cycles.

A snap shot of the timing simulation is shown in fig. 4.2. The timing analysis of the unit has been done and the maximum possible frequency of operation is found to be 15.576 MHz. The space required to implement this unit is 398 CLBs. This includes the size of the three RAMs which are inside the unit and which consume 24 CLBs. Table 4.1 shows the estimated time taken for some cases using the above figures.

No. of Multiplicand digits (m)	No. of Multiplier digits (n)	Time ( in $\mu$ secs)
10	10	7.64
20	10	14.70
20	20	28.18

Table 4.1 Estimated performance of the multiplication unit (with base 256 numbers)

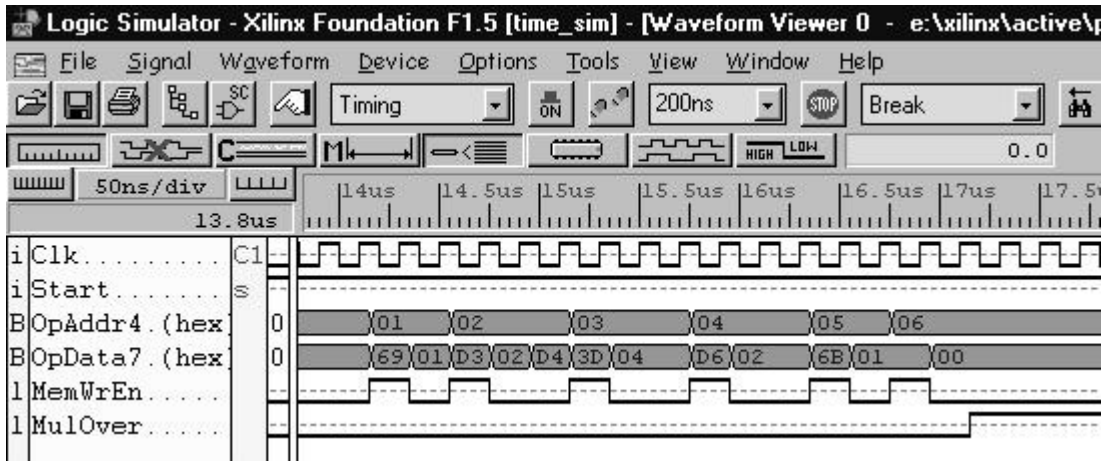


Fig. 4.2 Timing simulation of the multiplication unit

The multiplicand and the multiplier are  $(13\ 13\ 13)_{256}$ . The product obtained is  $(00016BD63DD369)_{256}$ . These values are seen in the output data bus (OpData) when the write enable signal (MemWrEn) is high. When the multiplication is over, the complete signal (MulOver) becomes high.

#### 4.2.2 The Squaring unit

This unit has also been implemented with base 256 unsigned numbers. The inputs to the unit are the number and its length. In the host side, they are disassembled into 4 bit values and sent in the order of number length and the number. The assembling unit assembles them into 8 bit values and writes them to the appropriate registers and memory inside the FPGA. The assembling unit reads the result from the memory, disassembles them into 4 bit values and sends them to the parallel port.

The major operation is the calculation of Dwandwa. A dwandwa of an  $m$  digit number requires  $\lceil m/2 \rceil$  multiplications and  $\lceil m/2-1 \rceil$  additions. This takes  $\lceil m/2+1 \rceil$  cycles. To calculate the square of an  $n$  digit number,  $2n-1$  Dwandwa calculations have to be made. There are two Dwandwa calculations each of  $m$  digit numbers, where  $m$  ranges from 1 to  $n-1$  and one Dwandwa calculation of length  $n$ . Hence the total time required for squaring a  $n$  digit number is  $n^2+2n-1$  cycles.

Using the timing analysis, the maximum frequency of operation of this unit has been found to be 14.272 MHz. The space required to implement this unit is 340 CLBs which includes two memory banks which consume a size of 64 CLBs. Table 4.2 shows the actual time for a few cases. A snap shot of the timing simulation is shown in fig. 4.3.

Number of digits (n)	Time ( in $\mu$ secs)
10	8.34
20	30.76

Table 4.2 Estimated performance of the Squaring unit (with base 256 numbers)

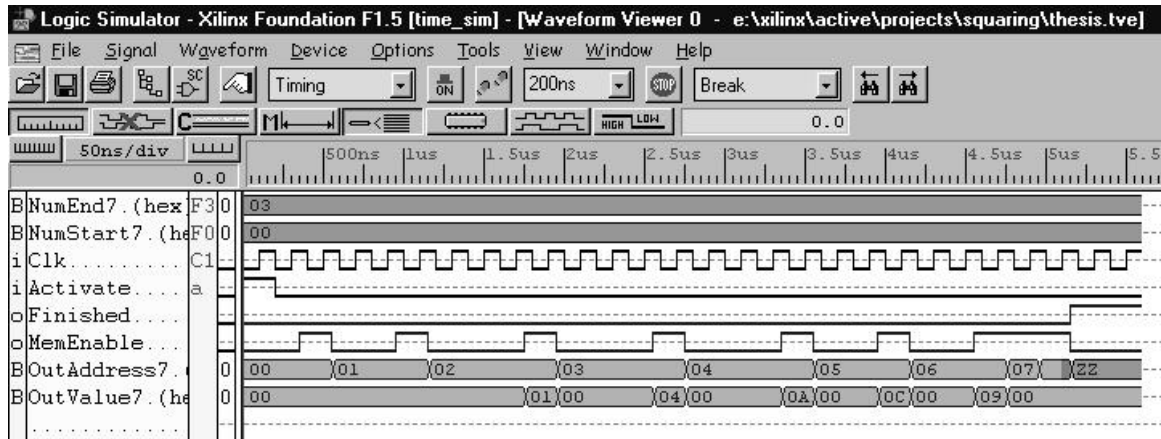


Fig. 4.3 Timing simulation of the squaring unit

The start address is 0 and the end address is 3. The contents of the memory in those locations are 00, 01, 02, 03 respectively. The square of  $(03020100)_{256}$  is  $(00090C0A04010000)_{256}$ . These values are seen in the OutValue (output value) bus. The corresponding output addresses are seen in OutAddress and the write enable is seen in MemEnable.

### 4.2.3 The Division unit

This unit has been implemented with base 8 SDNS numbers. The inputs to the unit are the precision required, length of the divisor, the dividend and the divisor. All these values are 4 bit values. So, the assembling unit receives them and writes them into memory and registers, without the need for any assembling. The result is sent back to the host by the assembling unit.

The major operation in this unit is the calculation of the cross product of divisor digits and the previous quotient digits. The vector whose cross product is needed varies in size till the number of quotient digits obtained is less than the number of divisor digits. After this, the vector size is  $m-1$ , where  $m$  is the size of the divisor. When the divisor size is equal to that of dividend, and the required precision is same as that of the size of the divisor, the number of cycles per quotient is  $\lceil n/4 + 3.5 + 3/n \rceil$ . To obtain a precision of  $p$ , the number of cycles required is  $\lceil np/4 + 3.5p + 3p/n \rceil$ . This value will change based on the number of corrections. Analysis of this algorithm [2] has shown that the average number of corrections needed for  $p$  digits is  $0.075p$ . This also contributes to the total time of operation. On an average, a single correction operation takes up three cycles. So the total number of cycles needed for corrections is  $0.225p$ . Adding this to the number of cycles needed in a normal operation, we get a total time of  $\lceil 3.725 p + np/4 + 3p/n \rceil$  cycles

By doing the timing analysis, the maximum frequency of operation of this unit has been found to be 16.313 MHz. The space required to implement this unit is 336 CLBs which include two memory banks which consume a size of 12 CLBs. Table 4.3 gives the estimated time for a few cases, assuming that  $m = n$ . A snap shot of the timing simulation is shown in fig. 4.4.

No. of digits of divisor (n)	Required precision (p)	Time (in $\mu$ secs)
10	10	4.00
10	20	8.00
20	10	5.44
20	20	7.815

Table 4.3 Estimated performance of the division unit with base 8 numbers

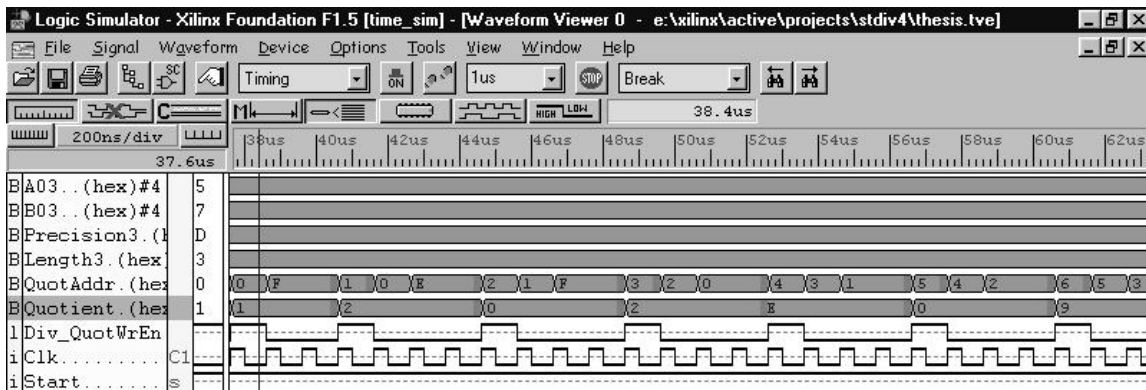


Fig. 4.4 Timing simulation of the division unit

The divisor is  $(5713)_8$  and the dividend is  $(747000)_8$ . The quotient digits are seen in the quotient bus, and their corresponding addresses in the QuotAddr bus. Note that the quotient values are valid only when the QuotWrEn signal is high.

#### 4.2.4 The Square root unit

This unit has been implemented for SDNS base 8 numbers. The inputs to this unit are the root ( $2a_0$ ) of the most significant two digits, the remainder obtained when the most significant two digits are divided by  $2a_0$ , the rest of the digits of the number, the length of the digits and the required precision. Among these, the size of the remainder is 5 bits, while the rest are of size 4 bits. Hence the remainder is obtained in two cycles and the values are assembled by the assembling unit. After the completion of operation, the digits of the square root, other than the first digit, are sent to the host through the parallel port.

The interfacing software on the host adds these digits with the first digit to obtain the result.

One of the main operations involved in square root is the calculation of Dwandwa. The calculation of the Dwandwa of an  $m$  digit number requires  $\lceil m/2 \rceil$  multiplications and  $\lceil m/2 - 1 \rceil$  additions. To get the square root with a precision of 'p' digits, a total of  $p-1$  Dwandwa calculations are required. The  $i^{\text{th}}$  dwandwa calculation is done on an  $i$  digit number and hence requires  $\lceil i/2 \rceil$  multiplications and  $\lceil i/2 - 1 \rceil$  additions. This takes  $\lceil i/2 \rceil + 1$  cycles. Other than Dwandwa calculation, for getting a digit of the quotient, two more cycles are needed. Hence, if there is no correction, a square root of precision 'p' requires  $n/4 + 3.5$  cycles. The average number of corrections has been found to be 0.003 [1] per digit. On an average, 4 cycles are needed for correction per digit. Hence to get a square root of precision 'p', a total of  $[np/4 + 3.512p]$  cycles are required.

By doing the timing analysis, the maximum frequency of operation of this unit has been found to be 17.039. The space required to implement this unit is 315 CLBs which include two memory banks which consume a size of 8 CLBs. Table 4.4 gives the actual timing for a few cases. A snap shot of the timing simulation is shown in fig. 4.5.

No. of digits (n)	Required Precision (p)	Time (in $\mu$ seconds)
10	10	3.528
10	20	7.056
20	10	4.996
20	20	9.991

Table 4.4 Estimated performance of the Square root unit (with base 8 numbers)

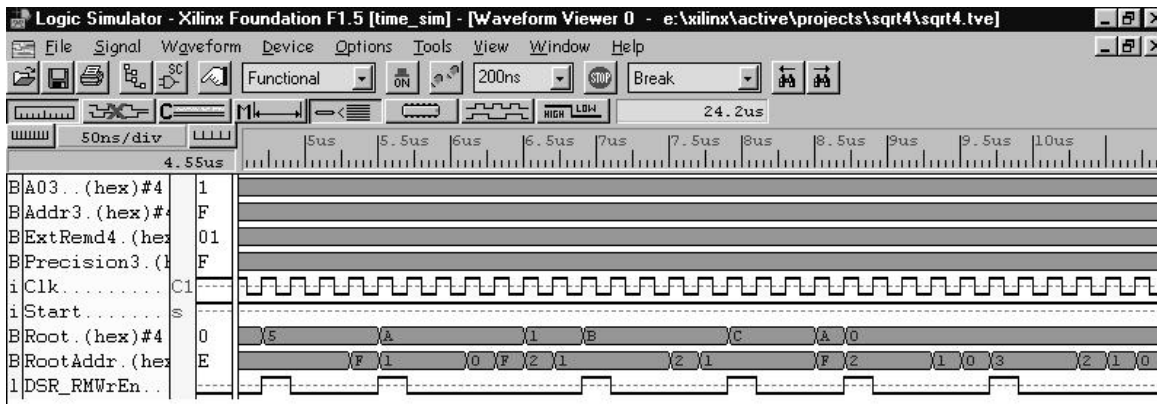


Fig. 4.5 Timing simulation of square root unit

The input given in this simulation is  $(33420)_8$ . In this number the square root of 03 is given as input (A0) and the remainder when 03 is divided by 2A0 is 1, which is given as the external remainder (ExtRemd). The square root digits, other than the first digit are (5, 4, 0, ) etc. These values are seen in Root bus and the corresponding addresses in the RootAddr bus. The values in Root bus are written only when the enable (DSR\_RMWrEn) signal. Note that for the same address 1, three values of root are generated because of correction. The final value is the corrected value.

#### 4.2.5 The Divisibility testing unit

This unit has been implemented with base 256 unsigned numbers. The inputs to the unit are the dividend, dividend length, the Ekadhika, Ekadhika length and the least significant dividend digit. In the host side, the Ekadhika of the divisor is calculated and all the data is disassembled into 4 bit values and sent. The assembling unit assembles them into 8 bit values and writes them to the appropriate registers and memory inside the FPGA. After the completion of the operation, the assembling unit reads the result from the memory, disassembles them into 4 bit values and sends them to the parallel port. If the length of the divisor is  $n$ , the result is the  $n+1$  digit number obtained after the osculation process. The host then checks  $n+1$  digit by  $n$  digit divisibility through standard techniques and reports the result.

The major operation is the osculation process. If  $m$  is the length of the dividend,  $n$  the length of the Ekadhika,  $p$  the number of times the carry propagates beyond the most significant digit of the dividend and  $t$  the average length of carry propagation beyond the  $n^{\text{th}}$  digit of the dividend, the number of cycles taken by the divisibility unit for reducing the  $m$  digit divisor to an  $n$  digit number is given by:

$$\text{Cycles} = 3 + (2 + n + t - p) * (m - n - 1 + p)$$

Using the timing analysis, the maximum frequency of operation of this unit has been found to be 13.902 MHz. The space required to implement this unit is 400 CLBs which includes two memory banks which consume a size of 128 CLBs.. Table 4.5 shows the actual time for a few cases. A snap shot of the timing simulation is shown in fig. 4.6.

*Note:* The values of  $t$  and  $p$  seldom exceed 1. This results in this table calculated assuming  $t$  and  $p$  to be 1.

Length of Dividend (m)	Length of Divisor (n)	Time (in $\mu$ seconds)
40	20	31.85
60	30	69.24
120	70	259.05

Table 4.5 Estimated performance of the Divisibility unit (with base 256 numbers)

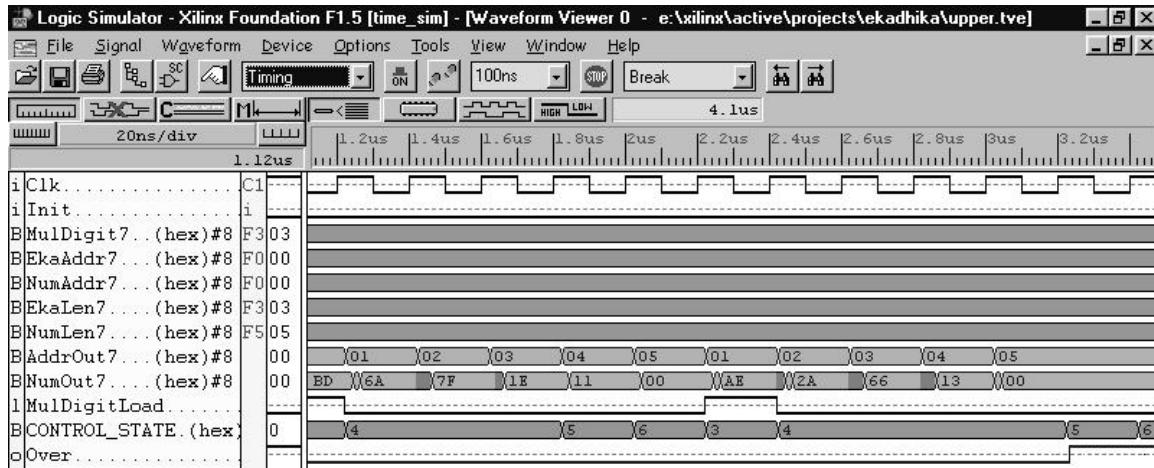


Fig. 4.6 Timing simulation of Divisibility unit

The inputs given in this simulation are Dividend =  $(111E76270103)_{256}$ . Ekadhika= $(031694)_{256}$ , corresponding to the Divisor  $(0B2289)_{256}$ . The least significant digit of the dividend  $(03)_{256}$  is given as input to MulDigit, and the starting addresses of the dividend and the Ekadhika are given as inputs to EkaAddr and NumAddr respectively. The lengths of the dividend and the Ekadhika are given as inputs to EkaLen and NumLen. Control\_State values correspond to the states of the state machine control. "Over" going high denotes the completion of the operation. AddrOut addresses the result memory. The result,  $(13662AAE)_{256}$  is put out in the NumOut Bus.

Unit	No. of CLBs including memory	No. of CLBs excluding memory	Maximum frequency (in MHz)
Multiplication	398	374	15.576
Squaring	340	276	14.272
Straight division	336	324	16.313
Square root	315	307	17.039
Ekadhika	400	272	13.902

Table 4.6 Time and space complexity of the units

Table 4.6 summarises the space requirements and the maximum speed of operation of all the units. All the designs have been tested and implemented.

## CHAPTER 5 – CONCLUSION

### 5.1 Summary

The project has attempted to evaluate the implementation of ancient Indian arithmetic algorithms on a reconfigurable architecture. The algorithms implemented were Urdhva Tiryak multiplication, Dwandwa squaring, Straight Division, Dwandwa square root determination, and Ekadhika divisibility testing. These algorithms have the advantage of being in-place, online, having scope for parallelism and being applicable to any radix. The choice of reconfigurable hardware was made because the units have many common functional elements and hence are with a large potential for resource re-use. The design and implementation of the various units has been done using Xilinx 4000XL series FPGA and Xilinx Foundation series 1.5i software. The results of the implementation have been presented.

On the basis of the work done, the following can be said:

- The algorithms perform better when there is a support at the hardware level for the characteristics of the algorithms like VLP representation, SDNS etc.
- The design time and complexity is highly reduced because the units share many basic structural elements. These elements – designed using VHDL, could be reused easily.
- The parallelism in the algorithms has been exploited to an extent (eg. two multipliers in division and multiplication units). This could be improved further.
- Normally the algorithms are efficient for larger radix. But a smaller radix has been selected in the design due to resource constraints.
- Given more resources (Larger FPGA) and the facility to reconfigure the FPGA partially, a significant improvement in the performance can be obtained. With such resources, an implementation of this kind would be a better alternative for special applications involving large numbers.

## 5.2 Suggested future work

The following can be suggested as possible future extensions for the work done:

- Some algorithmic enhancements tried out in software [1] (like the use of least remainder division, using Straight Division to find the remainder of the division etc.) can be explored.
- An evaluation of the effectiveness of partial reconfiguration and larger FPGA can be attempted.
- Apart from the algorithms considered, there are some more ancient Indian algorithms [11] that have been found efficient (e.g. Chakravala algorithm, for rational approximation of square roots). The performance of such algorithms when implemented in hardware can be evaluated.
- The applicability of these algorithms in applications involving large numbers (e.g. Cryptography) can be explored.
- An attempt to find more applications, where these algorithms might be applied for improving performance can be made.

## APPENDIX

### A.1 Straight Division

#### A.1.1 The straight division algorithm

The inputs to this algorithm are two numbers, the divisor and the dividend, of arbitrary length and base. For sake of simplicity assume that they are positive integral numbers. Decimal point manipulation and sign processing are trivial tasks.

*Step 1:* Consider the first digit of the divisor. Call this the abridged divisor  $D$ . Let the rest of the digits form the number  $S$ . The entire process of division is carried out with this number  $D$ .

*Step 2:* Consider the first digit of the dividend. If it is less than  $D$ , consider the first two digits. This forms the partial dividend. Divide this with  $D$  to get the first quotient digit and the remainder digit  $R$ .

*Step 3:*  $R$  is combined with the next dividend digit to form the next gross dividend. i.e. Multiply  $R$  with base and add it to the next dividend digit to form the gross dividend.

*Step 4:* Multiply the least significant quotient digit with the first digit of  $S$ . The next least significant digit with the second and so on, till all the digits of  $S$  or all digits of the quotient are consumed. Let it be  $CP$ . What we are doing here is essentially a cross product of the quotient digits with the divisor.

*Step 5:* Subtract  $CP$  from the gross dividend to obtain the partial dividend. Divide the partial dividend by  $D$  to obtain the next quotient digit and the remainder  $R$ .

*Step 6:* Right pad the dividend with zeros and carry out steps 3,4 and 5 till the desired precision.

***Corrections:***

Since the division is carried out only using the first digit of the divisor, there is a possibility that we overestimate the quotient digits. Under such circumstances, the partial dividend becomes negative. The last generated quotient digit is decremented by one and the remainder and partial dividend adjusted accordingly. This process is done till the partial dividend becomes positive.

***Use of Normalization:***

It is observed that smaller abridged divisors give more corrections, as there is more probability of over estimating the quotient digits. To remedy this, the divisor and the dividend is multiplied by a constant such that the first digit of the divisor becomes greater than half the base.

***Use of Look ahead:***

At any step of the division process, all the quotient digits except one, needed for the next cross product is generated. This can be used to further refine our estimate of the quotient digit. The least significant digit of the quotient is multiplied with the second digit of S, the next least with the third and so on till all quotient digits or all digits of S are consumed. This is the look ahead. This look ahead is divided by the base and added to the cross product. This sum is subtracted from the gross dividend to get the partial dividend. Dividing the partial dividend by D generates the quotient and the remainder digits by D. Divide the look ahead by the base and add it to the remainder to get R. The calculation of the look ahead does not impose any additional computational complexity, as the look ahead form a component of the next cross product, and can be used.

### Use of SDNS

If signed digit number systems were to be used, each digit would have its own sign. Under such circumstances, we need not generate corrections if the partial dividend becomes negative. We simply generate a negative quotient digit and proceed. However use of SDNS does not eliminate the occurrence of corrections altogether. We would still be forced to correct if the quotient obtained by dividing the partial dividend by D produces more than one digit. Use of SDNS, Normalization and Look ahead reduces the average number of corrections to 0.002 [1].

#### A.1.2 Straight division - examples

**Example 1:** Divide 31689 by 71 - basic algorithm, no corrections and no look ahead.

Let CP be cross product, GD the gross dividend, PD the partial dividend, Q the quotient and R the remainder. The abridged divisor is 7 and S is 1.

$$\begin{array}{r|l}
 1 & 31689.000 \\
 7 & \phantom{3}3 \\
 \hline
 & \phantom{3}4
 \end{array}$$

Divide 31 by 7 .R = 3  
 Q = 4.  
 GD= 36.

$$\begin{array}{r|l}
 1 & \phantom{3}32 \\
 7 & 31689.000 \\
 \hline
 & \phantom{3}44
 \end{array}$$

Subtract CP of 4 and 1 from GD=36  
 to PD =32.  
 Divide 32 by 7 obtain R=4 and Q=4.  
 Next GD=48



**Example 2.** Divide 31679 by 7121- with corrections and no Look ahead

Abridged Divisor=7

121	31	3 1 6 7 9 . 0 0 0	PD = 31
7	3	3	Q = 4
	4		R = 3
			GD = 36

121	31 32	3 1 6 7 9 . 0 0 0	PD = 36-CP(4,1) = 32
7	3 4	3 4	Q = 4
	4 .4		R = 4
			GD = 47

121	31 32 35	3 1 6 7 9 . 0 0 0	PD = 47-CP(44,12) = 35
7	3 4 0	3 4 0	Q = 5
	4 .4 5		R = 0
			GD = 9

121	31 32 35 63	3 1 6 7 9 . 0 0	PD = 9-CP(445,121) = -8 Correction
7	3 4 7 0	3 4 7 0	Correct earlier quotient to 4. New
	4 .4 <del>5</del> 9		GD= 79
			PD = 79-CP(444,121) = 63 Q=9 R=0

The quotient when 31679 is divided by 7121 is 4.449. The next step would involve a correction of 9 to 8 as the GD now is 0 and PD would become negative.

**Note:** In the above step, if SDNS were to be used, correction need not be performed. The step is reworked as:-

121	31 32 35 -8	
7	3 1 6 7 9 . 0 0	
	3 4 0 0	
	4 . 4 5 $\bar{2}$	PD = 9-CP(445,121) = -8 Q = -2 R = 6

The quotient is  $4.45\bar{2}$ , which is nothing but 4.448.

**Example 3** - The above computation, with look ahead

121	31	
7	3 1 6 7 9 . 0 0 0	
	3	
	4	PD = 31 Q = 4 R = 3 GD = 36
121	31 32	
7	3 1 6 7 9 . 0 0 0	
	3 4	
	4 . 4	LA = CP(4,2) = 8 PD = 36-CP(4,1) - 8/10 = 32 Q = 4 R = 4 GD = 47

$$\begin{array}{r|l}
 & 31 \ 32 \ 34 \\
 121 & 3 \ 1 \ 6 \ 7 \ 9 \ . \ 0 \ 0 \ 0 \\
 7 & 3 \ 4 \ 7 \\
 \hline
 & 4 \ . \ 4 \ 4
 \end{array}$$

$$\begin{aligned}
 LA &= CP(44,21) = 12 \\
 PD &= 47 - CP(44,12) - 12/10 = 34 \\
 Q &= 4 \\
 R &= 7 \\
 GD &= 79
 \end{aligned}$$

$$\begin{array}{r|l}
 & 31 \ 32 \ 34 \ 62 \\
 121 & 3 \ 1 \ 6 \ 7 \ 9 \ . \ 0 \ 0 \ 0 \\
 7 & 3 \ 4 \ 7 \ 7 \\
 \hline
 & 4 \ . \ 4 \ 4 \ 8
 \end{array}$$

$$\begin{aligned}
 LA &= CP(44,21) = 12 \\
 PD &= 79 - CP(444,121) - 12/10 = 62 \\
 Q &= 8 \\
 R &= 7
 \end{aligned}$$

Here we observe that the correction in the second place of the decimal is avoided by using look ahead

## A.2 Square rooting

### A.2.1 The square rooting algorithm

Without loss of generality, let us assume that the input number is an integral number of arbitrary base and length. The output of this algorithm is the number's square root with arbitrary precision.

First the digits of the input number  $N$  is grouped into groups of two starting from the rightmost digits.

*Step 1* Determine the first most significant non zero group of the given number  $N$ . Let it be  $G$

*Step 2* Set divisor  $D=2S$  where  $S$  is the largest number whose square is equal to or less than the first non zero group.  $S$  forms the first digit of the square root.

*Step 3* Set remainder  $R= G-D*D$ .

*Step 4* Combine the remainder with the next digit of  $N$  to form the next partial dividend. ( if  $p$  is the next digit, the partial dividend is  $R*Base+ p$  )

*Step 5* Divide the partial dividend by  $D$  to obtain the next remainder  $R$ . The quotient forms the second digit of the square root.

*Step 6* Combine the remainder with the next digit of  $N$  to form the gross dividend. The procedure is same as the one explained in Step 4

*Step 7* There is a slight variation from this step onwards. From the gross dividend, subtract the Dwandwa of the generated square root digits other than the first to obtain the partial dividend.

*Step 8* Divide the partial dividend by  $D$  to obtain the next square root digit and the quotient. This step is repeated till all the digits of the number are consumed. Arbitrary precision can be obtained by padding the number with zeros and proceeding with the algorithm.

If  $M$  is the number of integral digits in the number, the number of integral digits in the solution is equal to  $M/2$  when  $M$  is even and  $(M+1)/2$  when  $M$  is Odd.

**Corrections:** In certain cases, when the Dwandwa of the square root digits is subtracted from the gross dividend to obtain the partial dividend, it is observed that the partial dividend becomes negative. In such cases, the last determined square root digit is reduced by 1 and the remainder and Dwandwa adjusted accordingly. This correction is repeated till the partial dividend becomes positive.

*Use of Look Ahead* We observe that corrections are generated when the difference between the gross dividend and the Dwandwa becomes negative. At any step, before the generation of the quotient digit, all the digits for the next Dwandwa except one digit is known. This information can be used to generate better estimates for the square root digit. In the revised algorithm, calculate the Dwandwa of the square root digits (except the first two). Divide it by the base. This is the look ahead. Call it L. Add it to the Dwandwa of the square root digits (except the first digit) and subtract this sum from the gross dividend to get the next partial dividend. Divide this by the divisor to get the next square root digit. Add the remainder to L to get R.

The calculation of the look ahead does not contribute to any computational overhead, because this value is needed for the calculation of the next dwandwa and can be used.

The modified algorithm using SDNS, Normalization and look ahead reduces the average correction per digit to 0.003 for radix 256 numbers[2].

### A.2.2 Square rooting - examples

Determine the Square root of 1947 without SDNS and look ahead

Let GD denote the gross dividend, PD the partial dividend, Q the quotient and R the remainder. Let LA denote the look ahead and D(X) denote the Dwandwa of X.

<div style="display: flex; justify-content: space-between; align-items: center;"> <span style="margin-right: 10px;">8</span> <span>19 4 7 0 0 0</span> </div> <div style="text-align: center; margin-top: 5px;">3</div> <hr style="border: 0.5px solid black;"/> <div style="text-align: center; margin-top: 10px;">4</div>	<p>Integral portion of Square root of 19 is 4. Divisor = <math>2 \times 4 = 8</math></p>
<div style="display: flex; justify-content: space-between; align-items: center;"> <span style="margin-right: 10px;">8</span> <span>19 4 7 0 0 0</span> </div> <div style="text-align: center; margin-top: 5px;">34</div> <div style="text-align: center; margin-top: 5px;">3 2</div> <hr style="border: 0.5px solid black;"/> <div style="text-align: center; margin-top: 10px;">4 4</div>	<p>GD = 34, PD = 34 Q = 4 R = 2</p>
<div style="display: flex; justify-content: space-between; align-items: center;"> <span style="margin-right: 10px;">8</span> <span>19 4 7 0 0 0</span> </div> <div style="text-align: center; margin-top: 5px;">34 11</div> <div style="text-align: center; margin-top: 5px;">3 2 3</div> <hr style="border: 0.5px solid black;"/> <div style="text-align: center; margin-top: 10px;">4 4 1</div>	<p>GD = 27 PD = <math>27 - D(4) = 11</math> Q = 1 R = 3</p>
<div style="display: flex; justify-content: space-between; align-items: center;"> <span style="margin-right: 10px;">8</span> <span>19 4 7 0 0 0</span> </div> <div style="text-align: center; margin-top: 5px;">34 11 22</div> <div style="text-align: center; margin-top: 5px;">3 2 3 6</div> <hr style="border: 0.5px solid black;"/> <div style="text-align: center; margin-top: 10px;">4 4. 1 2</div>	<p>GD = 30 PD = <math>30 - D(41) = 22</math> Q = 2 R = 6</p>

8	$\begin{array}{r} 34 \ 11 \ 22 \ 43 \\ 19 \ 4 \ 7 \ 0 \ 0 \ 0 \\ 3 \ 2 \ 3 \ 6 \ 3 \\ \hline 4 \ 4. \ 1 \ 2 \ 5 \end{array}$	$\begin{aligned} GD &= 60 \\ PD &= 60 - D(412) = 43 \\ Q &= 5 \\ R &= 3 \end{aligned}$
---	--	--

8	$\begin{array}{r} 34 \ 11 \ 22 \ 43 \ 74 \\ 19 \ 4 \ 7 \ 0 \ 0 \ 0 \ 0 \\ 3 \ 2 \ 3 \ 6 \ 11 \ 4 \\ \hline 4 \ 4. \ 1 \ 2 \ 5 \ 4 \ 9 \end{array}$	$\begin{aligned} GD &= 30, \\ PD &= 30 - D(4125) = -14 \text{ Correction} \\ &\text{Correct Previous Digit to 4.} \\ \text{New GD} &= 110 \\ PD &= 110 - D(4124) = 74 \quad Q = 9 \quad R = 4 \end{aligned}$
---	--	--

Square root of 1947 is 44.1249. Note that in the next step 9 would be corrected to 8 as the PD would become negative when D(41249) is subtracted from 40.

**Note:** In the above step, if SDNS were to be used, corrections need not be generated.

8	$\begin{array}{r} 34 \ 11 \ 22 \ 43 \ \overline{14} \\ 19 \ 4 \ 7 \ 0 \ 0 \ 0 \ 0 \\ 3 \ 2 \ 3 \ 6 \ 3 \ 2 \\ \hline 4 \ 4. \ 1 \ 2 \ 5 \ \overline{2} \end{array}$	$\begin{aligned} GD &= 30, \\ PD &= 30 - D(4125) = -14 \\ Q &= -2 \\ R &= 2 \end{aligned}$
---	---	--

Square root of 1947 is 44.125 $\overline{2}$  which is equivalent to 44.1248

**Example:** Determine the square root of 19, with look ahead and SDNS

8	19 0 0 0 0 0
	3
	4
8	30 19 0 0 0 0 0
	3 6
	4. 3
8	30 51 19 0 0 0 0 0
	3 6 3
	4. 3 6
8	30 51 22 19 0 0 0 0 0
	3 6 3 2
	4. 3 6 -1
8	30 51 22 -9 19 0 0 0 0 0
	3 6 3 2 -2
	4. 3 6 -1 -1

Integral portion of Square root of  
19 is 4.  
Divisor =  $2 \times 4 = 8$

GD = 30,  
PD = 30  
Q = 3  
R = 6

GD = 60  
PD =  $60 - D(3) = 51$   
Q = 6  
R = 3

GD = 30  
LA =  $D(6) = 36$   
PD =  $30 - D(36) - LA/10 = -9$   
Q = -1  
R = 2

GD = 20  
LA =  $D(61) = -12$   
PD =  $20 - D(361) - LA/10 = -9$   
Q = -1  
R = -2

Square root of 19 is 4.3589.

### A.3 Divisibility testing

#### A.3.1 Divisibility testing - example

1. To determine if  $N = (134724309)_{10}$  is divisible by  $M = (237)_{10}$

*Step 1:* To Determine Ekadhika of 237

$$\text{Multiply 237 by 7} : 237 * 7 = 1659$$

$$\text{Add 1 to the result} : 1659 + 1 = 1660$$

$$\text{Drop Trailing zero} : 166$$

$$E(237)_{10} = 166$$

*Step 2:* Multiply 166 by the last digit of 134724309

$$166 * 9 = 1494$$

*Step 3:* Add the result to 134724309 after dropping the last digit

$$13472430 + 1494 = 13473924$$

*Step 4:* Multiply 166 by the last digit of 13473924

$$166 * 4 = 664$$

*Step 5:* Add the result to 13473924 after dropping the last digit

$$664 + 1347392 = 1348056$$

*Step 6:* Proceed till the number is reduced to 4 digits

$$166 * 6 = 996$$

$$996 + 134805 = 135801$$

$$166 * 1 = 166$$

$$166 + 13580 = 13746$$

$$166 * 6 = 996$$

$$996 + 1374 = 2370$$

2370 is divisible by 237. Hence 134724309 is divisible by 237.

*Note:* Let the length of the divisor be  $r$ . The algorithm can be carried out till the dividend gets reduced  $r$  digits. This makes divisibility determination easier as we can visually examine and declare if the reduced number is divisible by the divisor. However, in certain cases, the number will not reduce to  $r$  digits as the  $r+1$  digit number will repeat after the osculation operation. During such cases the dividend is declared to be divisible by the divisor.

Note 2: A Number can repeat after the multiply and add operation with the Ekadhika only if it is of length  $r+1$  or less where  $r$  is the number of digits in the Ekadhika.

***Informal Proof:*** Let the length of the Ekadhika be  $r$  digits. Let the number  $N$  be  $m+1$  ( $m > r$ ) digits long. Multiplying the Ekadhika by 1 digit produces at most a  $r+1$  digit number  $S$ . Add  $S$  to  $N$  after dropping the last digit. Let the result be  $T$ . For  $T$  to be equal to  $N$ , they should have the same number of digits, which implies that there was a carry out of the  $m^{\text{th}}$  digit after the addition process. This digit can only be 1, ( as the addition of 2 digits can produce at most a carry of 1 ), which implies that the earlier digit should have been 9. This means that the first digit of the number  $N$  was 9. So  $T$  cannot be equal to  $N$ .

## REFERENCES

- [1] Ranjani Parthasarathi, Ashok Jhunjhunwala - "Modified Straight Division: A Computer implementation of multi-precision division", Microprocessing and Microprogramming, V41 (1995) 193-209.
- [2] Ranjani Parthasarathi, Ashok Jhunjhunwala - "Multi-precision square root using the Dwandwa square root algorithm", Journal of Systems Architecture, 44 (1997) 143-158.
- [3] S.C.Kak and A.O.Barbir-"The Brahmagupta Algorithm for Square Rooting", IEEE Transactions on Computers (1989) 456-459
- [4] JSS Bharathi Krishna Tirthaji Maharaj - "Vedic Mathematics", Motilal Banarasi Das, Varanasi, 1986.
- [5] Xilinx data book, Xilinx Inc., 1999.
- [6] Alexandre F. Tenca, Milos D. Ercegovac - "A Variable Long- precision arithmetic unit design for reconfigurable processor architectures", proceedings of the FCCM, 1998, pp.
- [7] Kishor S. Trivedi, Milos D. Ercegovac - "Online algorithms for division and multiplication", IEEE transactions on computers, Volume C-26, No. 7, July 1977, pp.
- [8] D.E.Knuth-"The Art of Computer Programming",Vol.2(Semi-numerical algorithms)Addison Wesley, 1980.
- [9] Kai Hwang, "Computer Arithmetic - Principles, Architecture and Design", John Wiley & Sons Inc. 1979.
- [10] Ashok Jhunjhunwala - "Indian Mathematics - an Introduction", Wiley Eastern Limited, Madras,1995.
- [11] Bibutibhusan Datta and Avadhesh Narayan Singh - "History of Hindu mathematics", Part II (Algebra), Motilal Banarsi Das, Lahore, 1938.