TRIGONOMETRIC FUNCTION GENERATOR IMPLEMENTATION ON FPGA

Thesis report submitted towards the partial fulfillment of requirements for the award of the degree of

Master of Technology (VLSI Design & CAD)

Submitted by

Anuradha Garg
Roll No 6040404

Under the Guidance of

Mrs. Alpana Agarwal
Assistant Professor, ECED

Department Of Electronics and Communication Engineering
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY,
(Deemed University)
PATIALA – 147004, INDIA
JUNE, 2006

DECLARATION

I hereby declare that the thesis report entitled "Trigonometric Function generator implementation on FPGA" is an authentic record of my own work carried out as requirements for the award of degree of M.Tech. (VLSI Design & CAD) at Thapar Institute of Engineering & Technology (Deemed University), Patiala, under the guidance of Mrs. Alpana Agarwal, Assistant Professor, ECED during January to June, 2006.

Anuradha Garg
Roll No. 6040404

Date:__________________

Certified that the above statement made by the student is correct to the best of my knowledge and belief.

Mrs. Alpana Agarwal
Assistant Professor, ECED

Countersigned by:

Head,
Electronics & Communication
Engineering Department

Dean,
Academic Affairs

ACKNOWLEDGEMENTS

To discover, analyze and to present something new is to venture on an untrodden path towards an unexplored destination is an arduous adventure unless one gets a true torchbearer to show the way. This enlightening guidance, I found in my revered guide Mrs. Alpana Agarwal, Assistant Professor, Electronics & Communication Engineering Department, Thapar Institute of Engineering & Technology (Deemed University), Patiala, without whose patronization it was never possible to give final shape to this thesis. I express my heartfelt gratitude towards her for her valuable guidance, encouragement, constant involvement, inspiration and the enthusiasm with which she solved my difficulties.

I shall be failing in my duties if I do not express my deep sense of gratitude towards Dr. R. S. Kaler, Professor & Head of the Department, Electronics & Communication Engineering Department and Dr. S.C. Chatterjee , P.G. Coordinator, Electronics and Communication Engineering Department.

I would also like to thank all the staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis.

I am also thankful to the authors whose works I have consulted and quoted in this work. Last but not the least I would like to thank God for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Anuradha Garg

M.Tech (VLSI Design & CAD)

ABSTRACT

CORDIC is an acronym for COrdinate Rotation Digital Computer. It is a class of shift-add algorithms for rotating vectors in a plane, which is usually used for the calculation of trigonometric functions, multiplication, division and conversion between binary and mixed radix number systems of DSP applications, such as Fourier Transform. The Jack E. Volder's CORDIC algorithm is derived from the general equations for vector rotation.

A new algorithm was presented by R. E. Fowkes[23] which significantly reduces the minimum amount of logic required to calculate sine, cosine, and square root. It is derived from an old method for computing certain inverse functions which was once considered for use in software, but then abandoned because of efficiency concerns. However, when reversed and combined with a restoring square root algorithm, a unique new design emerges which performs trigonometric calculations without the use of pre-stored constants or any internal operation more complex than binary subtraction.

The algorithm has been implemented in FPGA using VHDL and is found to be accurate with an error of -2.9% to 2% for square root, an error of -1.4% to .19% for cosine and an error of -6.5% to 1.9% for sine. The implementation of this algorithm requires less hardware than the comparable CORDIC algorithm.

This thesis consists of introduction to both the algorithms. The description of the algorithm and finally a comparative study of the algorithm is been presented in this thesis. The results constitute simulation of VHDL codes of different modules and their synthesis in Xilinx Foundation Series ISE-6.

List Of Figures

Figure No.	Title of Figure	Page No.
2.1	Ripple-carry adder consisting of n full adders	6
2.2	Block diagram of (a) multiplier and (b) divider for n-bit unsigned integers	7
2.3	Numerical example of restoring division	8
2.4	Numerical example of nonrestoring division	10
3.1	Block diagram for cosine	18
3.2	Square root block diagram	20
3.3	Square root and cosine with improved accuracy	23
3.4	Permutations of first two iterations	24
3.5	Optimized square root, sine, and cosine evaluator	25
3.6	Vector rotation	28
3.7	CORDIC iteration	29
3.8	CORDIC architecture	30
4.1	Evolution of FPGAs	36
5.1	Flowchart of Function Generator	44
5.2	Implementation of design flow	45
5.3	Word format	47
5.4	Flowchart of Square Root	50
5.5	State diagram of Square Root	51
5.6	Flowchart of Cosine	52
5.7	State diagram of Cosine	53
6.1(a,b)	I/O view of (a): subtractor_16, (b): subtractor_8	59
6.2	I/O view of square root	60
6.3	Simulation of square root	61
6.4	Splitted detailed view of square root module	62
6.5	Detailed View_1 of square root	63
6.6	Detailed View_2 of square root	64

6.7	Detailed View_3 of square root	65
6.8	I/O view of cosine	66
6.9	Simulation of cosine	67
6.10	Splitted detailed view of cosine module	68
6.11	Detailed View_1 of cosine	70
6.12	Detailed View_2 of cosine	71
6.13	Detailed View_3 of cosine	72
6.14	I/O view of function generator	73
6.15	Simulation of Function Generator	74,75,76
6.16	Splitted detailed view of function generator	77
6.17	Detailed View_1 of function generator	79
6.18	Detailed View_1 of function generator	80
6.19	Elaborated instance E of Function generator	81
6.20	Error in square root calculation	82
6.21	Error in cosine calculation	83
6.22	Error in sine calculation	84
6.23	Comparison with CORDIC	85

List Of Tables

Table No.	Title of Table	Page No.
2.1	Representation of special values	6
6.1	Representation of symbols in square root module	62
6.2	Representation of symbols in square root module	69
6.3	Representation of symbols	70

CHAPTER 1

Introduction

Rapid advances made in VLSI and WSI technologies have led the way to an entirely different approach to computer design for real-time applications, using special-purpose architectures with custom chips. By migrating some of the highly compute-intensive tasks to special-purpose hardware, performance which is typically in the realm of supercomputers, can be obtained at only a fraction of the cost. High-level Computer-Aided Design (CAD) tools for VLSI silicon compilation, allow fast prototyping of custom processors by reducing the tedium of VLSI design. Special-purpose architectures are not burdened by the problems associated with general computers and can map the algorithmic needs of a problem to hardware. Extensive parallelism, pipelining and use of special arithmetic techniques tailored for the specific application lead to designs very different from conventional computers.

Today's world of information interchange revolves around transmission and viewing real-time images. Many of the Digital Signal Processing (DSP) applications try to closely simulate real life images. Speed, clarity, and resemblance to real time objects are some of the many issues to be addressed in order to achieve this goal. Trigonometric function calculation is one of the primary tasks performed in DSP applications.

For a long time microprocessor-based systems have been used to perform this task. Software algorithms used by the processors do not meet the highly demanding needs of all DSP tasks. Using hardware systems to perform these DSP tasks is a competent solution to this problem. FPGAs are often used as coprocessors to perform all the high-speed tasks that cannot be achieved by microprocessors. FPGAs are chosen because they are on-site programmable and are highly suitable for hardware implementations. The software solutions adapted by the microprocessors to implement trigonometric functions are compute intensive. They do not suit hardware platforms because they need complex circuits to perform the mathematical operations. Hence hardware algorithms are adopted for the calculation of trigonometric functions.

1.1 Applications of the trigonometric functions

Digital sine and cosine functions have a large number of applications in digital systems. Since their inception, trigonometric algorithms have been investigated by many researchers and have been employed in software and/or hardware for a variety of applications such as performing:

1. Fourier and related Transforms (FFT/DFT, Discrete Sine/Cosine Transforms, etc.).
2. Householder Transformations, Singular Value Decomposition (SVD) and other Matrix Operations.
3. Filtering and Array Processing.

Some examples of applications:

- The trigonometric function implementation is used to compute the PARK Transformation in control theory for Electric vehicle. An induction motor presents the difficulties of the high degree differential equations. After CLARKE decomposition, the PARK transformation gives the following relation [17]:

  $$\begin{bmatrix} V_\alpha \\ V_\beta \end{bmatrix} = \begin{bmatrix} \cos\theta - \sin\theta \\ \sin\theta \cos\theta \end{bmatrix} * \begin{bmatrix} V_d \\ V_q \end{bmatrix} \quad (1.1)$$

  To compute the equation, we can use the trigonometric algorithm.

- Digital signal processing (DSP) is used in almost all the modern digital hearing aids available today. The processing of the sound signals is carried out in the frequency domain. Fast Fourier Transform (FFT) converts the time domain signal frequency domain and gives a complex number output representing each frequency component in the audible frequency spectrum. The magnitude

estimation block estimates the magnitude of these complex numbers. This estimated magnitude is used as an input to apply various DSP algorithms for sound processing in the hearing aid. Hearing aid is a device that helps in making the normally inaudible sounds audible to the hearing impaired people and thus enables them to maintain contact with the aura1 world. The study shows that around 10% of people in the developed countries suffer from hearing impairment [25].

This magnitude estimation can be done by the trigonometric algorithm.

1.2 Objective of thesis

In this thesis, we explore algorithm for the Trigonometric functions which has an additional functionality of calculating the square root and then implement it in FPGA.

The algorithm is hardware efficient and with more accuracy for some values. The project is divided into two major parts: algorithm comparison and implementation of algorithm on FPGA.

A description is given of the architecture of a digital trigonometric function generator that uses the COSINE algorithm to produce sine and cosine functions. This function generator also produces the square root function.

An full implementation of the function generator using integer arithmetic is written in VHDL. The thesis also presents the simulation and synthesis results obtained for hardware implementation of eight iterations of the COSINE algorithm. Computer aided design (CAD) tools are used for synthesis, simulation and implementation on the FPGAs. A comparative study on the accuracy is been done. Comparison and analysis are done on speed, area and accuracy. Xilinx® Integrated Software Environment (ISE) software is used to do all the synthesis.

4.3 Organization of thesis

This thesis opens with the basic concepts of the computer arithmetic in chapter 2 and the different number representations are discussed.

Chapter 3 deals with the literature survey regarding the trigonometric function computation algorithms and the research work done in the field of algorithms is described. The algorithms compared are CORDIC algorithm and COSINE algorithm which is the reverse of Morrison inverse function algorithm.

Description of FPGA (Spartan) architecture is discussed in chapter 4. Xilinx® Integrated Software Environment (ISE) software is used for synthesis and a brief introduction to it is given in this chapter. VHDL language basics are also discussed in this chapter.

The design and implementation of the function generator with the help of the COSINE algorithm is described in chapter 5. The state diagrams and flow-charts are described in this chapter. This chapter has addressed the work done on the coding of the function generator, cosine algorithm and square root algorithm in VHDL for implementation on an FPGA.

The final results of synthesis and implementation and the error comparison are in chapter 6. Chapter 7 gives the conclusion and future scope.

CHAPTER 2
Basics Of Computer Arithmetic

2.1 Introduction

Computer arithmetic is one of the oldest research topics. Early civilizations used marked bones to memorize numerical values. This approach can be viewed as a combination of read only memories and a unary number system in which addition was very efficient. Around 3300 BC, the first true number systems appeared when Sumerian clay tablets were used to implement a radix-60 number system. A variation of this system is still used today when dealing with minutes and seconds. The basis of our current decimal number system originated during the sixth century in India and was later adopted in Europe during the 12th century. As mankind continues to advance and new applications and technologies emerge, innovative research in computer arithmetic is essential. Important applications, including multimedia processing, cryptography, and computer graphics, all need novel arithmetic algorithms and hardware designs to satisfy stringent area, delay, and power requirements.

Furthermore, advances in VLSI technology and innovations in tools for electronic design automation offer new opportunities for implementing complex algorithms in hardware [8]. The continuing evolution of hardware speed and expanding storage in cache and memory provides that the choice of fundamental arithmetic algorithms and numeric representations within the arithmetic logic unit must be continually addressed to obtain competitive arithmetic unit implementations.
Although computer arithmetic is viewed as a specialized part of CPU design, it is very important part [13].

2.2 Basic Techniques Of Integer Arithmetic

Adders are usually implemented by combining multiple copies of simple components. The natural components for addition are half adders and full adders. The half adder takes

two bits a and b as input and produces a sum bit s and a carry bit c_{out} as output. Mathematically, s=(a+b)mod 2,and c_{out}=floor((a+b)/2).the half adder is also called (2,2) adder, since it takes two inputs and produces two outputs. The full adder is a (3,2) adder and is defined by s= (a+b+c) mod 2, c_{out} = floor((a+b+c)/2).

The principal problem in constructing an adder for n-bit numbers out of smaller pieces is propagating the carries from one piece to the next. The most obvious way to solve this is with a ripple-carry adder, consisting of n full adders. The inputs to the adder are $a_{n-1}a_{n-2}\ldots a_0$ and $b_{n-1}b_{n-2}\ldots b_0$, where $a_{n-1}a_{n-2}\ldots a_0$ represents the number $a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\ldots\ldots+a_0$. The c_{i+1} output of the ith adder is fed into the input of the next adder (the (i+1)-th adder) with the lower-order carry-in c0 set to 0. Typical values of n are 32 for integer arithmetic and 53 for double-precision floating point. The ripple-carry adder is the slowest adder, but also the cheapest. It can be built with only n simple cells, connected in a simple, regular way. In technologies like CMOS, even though ripple adders take time O(n), the constant factor is very small. In such cases short ripple adders are often used as building blocks in larger adders [15, 19].

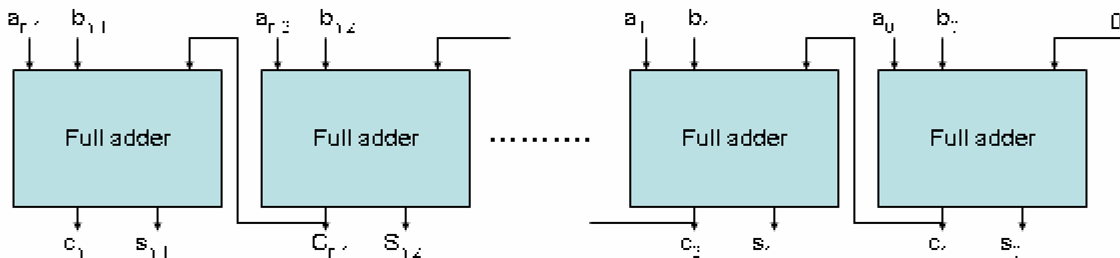


Figure 2.1. Ripple-carry adder consisting of n full adders

2.3 Radix-2 Multiplication And Division

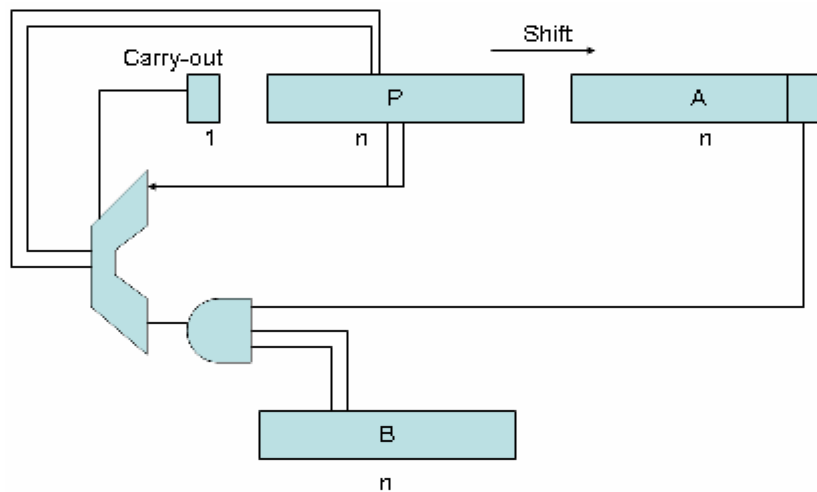
The simplest multiplier computes the product of two unsigned numbers, one bit at a time. The numbers to be multiplied are $a_{n-1}a_{n-2}\ldots a_0$ and $b_{n-1}b_{n-2}\ldots b_0$ and they are placed in registers A and B, respectively. Register P is initially 0. Each multiply step has two parts.

Multiply step

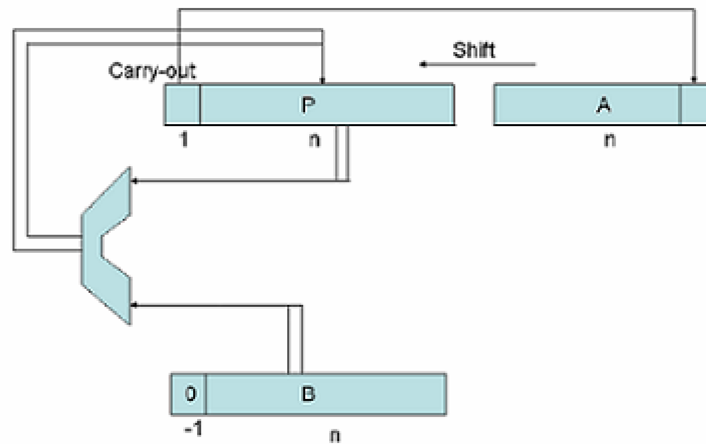
(i) If the least significant bit of register A is 1, then register B, containing $b_{n-1}b_{n-2}....b_0$, is added to P; otherwise 00….00 is added to P. the sum is placed back into P.

(ii) Registers P and A are shifted right, with carry-out of the sum being moved into the high-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.

After n steps, the product appears in the registers P and A, with A holding the lower-order bits [29].



(a)



(b)

Figure 2.2. Block diagram of (a) multiplier and (b) divider for n-bit unsigned integers.

The simplest divider also operates on unsigned numbers and produces the quotient bits one at a time. A hardware divider is shown in the Figure 2.2. To compute a/b, put a in the A register, b in the B register, 0 in the P register, and then perform n divide steps. Each divide step consists of four parts:

Divide step

(i) Shift the register pair (P,A) one bit left.

(ii) Subtract the contents of register B (which is $b_{n-1}b_{n-2}\ldots b_0$) from the register P, putting the result back into P.

(iii) If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.

(iv) If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

After repeating this process n times, the register A will contain the quotient, and the P register will contain the remainder. The algorithm is binary version of the paper-and-pencil method; a numerical example is illustrated in Figure 2.3.

P	A	
00000	1110	Divide 14 = 1110_2 by 3 = 11_2. B always contains 0011_2.
00001	110	Step 1(i): shift
-00011		Step 1(ii): subtract
-00010	1100	Step 1(iii): result is negative, set quotient bit to 0.
00001	1100	Step 1(iv): restore
00011	100	Step 2(i): shift
-00011		Step 2(ii): subtract
00000	1001	Step 2(iii): result is nonnegative, set quotient bit to 1.
00001	001	Step 3(i): shift
-00011		Step 3(ii): subtract
-00010	0010	Step 3(iii): result is negative, set quotient bit to 0.
00001	0010	Step 3(iv): restore
00010	010	Step 4(i): shift

-00011		Step 4(ii): subtract
-00001	0100	Step 4(iii): result is negative, set quotient bit to 0.
00010	0100	Step 4(iv): restore. The quotient bit is 0100_2 and the remainder is 00010_2

Figure 2.3. Numerical example of restoring division

Notice that the two block diagrams in Figure 2.2 are very similar. The main difference is that the register pair (P,A) shifts right when multiplying and left when dividing. By allowing these registers to shift bidirectionally, the same hardware can be shared between multiplication and division.

The division algorithm illustrated in the Figure 2.3 is called restoring, because if subtraction by B yields a negative result, the P register is restored by adding B back in. the restoring algorithm has a variant that skips the restoring step and instead works with the resulting negative numbers. Each step of the nonrestoring algorithm has three parts:

Nonrestoring Divide step

If P is negative,

(i-a) Shift the register pair (P,A) one bit left.

(ii-a) Add the contents of register B (which is $b_{n-1}b_{n-2}\ldots b_0$) from the register P.

Else,

(i-b) Shift the register pair (P,A) one bit left.

(ii-b) Subtract the contents of register B (which is $b_{n-1}b_{n-2}\ldots b_0$) from the register P.

(iii) If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

After repeating this n times, the quotient is in A. if P is nonnegative, it is the remainder. Otherwise, it needs to be restored (i.e., add B), and then it will be the remainder. The explanation for why the nonrestoring algorithm works is this. Let r_k be the contents of the (P,A) register pair at step k, ignoring the quotient bits (which are simply sharing the unused bits of register A). In the Figure 1.3, initially A contains 14, so r_0=14. At the end of the first step, r_1 =28, and so on. In the restoring algorithm, part (i) computes $2r_k$ and then part (ii) $2r_k - 2^n b$ ($2^n b$ since b is subtracted from the left half). If $2r_k - 2^n b \geq 0$, both algorithms end the step with identical values in (P,A). If $2r_k - 2^n b < 0$, then the restoring

algorithm restores this to $2r_k$, and the next step begins by computing $r_{res} = 2(2r_k) - 2^n b$. In the nonrestoring algorithm, $2r_k - 2^n b$ is kept as a negative number, and in the next step $r_{nonres} = 2(2r_k - 2^n b) + 2^n b = r_{res}$. Thus (P,A) has the same bits in both algorithms.

If a and b are unsigned n-bit numbers, hence in the range $0 \le a,b \le 2^n - 1$, then the multiplier in figure will work if register P is n bits long. However, for division, P must be extended to n+1 bits in order to detect the sign of P. Thus the adder must also have n+1 bits [9, 13].

P	A	
00000	1110	Divide $14 = 1110_2$ by $3 = 11_2$. B always contains 0011_2.
00001	110	Step 1(i-b): shift
+11101		Step 1(ii-b): subtract b (add two's complement).
11110	1100	Step 1(iii): P is negative, set quotient bit to 0.
11101	100	Step 2(i-a): shift
+00011		Step 2(ii-a): add b
00000	1001	Step 2(iii): P is nonnegative, set quotient bit to 1.
00001	001	Step 3(i-b): shift
+11101		Step 3(ii-b): subtract b
11110	0010	Step 3(iii): P is negative, set quotient bit to 0.
00010	010	Step 4(i-a): shift
+00011		Step 4(ii-a): add b
11111	0100	Step 4(iii): P is negative, set quotient bit to 0.
+00011		Remainder is negative, so do final restore step
00010		The quotient bit is 0100_2 and the remainder is 00010_2

Figure2.4. Numerical example of nonrestoring division

2.4 Signed Numbers

There are four methods commonly used to represent signed n-bit numbers:

(i) Sign magnitude
(ii) Two's complement
(iii) One's complement
(iv) Biased

A useful formula for the value of a two's complement number $a_{n-1}a_{n-2}\ldots a_0$ is

$$-a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\ldots\ldots+a_1 2^1+a_0 \qquad (2.1)$$

In a biased system, all numbers are first represented by first adding them to the bias, and then encoding this sum as an ordinary unsigned number. Thus a negative number k can be encoded as long as k + bias ≥ 0.typical value for the bias is 2_{n-1}.
Overflow occurs when the result of the operation does not fit in the representation being used. For example, if signed numbers are being represented using 4 bits, then $6 = 0110_2$ and $11 = 1011_2$. Their sum (17) overflows because its binary equivalent (10001_2) doesn't fit into 4 bits. For unsigned numbers, detecting overflow is easy; it occurs exactly when there is carry-out of the most-significant bit. For two's complement, things are trickier: Overflow occurs exactly when the carry into of the high order bit is different from the (to be discarded) carry-out of the high order bit.
Negating a two's complement number involves complementing each bit and then adding 1. For instance, to negate 0011_2, complement it to get 1100_2 and then add 1 to get 1101_2. Thus, to implement a-b using an adder, simply feed a and b' (where b' is the number obtained by complementing each bit of b) into the adder and set the low-order, carry-in bit to1. This explains why the rightmost adder in ripple-carry adder is a full adder [9, 13].

2.5 Signed Multiplication

When we are multiplying unsigned numbers using the hardware as shown in Figure 2.2(a). If B is potentially negative but A is nonnegative, the only change needed to convert the unsigned multiplication algorithm into two's complement one is to ensure that when P is shifted, it is shifted arithmetically; that is, the bit shifted into the high-order bit of P should be the sign bit of P (rather than carry-out from the addition). Note that our n-bit adder will now be adding n-bit two's complement numbers between $-_{2n-1}$ and 2_{n-1}-1[13].

2.5.1 Booth Algorithm

Next, suppose a is negative. The method for handling this case is called Booth Recoding. Booth recoding is a very basic technique in computer arithmetic. Booth recoding "recodes" the number 7 as 8 -1 $=1000_2 - 0001_2 = 100\bar{1}$, where $\bar{1}$ represents -1. This gives an alternate way to compute a * b; namely, successively subtract B, add 0, add 0, and add B. This is more complicated than the unsigned algorithm. If the initial content of A is $a_{n-1}....a_0$, then at the ith multiply step, the low-order bit of the register A is a_i, and step (i) in the multiplication algorithm becomes:

I. If $a_i = 0$ and $a_{i-1} = 0$, then add 0 to P.

II. If $a_i = 0$ and $a_{i-1} = 1$, then add B to P.

III. If $a_i = 1$ and $a_{i-1} = 0$, then subtract B to P.

IV. If $a_i = 1$ and $a_{i-1} = 1$, then add 0 to P.

For the first step, when I = 0, take a_{i-1} to be 0.

The four cases above can be restated as saying that in the ith step you should add $(a_{i-1}-a_i)B$ to P. With this observation, it is easy to verify that these rules work, because the result of all the additions is

$$\sum b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+......+ a_1 2^1+a_0) + ba_{-1} \qquad (2.2)$$

Using equation (2.1) together with $a_{-1} = 0$, the right-hand side is seen to be the value of b*a as a two's complement number [31].

2.6 Floating Point Numbers

Many applications require numbers that aren't integers. There are a number of ways that non-integers can be represented. These are:

(a) Fixed Point; that is use simple arithmetic and simply imagine the binary point somewhere other than just to the right of the least-significant digit.

(b) Logarithmic Number System; this is storing the logarithm of a number and doing multiplication by adding the logarithms, or using a pair of integers (a,b) to represent the fraction a/b.

(c) Floating Point; in this system, a computer word is divided into two parts, an exponent and a significand.

The computer industry is rapidly converging on the format specified by IEEE standard 754-1985. IEEE arithmetic differs from many previous arithmetics in the following major ways:

1. When rounding a “halfway” result to the nearest floating-point number, it picks the one that is even.
2. It includes the special values NaN,∞ and -∞.
3. It uses denormal numbers to represent the result of computations whose value is less than $1.0 * 2^{Emin}$.
4. It rounds to nearest by default, but it also has three other rounding modes.
5. It has sophisticated facilities for handling exceptions.

When operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating-point number, such halfway cases are rounded to the number whose lower-order digit is even. The standard actually has four rounding modes. The default is round to nearest, which rounds ties to an even number as just explained. The other modes are round toward 0, round toward ∞, round toward -∞ [2, 30].

2.6.1 Special Values And Denormals

Probably the most notable feature of the standard is that by default a computation continues in the face of exceptional conditions, such as dividing by 0 or taking the square root of a negative number. For example, the result of taking the square root of a negative number is a NaN (Not a Number), a bit pattern that does not represent an ordinary number. In IEEE arithmetic, if the input to an operation is a NaN, the output is NaN (e.e., 3 + NaN = NaN). Because of this rule, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks.

While the result of $\sqrt{-1}$ is a NaN, the result of 1/0 is not a NaN, but $+\infty$, which is another special value. The standard defines arithmetic on infinities (there is both $+\infty$ and $-\infty$) using rules such as $1/\infty = 0$. The final kind of special values in the standard are denormal numbers. In many floating-point systems, if Emin is the smallest exponent, a number less than $1.0 * 2^{Emin}$ cannoy be represented, and a floating-point operation that results in a number less than this is simply flushed to 0. In the IEEE standard, on the other hand numbers less than $1.0 * 2^{Emin}$ are represented using significands less than 1. this is called gradual underflow. Thus, as numbers decrease in magnitude below 2^{Emin}, they gradually lose their significance and sre only represented by 0 when their significance has been shifted out. Denormals make dealing with small numbers more predictable by maintaining familiar properties such as $x = y \leftrightarrow x-y = 0$. for example, in a flush-to-zero system (again in base 10 with four significant digits), if $x = 1.256 * 10^{Emin}$ and $y = 1.234 * 10^{Emin}$, then $x - y = 0.022 * 10^{Emin}$, which flushes to zero. So even though $x \neq y$, the computed value of $x - y = 0$. this never happens with gradual underflow. In this example, $x - y = 0.022 * 10^{Emin}$ is a denormal number, and so the computation of $x - y$ is exact [13, 15].

2.6.2 Representation Of Floating Point Numbers

Single-precision numbers are stored in 32 bits: 1 for sign, 8 for the exponent, and 23 for the fraction. The exponent is a signed number represented using the bias method with a bias of 127. the fraction represents a number less than 1, but the significand of the floating-point number is 1 plus the fraction part. In other words, if e is the biased exponent (value of exponent field) and f is the value of the fraction field, the number being represented is

$$1.f * 2^{e-127} \tag{2.3}$$

When performing arithmetic on IEEE format numbers, the fraction part is usually unpacked, which is to say the implicit one is made explicit [13].

2.6.3 Precisions

The standard specifies four precisions:

a) single

b) single extended

c) double

d) double extended

Implementations are not required to have all the four precisions, but are encouraged to support either the combination of single and single extended or all of single, double, and double extended. Because of the widespread use of double precision in scientific computing, double precision is almost always implemented. Thus the computer usually only has to decide whether to support double extended and, if so, how many bits it should have. The exponent for single precision range from -126 to 127; accordingly, the biased exponent ranges from 1 to 254. the biased exponents of 0 and 255 are used to represent special values.this is summarized in the figure. When the biased exponent is 255, a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the biased exponent and the fraction field are 0, then the number represented is 0. Because of the implicit leading 1, ordinary numbers always have a significand greater than or equal to 1. Thus, a special convention such as this is required to represent 0. Denormalized numbers are implemented by having a word with a zero exponent field represent the number

$$0.f * 2^{Emin} \tag{2.4}$$

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that 0 is represented by a word of all 0's. the downside of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum[13].

Table 2.1. Representation of special values.

Exponent	Fraction	Represents
$e = E_{min} - 1$	$f = 0$	± 0
$e = E_{min} - 1$	$f \neq 0$	$0.f * 2^{E_{min}}$
$E_{min} \leq e \leq E_{max}$	-	$1.f * 2^{e}$
$e = E_{max} + 1$	$f = 0$	$\pm\infty$
$e = E_{max} + 1$	$f \neq 0$	NaN

CHAPTER 3

Trigonometric Algorithms

3.1 Introduction

In the mid-1950's Morrison [4] presented a general technique for generating algorithms to compute the inverses of certain functions. The principal requirement of this technique is the existence of a simple function G such that F(2.x) = G (F (x) . For example, if we let F (x)= cos(πx) then G(x) = ($2x^2 - 1$) since cos(πx)= $\cos^2(\pi x)$-1. Morrison's method then uses the values produced by iterations of G(x) to calculate $x=F^{-1}(y) = \mathrm{Cos}^{-1}(y)/ \pi$.
The resulting algorithm is shown below. Proofs for the procedure are given in [4].

Algorithm [ARCCOSINE]
[Input]
    Y : $-1 \le Y \le 1$
[Output]
    X_n : the first n bits of the binary representation of $\mathrm{Cos}^{-1}(Y)/\pi$ (the principal arccosine)

[Algorithm]
Step 1: $X_0 := 0$
    $Z_0 := Y$

Step 2: for i := 1 to n do
    begin
        if $Z_{i-1} \ge 0$ then
        begin
            $X_i := X_{i-1}$
            $Z_i := 2(Z_{i-1})^2 - 1$
        end
        else

begin

$X_i := X_{i-1} + 2^{-1}$

$Z_i := 1 - 2(Z_{i-1})^2$

end

end

This algorithm, though unique in its simplicity, was dismissed by some who correctly asserted that the error in X_n may be abnormally large for certain values of Y, and that simple linear convergence with a multiplication in each iteration results in slow operation when compared with other software methods. What is more, since there is no simple function G' to map $Cos^{-1}(2y)= G'(Cos^{-1}(y))$, the technique cannot be used to calculate y = Cos (x) directly. Fortunately, these concerns do not hold in the hardware implementation presented below.

Morrison's method was later extended to other functions by Wensley, but then largely abandoned by the literature, though it is still occasionally mentioned.

3.1.1 COSINE algorithm

The arccosine algorithm can be reversed to produce a new algorithm for cosine using the bitwise Exclusive-OR function as follows:

Algorithm [COSINE]

[Input]

X_n: a binary number such that

$X_n = 0.\, b_1\, b_2\, b_3\, b_4\, b_5 \ldots\ldots\ldots\ldots\ldots\, b_{n-1}\, b_n0000\ldots\ldots\ldots$

[Output]

Y : $Y = \cos(\pi X_n)$(exactly)

[Algorithm]

Step 1: $Z_{n+1} := 1.0$

$b_0 := b_{n+1} := 0$

Step 2: for i := n downto 0 do

begin

Si := { +1 if bi xor bi+1 = 0

{ -1 if bi xor bi+1 = 1

Zi := Si √((1 + Zi+1) / 2)

end

Step 3: Y := Z_0

As an example, find cos(0.6875 π) with n= 4.

First, X_4=0.6875=0.1 0 1 1 0 0 0…..

Then [S_0,……,S_4] = [-1, -1, -l, +l, -1], and so

$Z_5 = 1.0$

$Z_4 = -1.0$

$Z_3 = +0.0$

$Z_2 = -0.707\ 106$

$Z_1 = -0.382\ 083$

$Z_0 = -0.555\ 570 = Y = \cos(0.6875\ \pi)$.

Clearly the most difficult step in this process is taking the square root at each iteration. But if we consider a system which already requires hardware for computing square roots, then adding this extra trigonometric functionality requires very little additional circuitry (as shown in Figure 3.1).

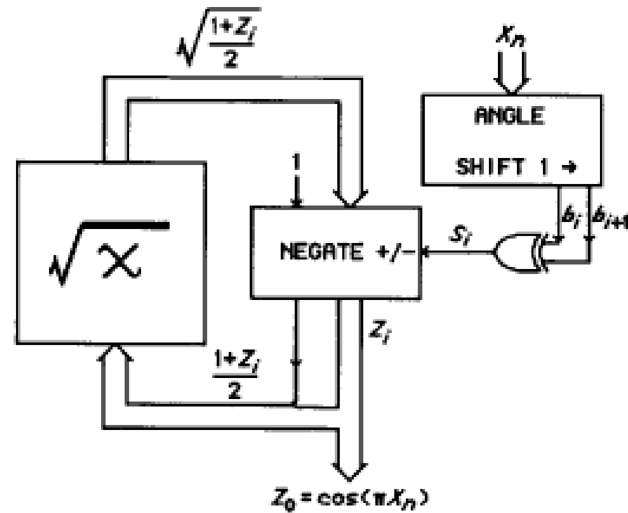


Figure 3.1. Block diagram for cosine

The usefulness of this design now hinges on finding a fast and simple root evaluator, and on resolving accuracy concerns.

3.1.2 Square Root

The algorithm for evaluating the square root is a binary version of the well known but often overlooked longhand method. In decimal this is a rather tedious process which somewhat resembles restoring division, except that two digits are taken from the radicand for each single digit produced in the result. Fortunately, the binary version of the algorithm is very straightforward since each new bit in the result (0 or 1) is chosen by performing a simple comparison. The full algorithm is given below, and an example showing the longhand binary process for calculating √10101001 (√169 = 13) appears in the example [7, 23].

Algorithm [ROOT]

[Input]

P : the radicand ($0 \le P < 1$)

[Output]

Q_n : the first n bits of the value $2^n \sqrt{P}$ in binary

[Algorithm]

Step 1: R_0 := P (R, is the partial remainder)

Q_0 := 0

Step 2: for i := 1 to n do

```
begin
    T_i := 4R_{i-1} - (4Q_{i-1} + 1)
    if T_i ≥ 0 then
        begin
            R_i := T_i
            Q_i := 2Q_{i-1} + 1
        end
    else
        begin
            R_i := 4R_{i-1}
            Q_i := 2Q_{i-1}
        end
end
```

Longhand binary square root:

Simply append 01 to the current partial result and subtract it from the current partial remainder.

```
√1 0 1 0 1 0 0 1. 0 0....          START-RESULT = O
- 0 1
  0 1 1 0                          OK - RESULT = 1
   - 1 0 1
     0 0 1 1 0                     OK - RESULT = 1 1
      - 1 1 0 1
       *negative*                  REJECT - RESULT = 1 1 0
        0 1 1 0 0 1
        - 1 1 0 0 1
          0 0 0 0 0 0 0            OK - RESULT = 1 1 0 1
```

It has also been implemented in assembly language and as a parallel TTL gate array. In addition, it is a member of the class of algorithms originally described by Morrison and Wensley, and it can be grouped with a class of "direct methods" which has been found to be better suited to realization in hardware than is the traditional method. This process can be replicated in hardware using just three shift registers and a subtractor (Figure 3.2) [23].

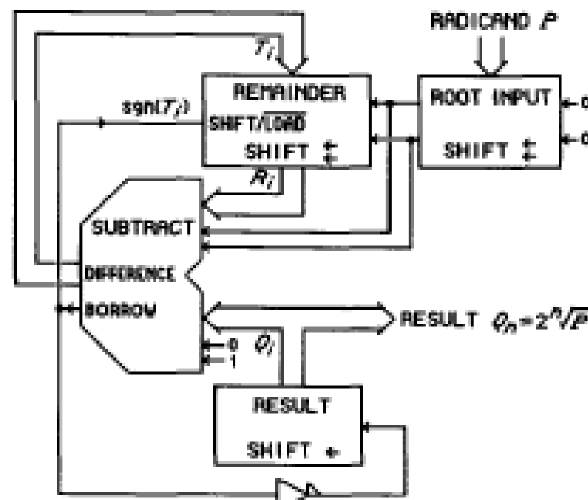


Figure 3.2. Square root block diagram.

3.1.3 Accuracy of square root

The square root circuit presented here exhibits no inaccuracy, in that it generates the first n bits of the true result, where n is limited by register and subtractor widths. (In this design the subtractor must be n + 2 bits, the remainder n + 1 bits, and the result register n bits wide.)

As the square root diagram of Fig. 2.2 implies, two bits of the radicand are shifted out of the input register for every one bit shifted into the result register. Thus, the input word is exhausted halfway through the process, and the remaining n bits which fill out the radicand are zeros. We can take advantage of this fact by doubling the size of the input register so that an n-bit root is now produced from a 2n-bit radicand with no speed penalty!

Extending the precision of the output of the square root is even easier, though less apparent. In the critical cases where values approach 0.11 11 . . . the needed extra bits of precision are not actually lost. In fact, they are immediately available in the remainder register!

As demonstrated next the closer 0 is to 1.0, the more accurate the remainder can be in completing a double length output word. (In other cases the remainder does not make an accurate extension of the result, though it is somewhat better than simply appending zero bits).

Let P be the radicand with up to 2n bits of precision ($0 \leq P < 1$). The algorithm produces result Q_n and remainder D of size n and n + l bits, respectively, where $Q_n^2 + D = 2^{2n} P$ so that $Q_n \rightarrow 2^n \sqrt{P}$.

The accuracy of Q_n is such that $Q_n^2 \leq 2^{2n} P < (Q_n + 1)^2$.

Now, let $P \rightarrow 1.0$ and $n \geq 4$:

$$2^{2n} P = Q_n^2 (1 + \frac{D}{Q_n^2}), \text{ so } 2^n \sqrt{P} = Q_n \sqrt{1 + \frac{D}{Q_n^2}}$$

$$\cong Q_n (1 + \frac{D}{2Q_n^2})$$

Hence, $Q_n + 2^{-n-1} D \cong 2^n \sqrt{P}$ when $P \cong 1.0$.

Example

With n = 8 let P = 0.11111001000……… (= 249/256)

then $2^8 \sqrt{P}$ = 11111100.01111001 (= 252.4726)

The algorithm gives Q_8 = 11111100 (= 252)

and $D = 2^{16}P - Q_8$ = 011110000 (= 240)

so $Q_8 + 2^{-9}D$ = 11111100.01111000 ($\approx 2^8 \sqrt{P}$)

Now moving the binary point gives

$\sqrt{249/256}$ = 0.111111000111100 ($\approx$ 0.9862)

accurate to 15 bits![23]

3.1.4 Accuracy of COSINE algorihm

The square root circuit presented here exhibits no inaccuracy, in that it generates the first n bits of the true result, where n is limited by register and subtractor widths. (In this design the subtractor must be n + 2 bits, the remainder n + 1 bits, and the result register n bits wide.)

Unfortunately, the cosine algorithm is not as well behaved. The problem occurs when the angle is such that there are many Si = +1 in a row. This causes $Z_i = S_i \sqrt{(1 + Z_{i+1})/2}$ to rapidly approach 0.1111.. . = 1.0, after which point any further iterations are likely to produce meaningless values. In such cases, the internal precision may need to be as high as 2n bits for an n-bit angle and cosine. Therefore, for the circuit of Figure 3.1 to behave consistently, the square root will have to accept and produce up to twice the number of bits originally anticipated. Fortunately, the design we have chosen admits a simple and effective solution to this problem as explained in the above section.

Figure 3.3 shows how the cosine interfaces with the square root to take advantage of the increased accuracy.

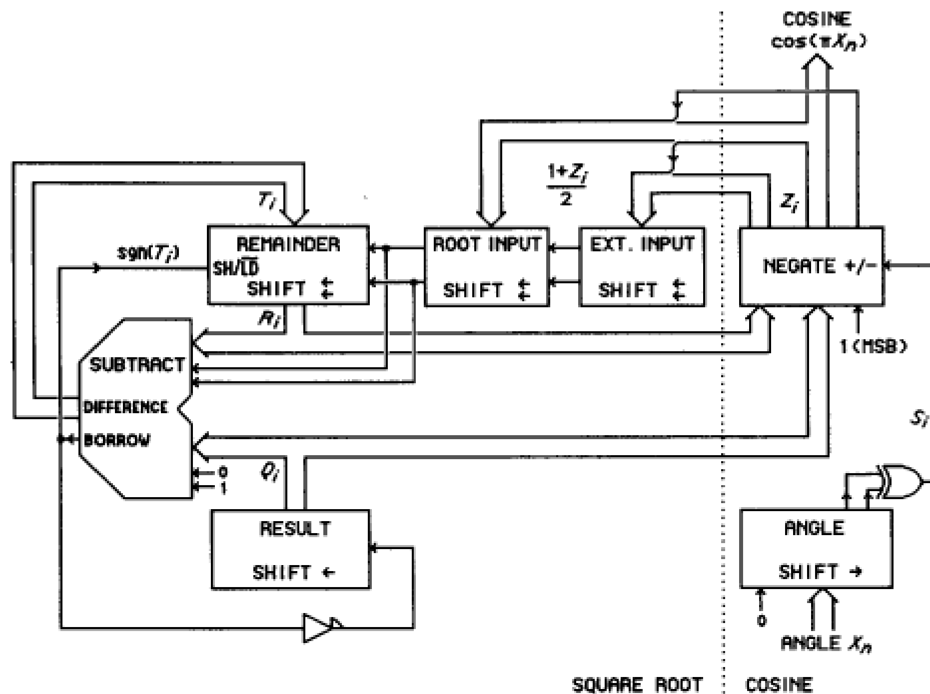


Figure 3.3. Square root and cosine with improved accuracy

3.1.5 Optimization

Several additional improvements may now be made. First, rather than using a parallel negator as in Figures 3.1 and 3.3, two exclusive-or gates placed at the serial outputs of the square root input register perform an effective one's complement negation. Unfortunately, this leaves only the absolute value of the result available once the iterations are complete. But we can modify the algorithm to calculate: $Y = \cos(\pi X_n/2)$ where $0 \leq X_n < 1$ so that $0 \leq Y < 1$ also.

Then range reduction and sign assignment are left to software or external circuitry rather than occupying bits now used for precision. This modification is effected by performing one additional cosine iteration, since $\cos(\pi X_n/2) =) \sqrt{((1 + \cos(\pi X_n)) / 2)}$. A change in the initialization procedure can reduce the total number of required steps. The four possible permutations of the first two iterations of the COSINE algorithm are as follows:

S_n	S_{n+1}	Z_n	Z_{n-1}	Z_{n-2}
+1	+1	$+\sqrt{1}$	$+\sqrt{1}$	$S_{n-2}\sqrt{1.0}$
+1	−1	$+\sqrt{1}$	$-\sqrt{1}$	$S_{n-2}\sqrt{0.0}$
−1	+1	$-\sqrt{1}$	$+\sqrt{0}$	$S_{n-2}\sqrt{0.5}$
−1	−1	$-\sqrt{1}$	$-\sqrt{0}$	$S_{n-2}\sqrt{0.5}$

Figure 3.4. Permutations of first two iterations

Rather than waste cycles on trivial values, the numbers 1.0 (=0.1111...), 0.0 (= 0.0000...), and 0.5(= 0.1000... or 0.0111...) could be injected directly into the square root by modifying the initial value of S_i (SIGN) and the most significant bit (MSB) of the cosine series initialization value during the loading cycle, according to the lowest order bits of the angle input X_n.

It is also possible to calculate $\sin(\pi X_n/2)$. At the beginning of the last cosine iteration, the square root input registers contain $\cos^2(\pi X_n/2)$. If his is inverted to make the one's complement $1 - \cos^2(\pi X_n/2)$, then the final result is $\sin(\pi X_n/2)= \sqrt{1 - \cos^2(\pi X_n/2)}$.

Replacing the leading zero of the angle input with a SINE/(COSINE) signal has the desired effect, controlling the polarity of the final SIGN as it comes out of the exclusive-OR gate on the angle register.

Finally, it is often desirable to compute both sine and cosine simultaneously, as when preparing for tan(α) = sin(α)/ cos(α). Simply adding two inverters in a feedback loop around the root input registers allows the cofunction to be generated in a single extra step. This is due to the fact that the final radicand $\cos^2(\pi X_n/2)$ is replaced with $(1 - \cos^2(\pi X_n/2)) = \sin^2(\pi X_n/2)$ (or $\sin^2(\pi X_n/2)$) with $1- \sin^2(\pi X_n/2) = \cos^2(\pi X_n/2)$) as these values shift out and around. Then executing a square root once more without reloading the radicand produces the cofunction very quickly.

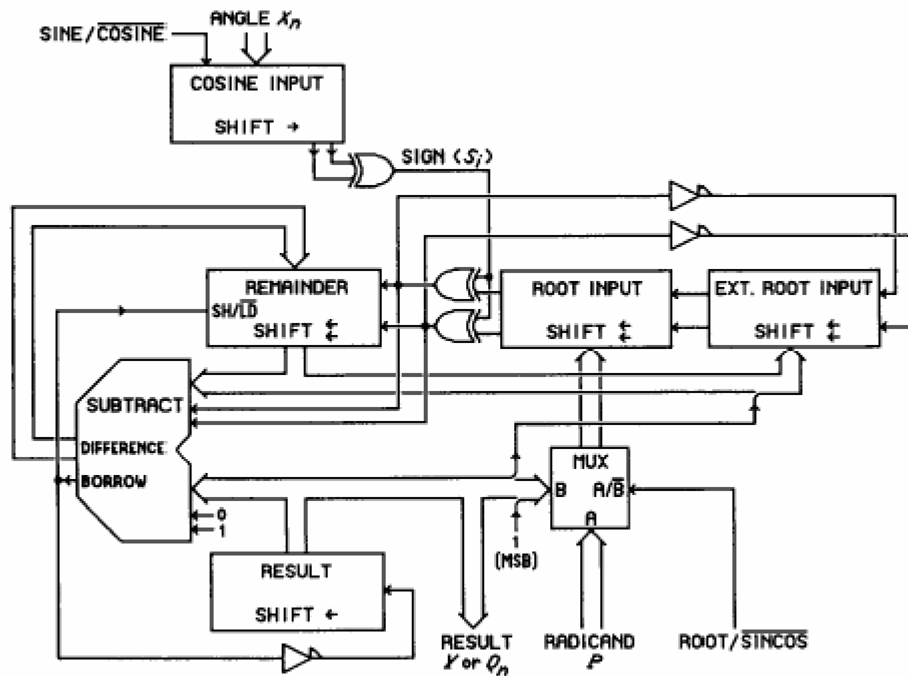


Figure 3.5. Optimized square root, sine, and cosine evaluator

These optimizations are brought together in Figure 3.5. Further implementation details can be found in [24].

3.1.6 Benchmarks

The total error is at most 1.3 in 2^{16}, and the square root exhibits no error beyond word size restrictions. The CORDIC code is built with similar technology and is useful for comparison. Though originally designed for coordinate rotation, it can be modified to calculate sine and cosine by rotating the unit vector (1,0). Then the ROM's used to divide the CORDIC magnification factor K out of the result become unnecessary if the vector inputs are fixed at (l/K,0). Such a circuit would compute sine and cosine simultaneously using 12-bit words.

Square root evaluation is not supported by this circuit. If the precision of the new root-based design were also reduced to 12 bits, its computation times would fall to 6.3 µs for either the sine or cosine (2+12*(1+12)=158 cycles) and an additional 520 ns for the cofunction (1+12=13 cycles). Note that the change in speed with word size is somewhat similar for both implementations ($O(n^2)$) .
(The CORDIC method requires an increasing number of shifts in each step.)

Based on these limited comparisons, the CORDIC circuit appears to be about 50% faster for sine and cosine, but does not perform square roots, and yet requires substantially more hardware, and is a bit less accurate [23].

3.2 CORDIC algorithm

CORDIC is an acronym for COrdinate Rotation Digital Computer. It is a class of shift-add algorithms for rotating vectors in a plane, which is usually used for the calculation of trigonometric functions, multiplication, division and conversion between binary and mixed radix number systems of DSP applications, such as Fourier Transform. [14, 22]. Jack E. Volder's algorithm [12] is derived from the general equations for vector rotation.

The CORDIC computing technique was developed especially for use in real-time digital computer where the majority of the computation involved the discontinuous, programmed solution of the trigonometric relationships of navigation equations and a

high solution rate for the trigonometric relationships of coordinate transformations. A prototype computer, CORDIC I, based on this computing technique, has been designed and constructed at Convair, Forth Worth. Although CORDIC I may be classified as an entire-transfer computer, its design is not based on the conventional “pencil and paper” computing technique general-purpose computers.

3.2.1 Functional description

For the sake of simplicity, the trigonometric operations in the CORDIC computer can be functionally described as the digital equivalent of an analog resolver. Similar to the operation of such a resolver, there are two computing modes, ROTATION and VECTORING. In the ROTATION mode, the coordinate components of a vector and an angle of rotation are given and the coordinate components of the original vector, after rotation through the given angle, are computed. In the second mode, VECTORING, the coordinate components of a vector are given and the magnitude and angular argument of the original vector are computed. Similarly, as in the case of resolvers, the computing device of ROTATION plus feedback is employed in the VECTORING mode. The original coordinates are rotated until the angular agument is zero, so that the total amount of rotation required is the negative of the original argument, in which case the value of the X-component is equal to the magnitude of the original vector.

In essence, the basic technique used in both the ROTATION and VECTORING modes in CORDIC is a step-by-step sequence of pseudo rotations which result in an over-all rotation through a given angle (ROTATION) or result in a final angular argument of zero (VECTORING).

It is necessary that the angular increments of rotation be computed in a decreasing order. There are several permissible values which may be chosen for the angular magnitude of the first rotation step. The magnitude actually chosen for the first increment is 90°. The expression for a set of coordinate components, Y_1 and X_1, rotated through plus or minus 90° is simply:

$$Y_2 = \pm X_1 = R_1 \cos(\theta_1 \pm 90°) \qquad (3.1)$$

$$X_2 = \pm(-Y_1) = R_1 \cos(\theta_1 \pm 90°) \qquad (3.2)$$

The first step is unique in that a perfect rotation step is performed. The rest of the computing steps can be clarified by examining the relationships, involved in a typical rotation step. In this discussion, the quantity i is equal to the number of the particular step under consideration [22]. The components, Y_i and X_i, are associated with the ith step and describe a vector of magnitude R_i at an angle θ_i from the origin according to the relationship:

$$Y_i = R_i \sin\theta_i \qquad (3.3)$$

$$X_i = R_i \cos\theta_i \qquad (3.4)$$

If a vector V with components (x, y) is to be rotated through an angle φ , then a new vector V ' with components (x', y') is formed by:

$$V' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x.\cos(\phi) - y.\sin(\phi) \\ y.\cos(\phi) + x.\sin(\phi) \end{bmatrix} \qquad (3.5)$$

The result of the rotation is shown in Figure 3.6:

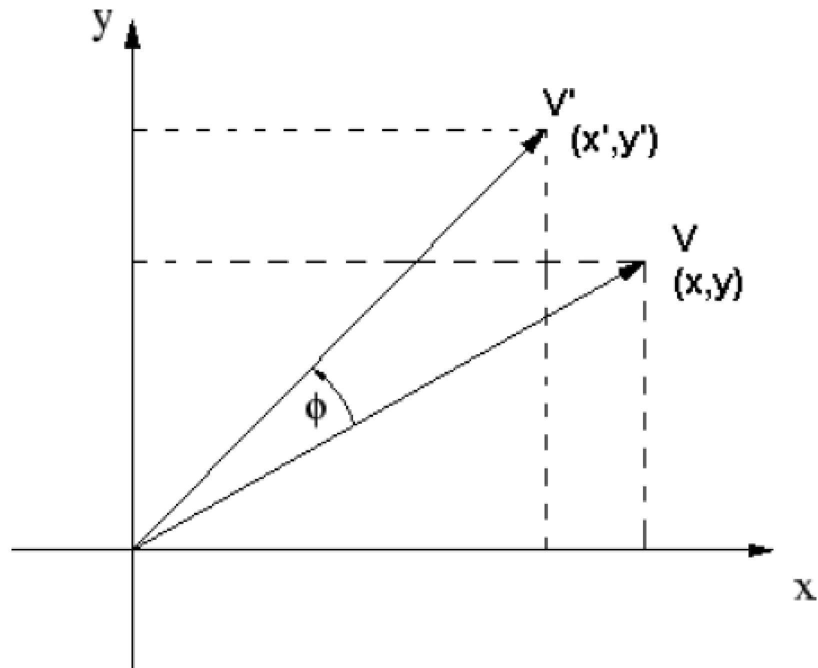


Figure 3.6. Vector rotation

The algorithm can be rearranged as [28]:

$$x' = x.\cos(\phi) - y.\sin(\phi) = \cos(\phi)(x - y.\tan(\phi)) \tag{3.6}$$

$$y' = y.\cos(\phi) + x.\sin(\phi) = \cos(\phi)(y + x.\tan(\phi)) \tag{3.7}$$

Now, if we break the angel into smaller and smaller pieces, the equation above becomes an iteration operation. By this way, the x' and y' of any arbitrary vector can be calculated by the iterations from an original vector containing known x and y values, such as 0 degree. The iteration is shown as Figure 3.7:

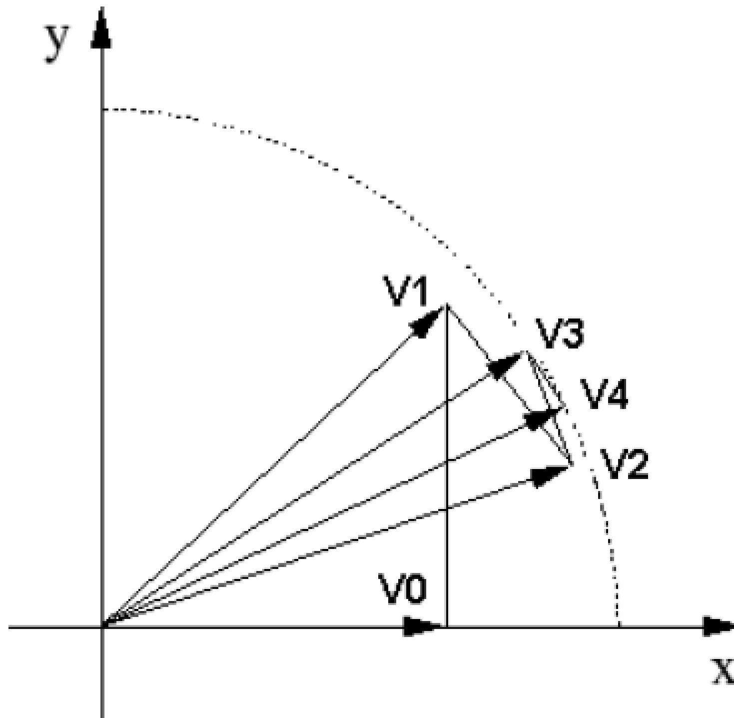


Figure 3.7. CORDIC iteration

The tangent term can be reduced by restricting the tangent terms to: tan(φ) = $\pm 2^{-i}$. The expressions above can be rewritten as:

$$x_{i+1} = K_i \left[x_i \mp y_i . 2^{-i} \right] \quad (3.8)$$

$$y_{i+1} = K_i \left[y_i \pm x_i . 2^{-i} \right] \quad (3.9)$$

where: $K_i = \cos(\arctan(2^{-i}))$

By this the multiplications can be reduced to simple shift operations. We can also calculate the K factor

$$K = \prod_{i=0}^{n-1} K_i \quad (3.10)$$

in advance and implemented elsewhere. The equations 3-2 can be simplified as:

$$x_{i+1} = x_i \mp y_i . 2^{-i} \quad (3.11)$$

$$y_{i+1} = y_i \pm x_i . 2^{-i} \qquad (3.12)$$

3.2.2 Quantization errors of CORDIC algorithm

The quantization errors of CORDIC algorithm are introduced by limitations of iteration number, word length, and truncation and others operations. Due to the limited iteration number, the smallest angel also has its limitation. For example, a 16 step iteration can use the angel $\arctan(2^{-15})$ as the smallest calculation angel. Another fact to introducing quantization errors is the finite word length effect. More detailed information about quantization effects in CORDIC algorithm can be found in [27].

3.2.3 CORDIC architecture

There are many different ways to achieve the CORDIC equations (3.11 and 3.12) described above, such as: bit-parallel iterative CORDIC, bit-parallel unrolled CORDIC, bit-serial iterative CORDIC. The most common implementation of CORDIC in FPGA is bit-parallel iterative CORDIC. The schematic of bit-parallel iterative CORDIC is as Figure 3.8 below:

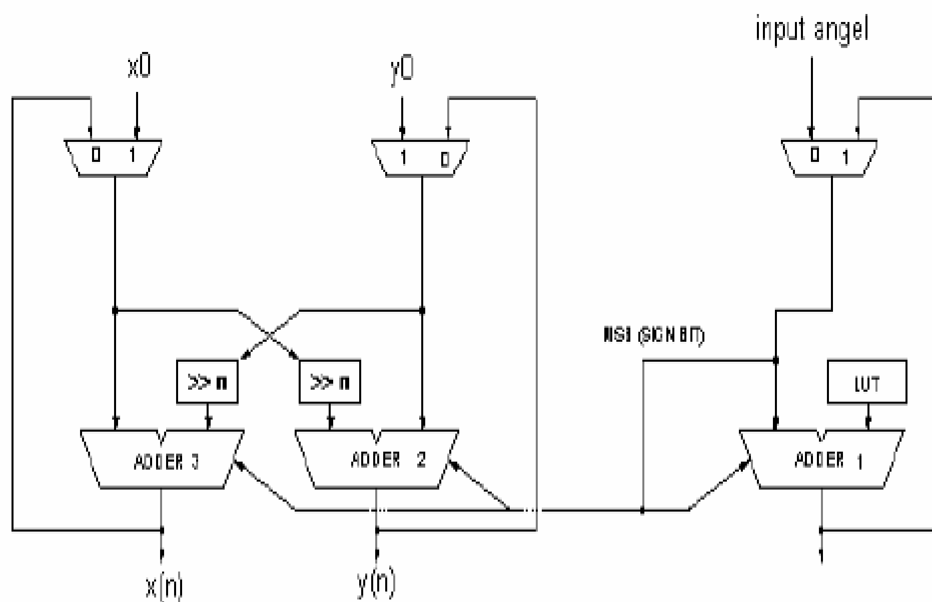


Figure 3.8. CORDIC architecture

The angel values arctan(2^{-i}) are stored in a LUT (look-up table). According to the iteration number, the angel value is loaded from LUT and depending on the MSB (sign bit) of the output of adder1, the next operations (addition or subtraction) can be decided. The MSB is also used to control two other adders to perform the operations (addition or subtraction) as described in equations (3.11 and 3.12).

Furthermore, the architecture can be divided into two major functional parts, which work in parallel:

One (CORDIC control) estimates the input angel with the angels pre-stored in LUT and generate the MSB signal. A counter is used as a small finite state machine to control the CORDIC.

The other (CORDIC data) calculates the unit sine and cosine values of the input angel which is controlled by the MSB signal. Because the shift function needed in CORDIC has a changing shift pace for different iteration steps, there are two major different shifters that can be used: barrel shifter and logarithm shifter.

As discussed in section 3.2.2, the word length has a great influence on the accuracy of the calculation result. By converting all the angels into the first quadrant, which is performed as truncating the first two bits from MSB side and add two '0' bits at LSB side, the word length can increase two bits, because we need two bits to represent the quadrant information of the input angel. The truncated two MSB are stored until the sine and cosine values are calculated. Then, the full quadrant sine and cosine can be generated by the two MSB that carry the quadrant information [1, 26].

When summarized we can say that Coordinate Rotation Digital Computer was invented in late 1950's. It was based on the observation that,if we rotate a unit-length vector (1,0) by an angle z then its new end-point will be at (cos z, sin z). It can evaluate virtually all

functions of interest but different equations are needed for it. K iterations require for k-bits accuracy [12].

3.2.4 Enhancements done in CORDIC

In 1987 Ercegovac and Lang proposed the redundant CORDIC. In sine and cosine computation by the redundant CORDIC, X_j's, Y_j's and Z_j's are represented by a redundant representation, q_j is selected from $\{\bar{1},0,1\}$ by evaluating a few most significant digits of Z_{j-1}, and the calculations of the iteration equations are performed in the redundant number system. The accurate evaluation of a redundant number takes a long time. Since no rotation-extension is performed for some angles, the scale factor becomes a variable dependent on the operand. Therefore, the scale factor has to be calculated during the computation, and the result has to be corrected with it. The calculation of the scale factor and the correction of the result with it require multiplications, a square rooting and a division, and increase the computation time and the amount of hardware. Ercegovac and Lang showed an on-line implementation of the redundant CORDIC to make the calculation of the scale factor and the correction overlap with the calculations of X_j's and Y_j's [16].

It was proposed in 1991 that CORDIC can be accelerated by the use of a redundant binary number representation, as in the previously proposed redundant CORDIC. Two new redundant CORDIC methods with a constant scale factor for sine and cosine computation, called the double rotation method and the correcting rotation method were proposed. In these methods, since the number of rotation-extensions performed for each angle was a constant, the scale factor was a constant independent of the operand. Hence, there was no need to calculate the scale factor during the computation, and a more efficient sine and cosine generator could be made based on the previous redundant CORDIC. The proposed methods accelerated the CORDIC by the use of a redundant8

number representation, as the previously proposed redundant CORDIC does. When a sine and a cosine generator is implemented as a combinational circuit consisting of logic gates with restricted fan-in, the depth (the computation time) of a generator based on either of the proposed methods is proportional to n (the word length of the operand and the results), while that of one based on the conventional CORDIC with ripple adders is proportional to n^2. The gate count of the former, as well as that of the latter, is proportional to n^2 [18].

Compared to the previous redundant CORDIC, the proposed methods require slightly more calculations in the iteration step. In the double rotation, the iterations equations are more complex. In the correcting rotation method, extra rotations are performed and more digits are looked into for the determination of the rotation direction. However, in the proposed methods, there is no need to calculate the scale factor during the computation nor to correct the result with it. A combination of the two proposed methods is also interesting. In the latter about half of the iterations, the calculations of the double rotation method are simpler than those of the coreecting rotation method. Hence, a combination where the calculations of the correcting rotation method are performed in the former half and those of the double rotation method are performed in the latter half is also efficient [18].

Duprat and Muller [1] introduced the ingenious "Branching CORDIC" algorithm. It enables a fast implementation of CORDIC algorithm using signed digits and requires a constant normalization factor. The speedup is achieved by performing two basic CORDIC rotations in parallel in two separate modules. In their method, both modules perform identical computation except when the algorithm is in a "branching" [11].

Improvement in the algorithm was done in 1998 and it was proposed as the Double Step Branching CORDIC algorithm and it was shown that it is possible to perform two circular mode rotations in a single step, with little additional hardware. In this method, both modules perform distinct computations at each step which leads to a better utilization of the hardware and the possibility of further speedup over the original

method. Double stepping appears to be the optimum when speed, hardware cost and utilization are considered [3].

In 1999, a very–high radix algorithm and implementation for the circular CORDIC in vectoring mode, which is used to compute arctan(b/a) and the modulus of vector (a,b) was presented. In order to use the simple selection by rounding, two pre–scalings are performed, one before the first iteration and another after it. Moreover, to reduce the overhead of the pre–scaling and adapt to the required precision, the first iteration is done using a smaller radix than the rest. The main requirement for this algorithm, as compared with radix 2 and radix 4, corresponds to an increase in the size of the tables for storing the elementary angles, the need for tables for the pre–scaling factors as well as the need of rectangular multipliers instead of adders in the serial implementation (for the pipelined implementation it was estimated that the total cost of the rectangular multipliers and of the adders is comparable). Moreover, when the modulus is required, the scale–factor compensation is performed by a logarithm–exponential approach, so that additional tables are needed. The number of input bits to each of these tables is about the logarithm of the radix, so that it was estimated that the implementation is reasonable up to a radix around 1024. A rough evaluation for 32–bit precision of both a pipelined and a word–serial implementation was performed and it showed a substantial speed up with respect to radix–2 and radix–4 implementations. This module complements the unit for CORDIC rotation presented in [6].

In 2005 Elisardo Antelo and Julio Villalba extended the approach of final multiplication to the vectoring mode of the CORDIC algorithm, by computing a reciprocal concurrently to the first iterations and a final multiplication using parallel tree multipliers. This is in contrast to previous proposals where the implementation of both modes of operation in the same architecture was constrained by a division operation, preventing the use of fast parallel tree multipliers. The linear approximation scheme is combined with the scale factor compensation, thus further reducing the delay. A comparison using a rough area–time model indicated that the proposed scheme achieved the significant delay and/or dynamic power reductions with no increase in area in actual implementations [5].

3.3 Algorithm comparison

The COSINE algorithm is hardware efficient as it uses only one subtractor to implement the square root functionality and one more subtractor which is used to subtract the intermediate result of each iteration. The CORDIC algorithm needs three subtractors which simultaneously do subtractions for its three equations for the simultaneous calculation of sine and cosine.

Inputs are in floating-point in the CORDIC algorithm while the inputs to the COSINE algorithm are in fraction (fixed-point). CORDIC needs variable shifter as the shifting done in each iteration is different while in case of COSINE algorithm the shifting of only two bits done and is not variable. An extra memory (ROM) is used in CORDIC while there is no need if an extra ROM in the COSINE algorithm.

The numbers of iterations for COSINE algorithm are n times more than that of CORDIC algorithm, where n is the number of input bits and hence the speed of this algorithm is slow.

CHAPTER 4

VHSIC Hardware Description Language (VHDL) and Field Programmable Gate Arrays (FPGAs)

4.1 FPGAs

4.1.1 Evolution of FPGAs

There are several options for an electronic system designer to implement any digital logic. These options include discrete logic devices such as Small-Scale Integrated Circuits (SSIC); Programmable Logic Devices (PLDs); Masked-Programmed Gate Arrays (MPGAs) and Field Programmable Gate Arrays (FPGAs). Programming of such a device often involves placing the chip into a special Programming unit, but some chips can also be configured "in-system". Another name for PLDs is Field Programmable Devices (FPDs). A Programmable Logic Device (PLD) is a set of fully connected macro cells. These macro cells are typically comprised of some amount of combinational logic (for example, AND-OR gates) and a flip-flop. In other words, a small Boolean logic equation can be built within each macro cell [32].

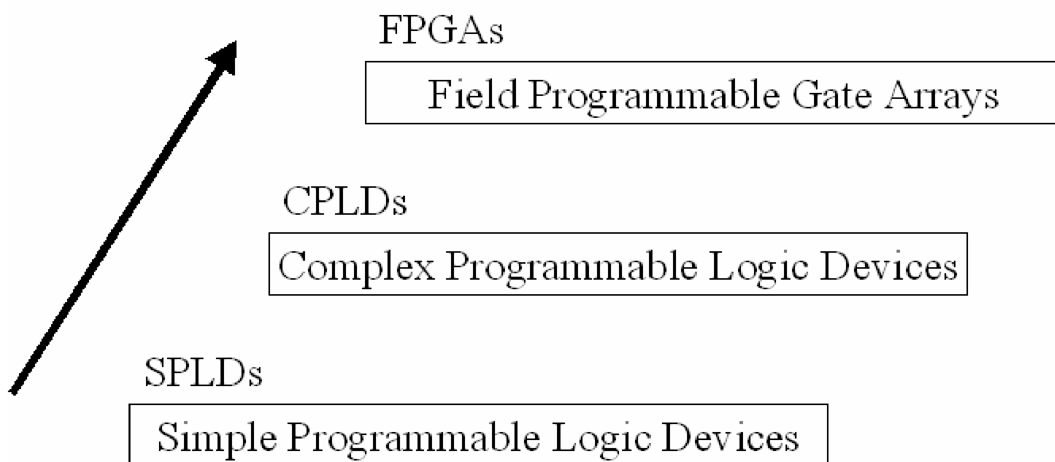


Figure 4.1. Evolution of FPGAs

4.1.2 FPGA and its internal Structure

FPGA is general-purpose programmable chip, which can be programmed to carry out a specific hardware function specified by the user. It is an array of programmable logic blocks connected with programmable interconnects. FPGAs can have medium-to-high capacity (equivalent to that of thousands to millions of logic gates).

FPGA is a device in which the logic structure can be directly configured by the end user without the use of an IC fabrication facility. It is high density Programmable Logic Device containing small logic cells interconnected through a distributed array of programmable switches. This type of architecture produces statistically varying results in performance and functional capacity, but offers high register counts. Programmability typically is via volatile SRAM or one-time-programmable anti-fuses. It is a very complex PLD. These devices are fastest programmable logic devices with gate counts running into millions. These devices are user customizable and programmable on an individual device basis. The speed of FPGA can be up to 200MHz or more. There are many vendors of FPGAs. Some of them are Xilinx, Altera, Lattice, Actel, Quicklogic, Cypress and Atmel.

Internal structure of FPGA has three key parts:

- Logic blocks
- Interconnect
- I/O Blocks

The I/O blocks form a ring around the outer edge of the part. Each of these provides individually selectable input, output, or bi-directional access to one of the general-purpose I/O pins on the exterior of the FPGA package. Inside the ring of I/O blocks lies a rectangular array of logic blocks. And connecting logic blocks to logic blocks and I/O blocks to logic blocks is the programmable interconnect wiring [32].

4.2 VHDL Introduction

VHDL is a programming language that has been designed and optimized for describing the behavior of digital systems.

VHDL has many features appropriate for describing the behavior of electronic components ranging from simple logic gates to complete microprocessors and custom chips. Features of VHDL allow electrical aspects of circuit behavior (such as rise and fall times of signals, delays through gates, and functional operation) to be precisely described. The resulting VHDL simulation models can then be used as building blocks in larger circuits (using schematics, block diagrams or system-level VHDL descriptions) for the purpose of simulation.

VHDL is also a general-purpose programming language: just as high-level programming languages allow complex design concepts to be expressed as computer programs, VHDL allows the behavior of complex electronic circuits to be captured into a design system for automatic circuit synthesis or for system simulation. Like Pascal, C and C++, VHDL includes features useful for structured design techniques, and offers a rich set of control and data representation features. Unlike these other programming languages, VHDL provides features allowing concurrent events to be described. This is important because the hardware described using VHDL is inherently concurrent in its operation.

One of the most important applications of VHDL is to capture the performance specification for a circuit, in the form of what is commonly referred to as a test bench. Test benches are VHDL descriptions of circuit stimuli and corresponding expected outputs that verify the behavior of a circuit over time. Test benches should be an integral part of any VHDL project and should be created in tandem with other descriptions of the circuit [10].

4.2.1 History of VHDL

VHDL, which stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, was developed in the early 1980s as a spin-off of a high-speed

integrated circuit research project funded by the U.S. Department of Defense. During the VHSIC program, researchers were confronted with the daunting task of describing circuits of enormous scale (for their time) and of managing very large circuit design problems that involved multiple teams of engineers. With only gate-level design tools available, it soon became clear that better, more structured design methods and tools would be needed.

To meet this challenge, a team of engineers from three companies — IBM, Texas Instruments and Intermetrics — were contracted by the Department of Defense to complete the specification and implementation of a new, language-based design description method. The first publicly available version of VHDL, version 7.2, was released in 1985. In 1986, the Institute of Electrical and Electronics Engineers, Inc. (IEEE) was presented with a proposal to standardize the language, which it did in 1987 after substantial enhancements and modifications were made by a team of commercial, government and academic representatives. The resulting standard, IEEE 1076-1987, is the basis for virtually every simulation and synthesis product sold today. An enhanced and updated version of the language, IEEE 1076-1993, was released in 1994, and VHDL tool vendors have been responding by adding these new language features to their products.

Although IEEE Standard 1076 defines the complete VHDL language, there are aspects of the language that make it difficult to write completely portable design descriptions (descriptions that can be simulated identically using different vendors' tools). The problem stems from the fact that VHDL supports many abstract data types, but it does not address the simple problem of characterizing different signal strengths or commonly used simulation conditions such as unknowns and high-impedance.

Soon after IEEE 1076-1987 was adopted, simulator companies began enhancing VHDL with new, non-standard types to allow their customers to accurately simulate complex electronic circuits. This caused problems because design descriptions entered into one simulator were often incompatible with other simulation environments. VHDL was quickly becoming a nonstandard.

To get around the problem of nonstandard data types, another standard was developed by an IEEE committee. This standard, numbered 1164, defines a standard package (a VHDL feature that allows commonly used declarations to be collected into an external library) containing definitions for a standard nine-valued data type. This standard data type is called std_logic, and the IEEE 1164 package is often referred to as the Standard Logic package.

The IEEE 1076-1987 and IEEE 1164 standards together form the complete VHDL standard in widest use today. (IEEE 1076-1993 is slowly working its way into the VHDL mainstream, but it does not add significant new features for synthesis users.)

Standard 1076.3 (often called the Numeric Standard or Synthesis Standard) defines standard packages and interpretations for VHDL data types as they relate to actual hardware. This standard, which was released at the end of 1995, is intended to replace the many custom (nonstandard) packages that vendors of synthesis tools have created and distributed with their products.

IEEE Standard 1076.3 does for synthesis users what IEEE 1164 did for simulation users: increase the power of Standard 1076, while at the same time ensuring compatibility between different vendors' tools. The 1076.3 standard includes, among other things:

1) A documented hardware interpretation of values belonging to the bit and boolean types defined by IEEE Standard 1076, as well as interpretations of the std_ulogic type defined by IEEE Standard 1164.

2) A function that provides "don't care" or "wild card" testing of values based on the std_ulogic type. This is of particular use for synthesis, since it is often helpful to express logic in terms of "don't care" values.

3) Definitions for standard signed and unsigned arithmetic data types, along with arithmetic, shift, and type conversion operations for those types.

The annotation of timing information to a simulation model is an important aspect of accurate digital simulation. The VHDL 1076 standard describes a variety of language features that can be used for timing annotation. However, it does not describe a standard method for expressing timing data outside of the timing model itself.

The ability to separate the behavioral description of a simulation model from the timing specifications is important for many reasons. One of the major strengths of Verilog HDL (VHDL's closest rival) is the fact that Verilog HDL includes a feature specifically intended for timing annotation. This feature, the Standard Delay Format, or SDF, allows timing data to be expressed in a tabular form and included into the Verilog timing model at the time of simulation.

The IEEE 1076.4 standard, published by the IEEE in late 1995, adds this capability to VHDL as a standard package. A primary impetus behind this standard effort (which was dubbed VITAL, for VHDL Initiative Toward ASIC Libraries) was to make it easier for ASIC vendors and others to generate timing models applicable to both VHDL and Verilog HDL. For this reason, the underlying data formats of IEEE 1076.4 and Verilog's SDF are quite similar [33].

4.2.2 Entities and Architectures

Every VHDL design description consists of at least one entity/architecture pair.

An entity declaration describes the circuit as it appears from the "outside" - from the perspective of its input and output interfaces. If you are familiar with schematics, you might think of the entity declaration as being analogous to a block symbol on a schematic.

The second part of a minimal VHDL design description is the architecture declaration. Before simulation or synthesis can proceed, every referenced entity in a VHDL design description must be bound with a corresponding architecture. The architecture describes

the actual function—or contents—of the entity to which it is bound. Using the schematic as a metaphor, you can think of the architecture as being roughly analogous to a lower-level schematic referenced by the higher-level functional block symbol [20, 21].

4.2.3 Levels of Abstraction

VHDL supports many possible styles of design description. These styles differ primarily in how closely they relate to the underlying hardware. When we speak of the different styles of VHDL, we are really talking about the differing levels of abstraction possible using the language—behavior, dataflow, and structure [20, 21].

Equation 1

CHAPTER 5

Design And Implementation Of Function Generator

5.1 Function Generator

The Function generator implemented calculates the function of the 8 bit input. This 8 bit input is the degrees divided by 180 for the calculation of the trigonometric functions and hence is a fraction. The input to the square root could be any number of 8 bits fraction. The function is selected with the help of the input (function to do) of 2 bits. The function performed is one of the four functions (cos, sine, square root, tan) depending upon the 2 bits input. When these two bits are "00", the cosine function is performed, when it is "01", the sine function is performed, when it is "10", the square root function is performed and when it is "11", the tangent function is performed. The flowchart of the function generator is shown in the figure below.

The output of the function generator is also of 8 bits which is also a fraction and the one bit integer output shows if there is any integer in the output. Sign of the result is given in the sign output, if the sign output is 0 then the result is positive and if the sign output is 1 then the result is negative. There is different case for the output in tan, unlike other outputs the tan output is all integer and is contained in the 8 bit output word.

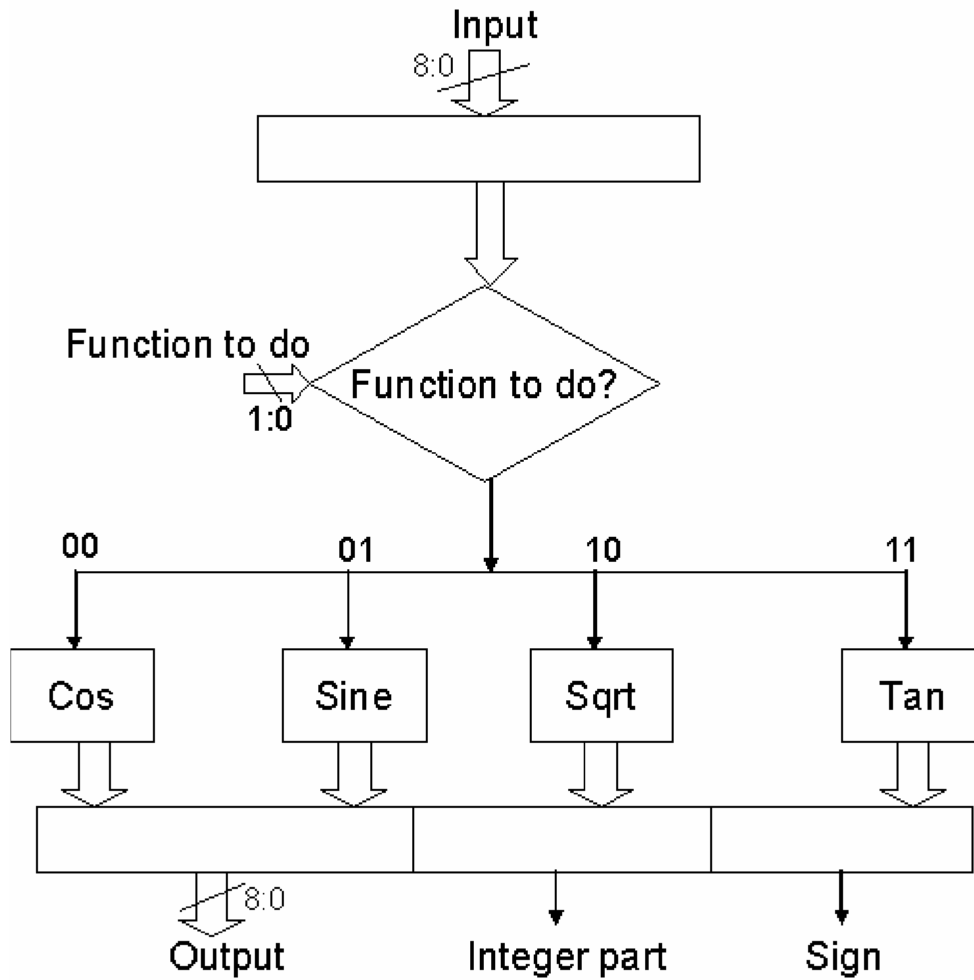


Figure 5.1. Flowchart of Function Generator

5.2 Design flow

The top down design methodology has been proposed to validate and verify the design, as shown in the figure 4.1 below. Based on the algorithms selected, a corresponding VHDL view has been created. Then the VHDL view is simulated, if the specification is satisfied, then the design is going through the synthesis flow, else if the specification is not satisfied, modifications on the VHDL view are introduced. After the synthesis flow, whether or not modifications are needed depend on the hardware constrains, such as area, and timing constrains [32, 33].

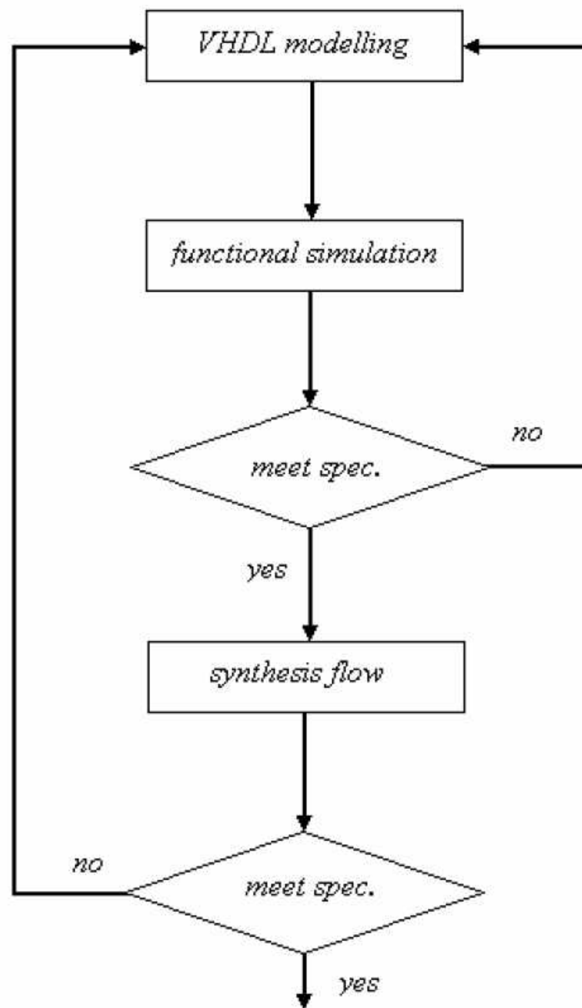


Figure 5.2. Implementation of design flow

As described in the above section, the function generator architecture consists of one major module which consists of a hierarchy of four sub-modules (cos values calculation, sine values calculation, square root values calculation, tan values calculation) which work according to the select input which decides which function is to be performed. Cos module consists of two sub-modules. The implementation started with modeling and simulation of sub-modules. Then, the major module is composed with a structural and dataflow view (mixed modeling style) of the validated sub-modules. The sub-modules are also simulated and modified. Finally, the whole implementation goes through the design flow for validation.

5.3 Algorithm implementation: floating-point versus fixed-point

According to the Webster Dictionary, an algorithm is "a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end especially by a computer."

Typical (although by no means the only) operations are those of addition and multiplication. When expressing the algorithm with pencil and paper, these operations are commonly taken to be within an algebraically complete number system such as the integers or the reals. However, when the time comes to implement the algorithm on a computer, these "ideal" number systems must be exchanged for something realizable. The number systems available today on common processors and digital hardware are broadly categorized as floating-point and fixed-point.

In a floating-point representation, the total number of bits available are partitioned into an exponent and mantissa. Generally speaking, the mantissa stores the "significant digits" of the value while the exponent scales the significant digits to the desired magnitude. The action of the exponent is to move, or "float," the decimal point depending on the magnitude being represented; thus the term "floating-point."

Because floating-point representations are typically at least 32 bits long (IEEE-754 is a popular standard for 32-bit and 64-bit floating-point numbers), there exists simultaneously high precision and high dynamic range. These traits of floating-point numbers allow most algorithms to be ported directly to floating-point implementations with little or no change, and this is the key reason floating-point representations are highly desirable. The disadvantage of floating-point implementations is that they require a significant amount of extra hardware over fixed-point implementations, which translates to higher parts costs, higher power consumption, slower execution, larger chip area, or a combination of these.

As the term "fixed-point" implies, fixed-point representations have the binary point at a fixed location. There are two subsets of fixed-point implementations: fractional and

integer. In a fractional fixed-point implementation, such as that provided on the Motorola 56K series of DSPs, the binary point is always assumed to be to the left of the most-significant digit. In an integer fixed-point implementation, such as that provided by the Texas Instruments TMS320C54xx series of DSPs, the binary point is to the right of the least-significant digit. In either case, the arithmetic operations implemented in the hardware are essentially integer, which results in a much simpler arithmetic logic unit in hardware that allows lower cost, lower power consumption, faster execution, smaller chip area, or a combination of these, over that of floating-point implementations.

In this implementation we have used the fractional fixed-point representation to represent the number which is totally a fraction.

5.4 Numerical representation

The accuracy of the arithmetic operations is crucial of the whole project, because there are a large number of iteration operations working.

Using integer arithmetic will greatly decrease the complexity of the implementation. Another advantage of using fractional-fixed point numerical representation in this design is that the intermediate results have a large range of fractional numbers and values bigger than 1 need not to be represented, which is of no better use if to be represented by fixed point number or floating point number. So using integer representation is a reasonable solution.

Similarly to the 8-bit standard integer representation has used 8 bits. And the word format as below:

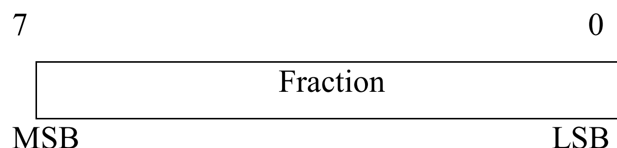


Figure 5.3. Word format

Instead of using the fixed point representation and assuming the point to be next to the MSB (most significant bit), we have used a shifter after each arithmetic operation to perform the function of adding and subtracting 1 to the fraction and then division by 2.

When performing iterative operations, due to the number which is all a fraction, there would be no problem in the representation because the result is also a fraction only. In the square root operations, the input will require a doubled word length to prepare the result of square root for next step operation in the COSINE algorithm.

Basic arithmetic operation of subtraction is done in 2's complement representation, after each arithmetic operation, the leading zeros or ones are left shifted out and the shifted zeros or ones decides the sign of the result of the subtraction. And when there are no overflows, the negative result is represented in 2's complement representation. By this the algorithm speed due to less computation will dramatically increase in all of its iterations.

The deviation from the accurate values introduced by each operation of square root algorithm will accumulate in the cos operation. Because using a 8 bits intrinsic word length, same as the inputs, can not lead to a satisfying results, the errors are accumulated after several steps of iterations, the intrinsic word length need to be increased. We have simulated with several different sets of input values, to keep a 8 bits accurate result, the arithmetic operations needed a much longer word length than 8 bits. And hence operations on doubled word length result from each multiplication.

Let's see two examples, how the square root works with 8-bits and 16-bits.

Example 1: 8-bits

In the fixed point case, suppose the fixed point is to the left of the MSB and the input is all fraction then the representation could be only integer. Result from a 8 bits input would be only in 4 MSBs and the 4 LSBs would be all zero. Result from a 8 bits input "10111000" will be "11010000". After the truncation operation, this will be the result.

Example 2: 16-bits
In the fixed point case, suppose the fixed point is to the left of the MSB and the input is all fraction then the representation could be only integer. If 8 zeros are appended after the LSB of the input then there would be no error as the number is a representation of a fraction in an integer representation. Result from a 16 bits input would be in 8 bits. Result from a 16 bits input "10111000" will be "11011001". This will be the result which has less error then the previous one.

5.5 Modeling and simulation

The design is modeled with text entries of HDL designer. Each level of design hierarchy is simulated with Modelsim (simulator) to verify the functionality. Modifications are applied when the performance is not satisfying. Modeling and simulation is divided into following steps:

I. The basic arithmetic operations have been written into a different file as different entities, such as addition/subtraction, inversion and so on.

II. After all the entities are verified by simulation, sub modules are created and simulated.

III. The major module is composed from validated sub modules.

IV. The top level of the Function Generator module consists of the four sub-modules.

5.6 Arithmetic functions

A VHDL entity file is created which includes all the basic arithmetic operations used for integer operations.

5.6.1 Square root implementation

The square root has 8 bit input and to make it 16 bits, 8 zeros are appended at the left side of the input. The input of the square root is made 16 bits for more accuracy and to give the result of full 8 bits. The input to the entity is a fraction of 8 bit and the output is also a fraction of 8 bit. The flowchart of the square root is shown below:

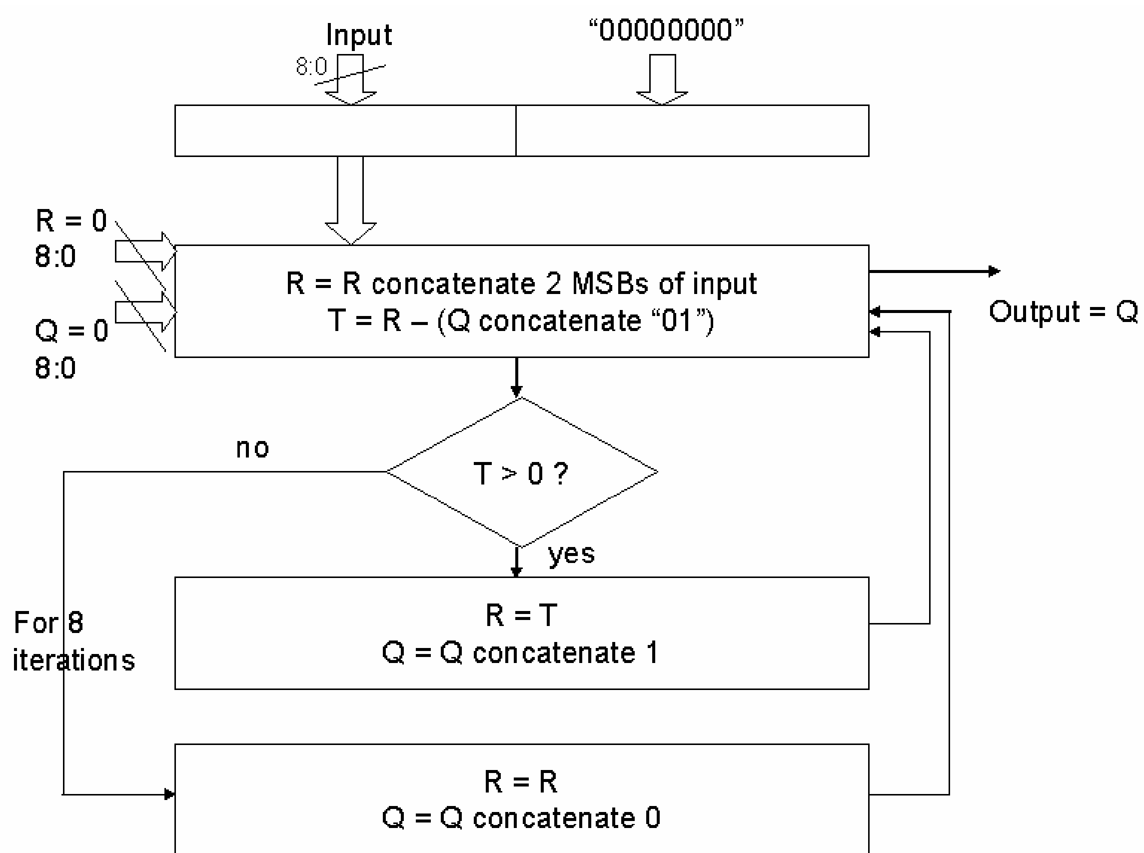


Figure 5.4. Flowchart of Square Root

There are temporary variables R and T which are used for the calculation of Q (output). The calculation of the result is an iterative procedure which runs for 8 iterations

according to the above flowchart. After the iterations are completed the output is now stored in the Q register and this is the square root of the input.

5.6.1.1 Square root state diagram

The square root is implemented with VHDL code with the help of Finite State Machine (FSM). The implementation of the square root is coded in 10 states which are explained with the help of the state diagram shown in the figure below:

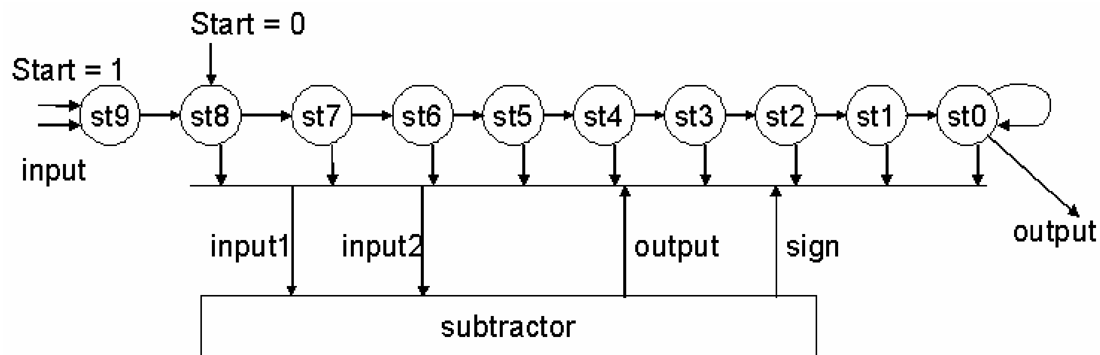


Figure 5.5. State diagram of Square Root

The start input takes the entity in the st9 state when one and when it goes zero the state changes to st8. In this state the variables are initialized, R has 2 MSBs of input at its 2 LSBs. The input1 to the subtractor from the st7 state is the temporary variable R and another input input2 is temp variable which has “01” at its 2 LSBs. The output is stored in another variable T. The sign bit output tells the sign of the subtraction performed. If it is one then the output from the subtractor is positive otherwise negative. If the output is positive then the LSB of Q is an one otherwise it is assigned a zero. The input is shifted two bits left. If the sign bit is one then the value of the temporary variable R becomes the output concatenated with the 2 MSBs of the input otherwise the variable R remains unchanged and concatenated with the 2 MSBs of the input. The temp variable becomes the Q concatenated with “01”. The state then jumps to the next state which is st6 and in

this state also the same as the st7 state happens. After going through the 8 iterations which are from state st7 to st0 the output is generated and is stored in the Q. This Q is the square root of the input. Until no input is given to the square root entity and the start bit is not made one, the state of this entity remains in the st0 state.

5.6.2 Cosine implementation

The Cosine module has 8 bit input. The input is in degrees divided by 180, hence this module calculate the cos of an angle which is between 0 to 180 degrees. The input to the entity is a fraction and the output is also a fraction. The output is of 8 bit which is the cos of the input angle and one sign bit which gives the sign of the cos of the input angle. The flowchart of the Cosine module is shown below:

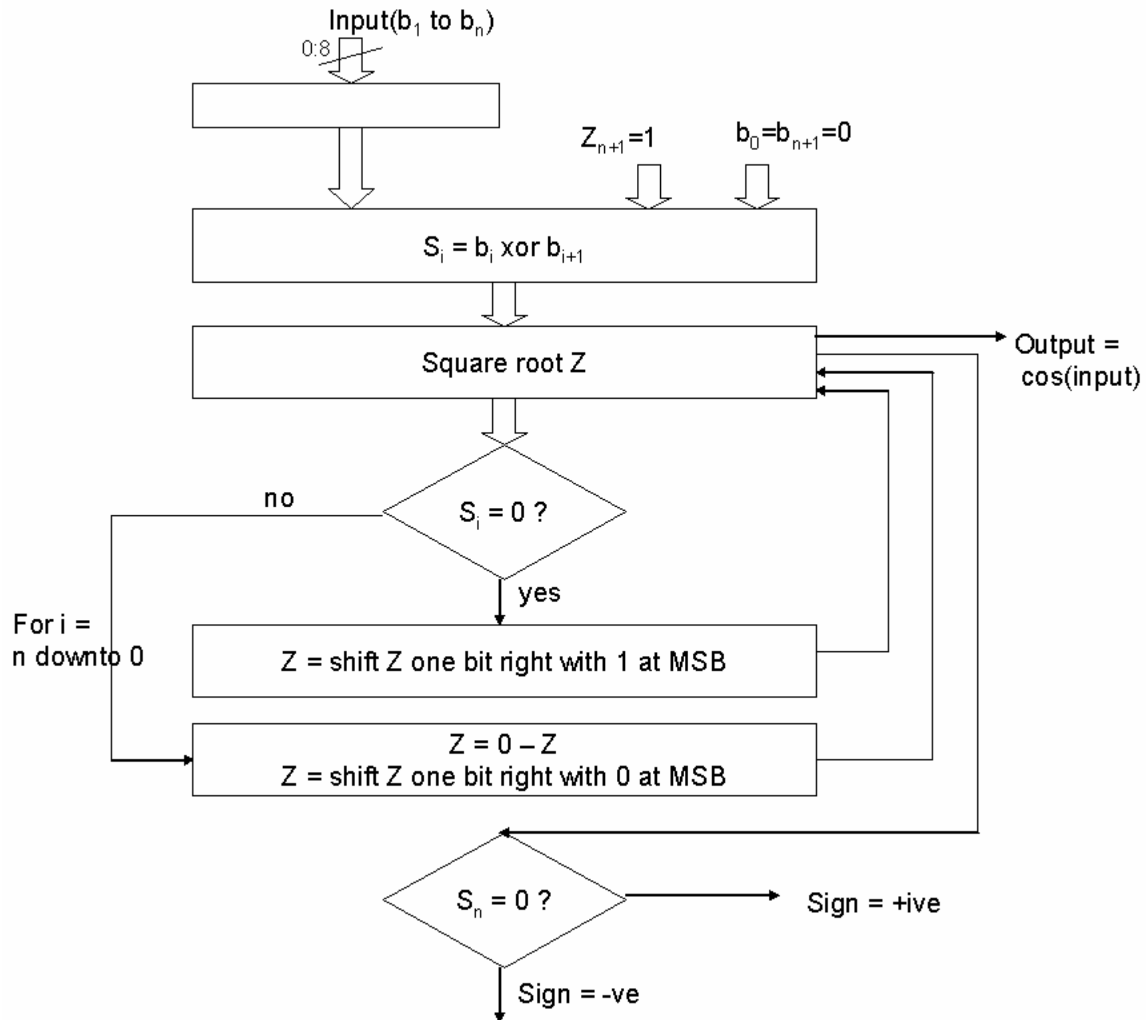


Figure 5.6. Flowchart of Cosine

There is a temporary variable S which stores all the Exclusive-ORs of the bit pairs in the input. There is a signal Z which is initially 1. The square root of this variable is taken repeatedly with the help of the square root module. In every iteration i, it is checked whether the ith bit of S is one or zero and accordingly the value of the Z signal is modified. If S_i is 0 then Z is only shifted right with 1 at the most significant bit position, this shifting implements the logic of adding with 1 and then dividing by 2. If S_i is 1 then Z is subtracted from 1. This logic is implemented by taking 1's complement of "00000000" and then adding it to Z with the help of a ripple-carry adder consisting of 8 full adders and the carry in bit is set to 1. The output is the subtraction of Z from 1, as Z is a fraction the output of this subtraction is also a fraction. The resultant is only shifted

right with 0 at the most significant bit position, this shifting implements the logic of subtracting Z from 1 and then dividing by 2.

The calculation of the result is an iterative procedure which runs for 8 iterations according to the above flowchart. After the iterations are completed the output is now stored in the Z register and this is the cos of the input angle. The value of the ith bit of S in the last iteration i decides the sign of the output. If it is 0 then the output is positive otherwise the output is negative.

5.6.2.1 Cosine state diagram

The cosine algorithm is implemented with VHDL code with the help of Finite State Machine (FSM). The implementation of the cosine algorithm is coded in 10 states which are explained with the help of the state diagram shown in the figure below:

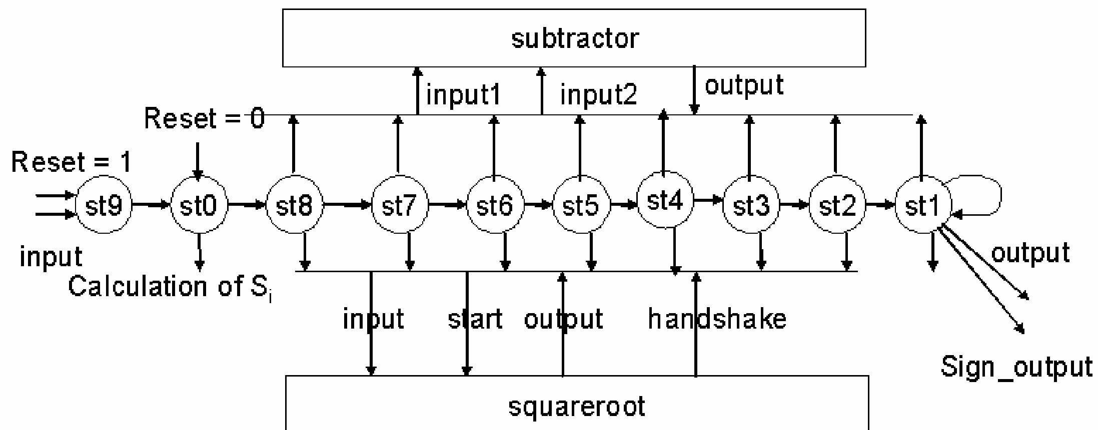


Figure 5.7. State diagram of Cosine

The reset input takes the entity in the st9 state when one and when it goes zero the state changes to st0. In this state the variables are initialized, S(1) to S(8) takes the values of the Exclusive-Ors of the bit pairs of the input, S(1) is ‘0’ xor input(7) and S(2) is input(7) xor input(6) and so on.

After this initialization the state changes to st8. in this state the value of Z is 1 and there is a one bit signal integ which tells whether the value of Z is 1 and if it is 0 then it represents that Z is a fraction. The square root of the fractional value of the Z is taken iteratively. The input to the square root from the st8 state is the fractional value of the Z and another input is start signal which starts the FSM (finite state machine) of the square root module by taking it into the start state. The handshake output from the square root module goes to the cosine module and changes the value of the start signal from 1 to 0 which again goes to the square root module and changes its state. In the last state the square root module gives the square root of the input which is the fractional part of the Z signal and again changes the handshake signal which in turn changes the state of the cosine module from st8 to st7.

In every state the value of the ith bit of S is checked and if it is 0 only shifting is done of Z otherwise subtraction of Z from 1 is to be done. For the implementation of the subtraction a subtraction module is used which does the eight bit subtraction. The input to this subtraction module is fractional part of Z and another input is "00000000" and the output of this module is also an 8 bit word representing a fraction.

In this st7 state again the square root of the modified signal Z is taken. The state then jumps to the next state which is st6 and in this state also the same as the st7 state happens. The output from the square root module again modifies the value of the Z signal according to the value of the next S bit as it was modified in the st7 state. After going through the 8 iterations which are from state st8 to st1 the output is generated and is stored in the Z. This Z is the square root of the input. There is one bit output sign which tells the sign of the cos of the input and this sign is the value of the ith bit of S in the last iteration i. Until no input is given to the cosine entity and the restart bit is not made one, the state of this entity remains in the st1 state.

5.6.3 Sine implementation

As the sine of any angle is equal to the cos of 90 minus that angle, we can implement the sine function by the use of the following equations:

$$\sin(\theta) = \cos(90 - \theta) \tag{5.1}$$

Suppose $\theta = 180 * \alpha$

$$\sin(\theta) = \cos(90 - (180 * \alpha)$$

$$= \cos(180(\tfrac{1}{2} - \alpha)) \tag{5.2}$$

Hence the sine of the input can be the cos of ½ - input. If the half in the above equation is represented in the binary form then it would be .1 and after subtracting the input from .1 we can take the cos of that which would be the sine of the input. The overflow bit is checked to see whether the output is negative or not if it is negative the 2's complement of the output is taken and the cosine of that is calculated. As the sign of sine is positive in the 0 to 180 range, there is no effect of sign of the subtraction on the result.

To implement the above logic a subtractor is used which does a subtraction of 8 bits. The same 8 bit subtractor used in the cosine module can be used in the sine module also. The sine module after doing the subtraction goes to the cosine module and then the output of the cosine module is the sine of the input.

5.6.4 Tan implementation

Tan is implemented with the help of the formula:

$$\tan(\theta) = \sin(\theta) / \cos(\theta) \tag{5.3}$$

First the cos of the input is calculated by making the select bits "00" and the when output is received the select bits are made "01"and the start bit is made 1 and then 0 to calculate the sine of the input. The values of the sign and cosine are assigned to the inputs of the divider and then when the select lines or the 2 bits of the function to do input are made "11", the tan of the input is calculated.

The sign of the output is same as the sign of the cosine module when it calculated the cos of the input and the integer bits of the output, i.e. the tan of the input are in the 8 bits of the output which were used to represent fraction in the case of cos, sine and square root. The integer bit is zero in this case. The tan has very less accuracy, it only has the accuracy of integer bits and the fractional part of the tan of the input is not represented in any form in the output.

5.6.5 Subtractor_16 implementation

The subtractor_16 does subtraction of 16 bits. The subtractor code has process statement with for loop in it which is executed 16 times and every time a full adder is implemented. The carry in bit is 1 and the input to be subtracted is in the 1's complement form. The carry out bit decides the sign of the output. This module is a sub-module of the square root module.

5.6.6 Subtractor_8 implementation

The subtractor_8 does subtraction of 8 bits. The subtractor code has process statement with for loop in it which is executed 8 times and every time a full adder is implemented. The carry in bit is 1 and the input to be subtracted is in the 1's complement form. The carry out bit decides the sign of the output. This module is a sub-module of both the cosine and sine modules.

5.6.7 Division implementation

Division module is the sub-module of the tangent module and is implemented with help of the non-restoring division algorithm. The code has process statement with for loop in it which is executed 8 times and every time an 8 bit subtractor is implemented. The inputs to this sub-module are fractions but the output of this sub-module is integer.

CHAPTER 6
Results & Discussions

6.1 Simulation and Synthesis results

There are different modules used in the functioning of function generator, which are programmed using VHDL [20,21], simulated using simulator Modelsim. Xilinx Integrated Softwrae environment (ISE) software is used for the synthesis of the code. Implementation is done on Xilinx Spartan2E xc2s600e-6Qfg676 [32].

Different modules of the function generator are:

1) Square root module
2) Cosine module
3) Subtractor_16 module
4) Subtractor_8 module

6.1.1 Simulation results of subtractor _16 module

After doing coding in VHDL language [10,20], the simulation is done using Modelsim Simulator in Xilinx Integrated Software Environment (ISE) software. The inputs to the subtractor_16 are two 16 bit number. The output is one 16 bit output and one sign bit. The simulation result for inputs of “1100101001010000” & “0111000101010011” is an output of “0101100011111101” & 1.

Design Statistics:

IOs : 49

Device utilization summary:

Number of Slices: 22 out of 6912 0%

Number of 4 input LUTs: 40 out of 13824 0%

Maximum combinational path delay: 30.005ns

I/O view of the subtractor_16 module is shown in Figure 6.1(a).

6.1.2 Simulation results of subtractor _8 module

After doing coding in VHDL language, the simulation is done using Modelsim Simulator in Xilinx Integrated Software Environment (ISE) software. The inputs to the subtractor_8 are two 8 bit number. The output from this module is one 8 bit output. The simulation result for an input of “01010100” and “00110011” and is an output of “00010000”.

Design Statistics

IOs : 24

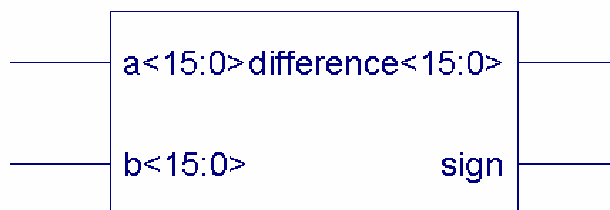
Device utilization summary:

Number of Slices: 15 out of 6912 0%

Number of 4 input LUTs: 27 out of 13824 0%

Maximum combinational path delay: 21.177ns.

I/O view of the subtractor_8 module is shown in Figure 6.1(b).



(a)

(b)

Figure 6.1. I/O view of (a): subtractor_16, (b): subtractor_8

6.1.3 Simulation results of square root module

The inputs to the square root are one 16 bit number, one bit clock and one bit start. The output from this module is one 8 bit output and one bit handshake. The simulation result for an input of "1010100100000000", clock with a period of 100ns and start bit 1 and then 0 and output "11010000" is shown in the Figure 6.3.

Design Statistics

IOs : 27

Device utilization summary:

Number of Slices: 6 out of 6912 0%

Number of Slice Flip Flops: 10 out of 13824 0%

Number of 4 input LUTs: 8 out of 13824 0%

Number of GCLKs: 1 out of 4 25%

Maximum combinational path delay: no path found

Minimum period: 3.191ns (Maximum Frequency: 313.421MHz)

I/O view of the square root module is shown in Figure 6.2. The Detailed view drawing of square root is split horizontally into three sheets representing its three views: first one is shown in Figure 6.5 as view_1, second one is shown in Figure 6.6 as view_2, and the last one is shown in Figure 6.6 as view_3.

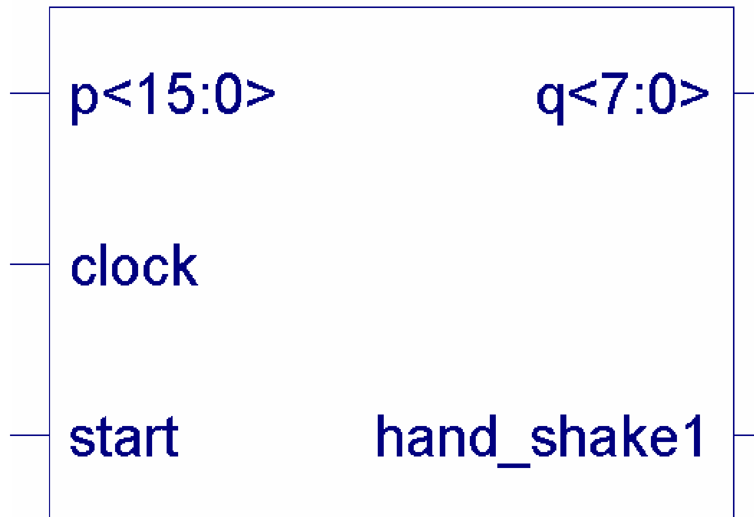


Figure 6.2. I/O view of square root

The three views are connected with help of connecters as shown in the Figure 6.4 and as explained in the Table 6.1. The inputs and outputs in the Figures 6.5, 6.6 and 6.7 are marked with the help of numbers and explained in the Table 6.1. The different instances in the Figure 6.5, 6.6 and 6.7 are marked with help of capital alphabets.

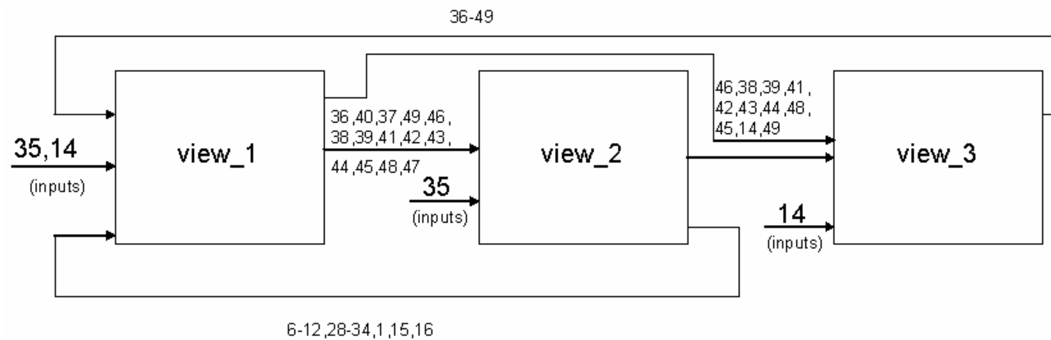


Figure 6.4. Splitted detailed view of square root module

With the help of alphabets X,Y the functionality of the instance is explained by supposing them as the inputs in Figures 6.5, 6.6 and 6.7. With the help of these supposed inputs the output of the instances are explained in the form of Boolean expression. In the detailed Figure 6.5 A is the subtractor_16, B is 13 bit latch, C is Mux. E has the functionality of X'Y and D instance has eight 2-input And gates whose output goes to 8-input Or gate and the output of the Or gate is the final output. F instance in Figures 6.5, 6.6 and 6.7 has seven 2-input And gates whose output goes to 7-input Or gate and the output of the Or gate is the final output.

Table 6.1. Representation of symbols in square root module

Symbol	**Representation**
35	p(15:0)
14	start
49,46,38,39,41-45	p_state(0)-p-state(8)
1-13,15-34	connectors connecting the output from view_2 and view_3 to inputs in view_1
36,40,37,49,46,38,39, 41,42,43,44,45,48,47	connectors connecting the output from view_1 to inputs in view_2

46,38,39,41,42,43, 44,48,45,14,49	connectors connecting the output from view_1 to inputs in view_3

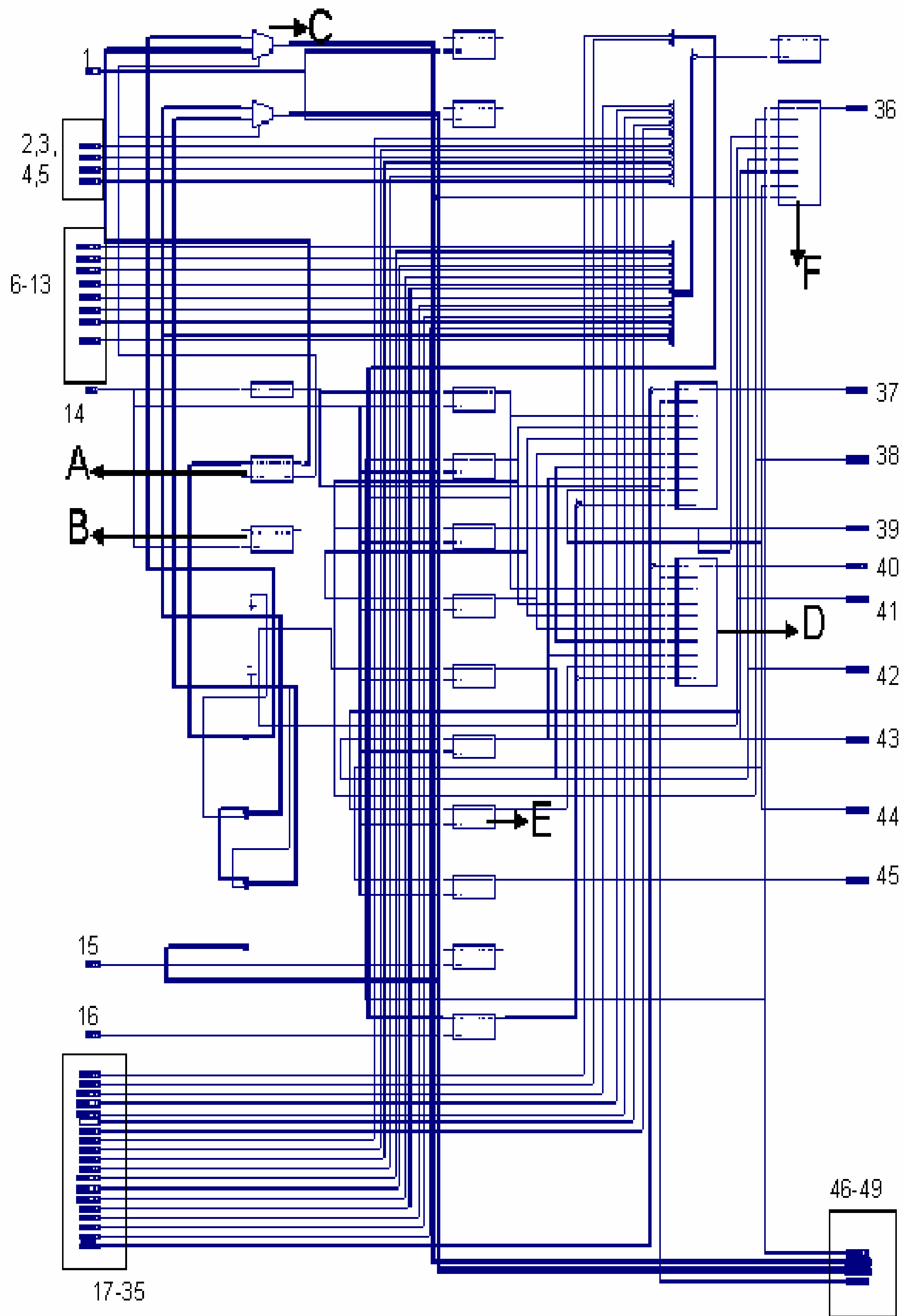

C
1
2,3,
4,5
6-13
36
F
14
37
A
38
B
39
40
D
41
42
43
E
44
45
15
16
46-49
17-35

Figure 6.5. Detailed View_1 of square root

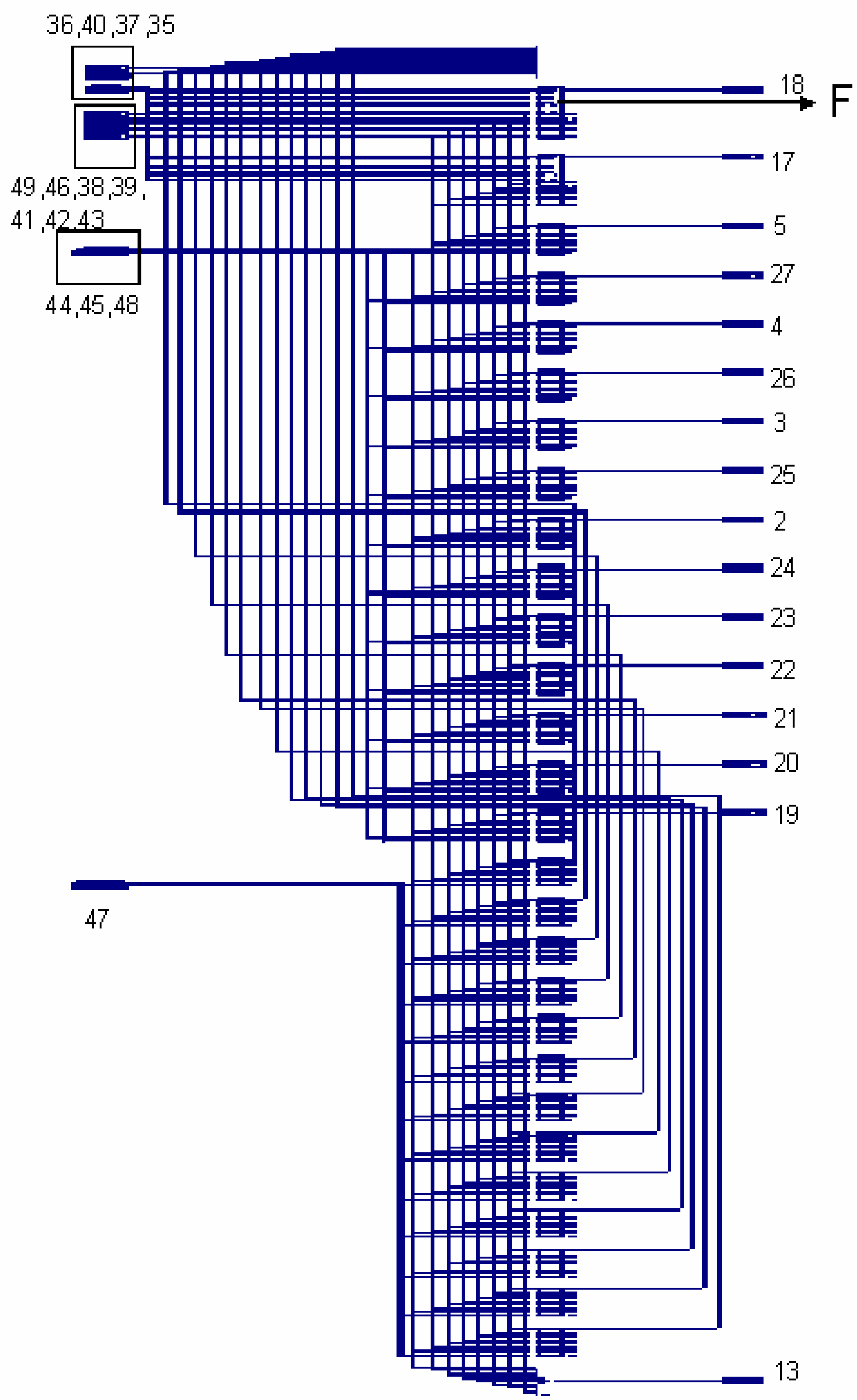

36 ,40 ,37 ,35
49 ,46 ,38,39 ,
41 ,42,43
44 ,45 ,48
47
18
F
17
5
27
4
26
3
25
2
24
23
22
21
20
19
13

Figure 6.6. Detailed View_2 of square root

46 ,38 ,39 ,
41 ,42 ,43 ,
44 ,48

34
F
12
33
11
32
10
31
9
30
8
29
7
28
6

48 ,45
14
16
1
15
49

Figure 6.7. Detailed View_3 of square root

6.1.4 Simulation results of cosine module

The inputs to the cosine are one 8 bit number, one bit clock1 and one bit reset. The output from this module is one 8 bit output and one sign bit which gives the sign of the output. The simulation result for an input of "10110000", clock with a period of 100ns and reset bit equal to 0 and output "10001110" and the sign output equal to 1 is shown in the Figure 6.9.

Design Statistics

IOs : 19

Device utilization summary:

Number of Slices: 17 out of 6912 0%

Number of Slice Flip Flops: 29 out of 13824 0%

Number of 4 input LUTs: 23 out of 13824 0%

Maximum combinational path delay: no path found

Minimum period: 4.991ns (Maximum Frequency: 200.377MHz).

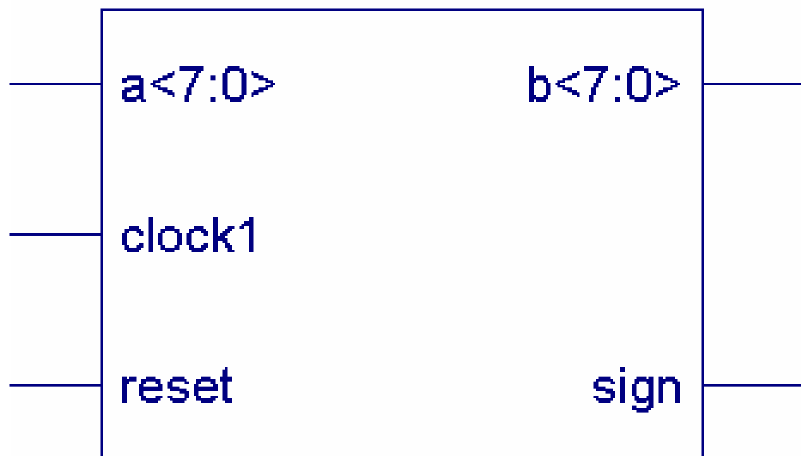


Figure 6.8. I/O view of cosine

I/O view of the cosine module is shown in Figure 6.8.

The Detailed view drawing of cosine is split horizontally into three sheets representing its three views: first one is shown in Figure 6.11 as view_1, second one is shown in Figure 6.12 as view_2, and the last one is shown in Figure 6.13 as view_3.

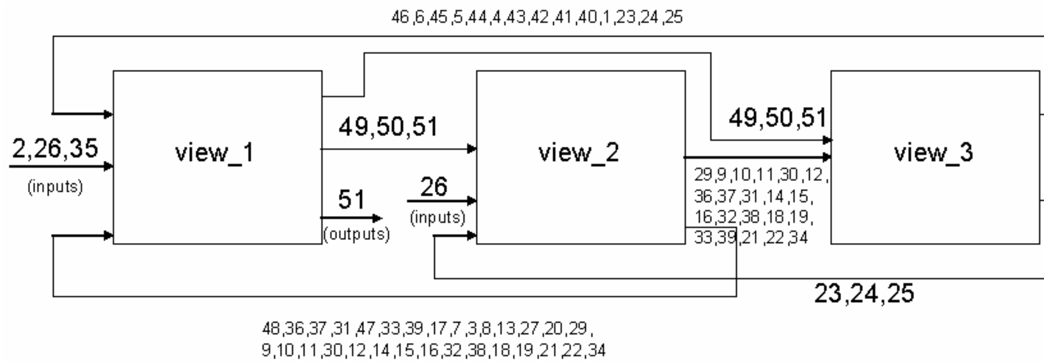


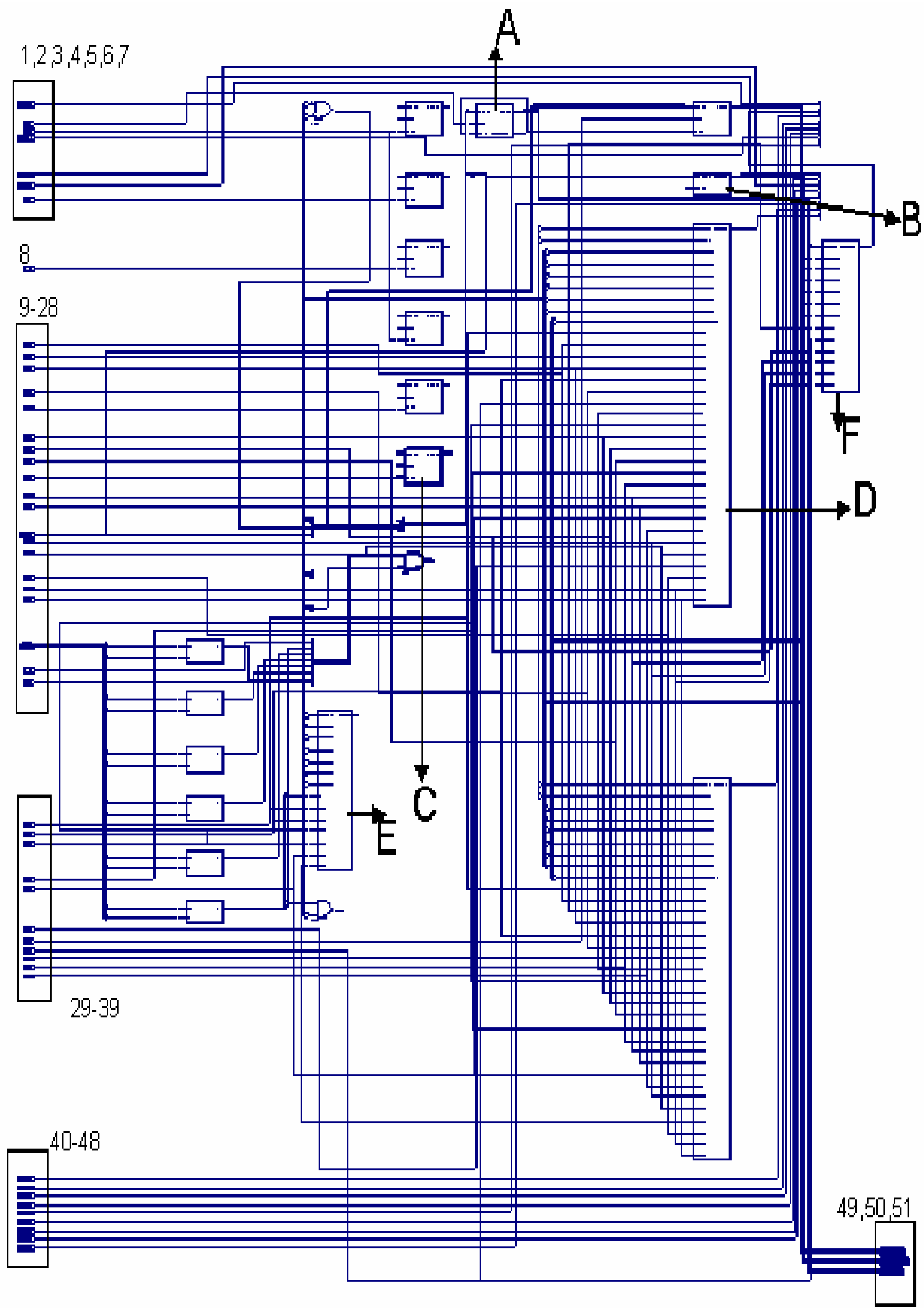
Figure 6.10. Splitted detailed view of cosine module

The three views are connected with help of connecters as shown in the Figure 6.10 and as explained in the Table 6.2. The inputs and outputs in the Figures 6.11, 6.12 and 6.13 are marked with the help of numbers and explained in the Table 6.2. The different instances in the Figure 6.11, 6.12 and 6.13 are marked with help of capital alphabets.

With the help of alphabets X,Y the functionality of the instance is explained by supposing them as the inputs in Figures 6.11, 6.12 and 6.13. With the help of these supposed inputs the output of the instances are explained in the form of Boolean expression. In the detailed Figure 6.11 A is the square root module, B is the subtractor_8 module and C is the latch. D instance in Figures 6.11, 6.12 and 6.13 has twenty four 2-input And gates whose output goes in combination of four in six 4-input Or gates and then anded with six 2-input And gates and then to a 6-input Or gate and the output of the Or gate is the final output. F instance in Figures 6.11 and 6.12 has six 3-input And gates whose output goes to 6-input Or gate and the output of the Or gate is the final output. E instance in Figure 6.11 has seven 2-input And gates whose output goes to 7-input Or gate and the output of the Or gate is the final output. G in Figure 6.12 has the functionality of X’Y. In Figure 6.12 instance H has outputs from 2-input six And gates and six Or gates goes to 9-input And gate and the output from the 9-input And gate is the output and I instance has seven 3-input And gates, six 2-input Or gates and one 8-input And gate.

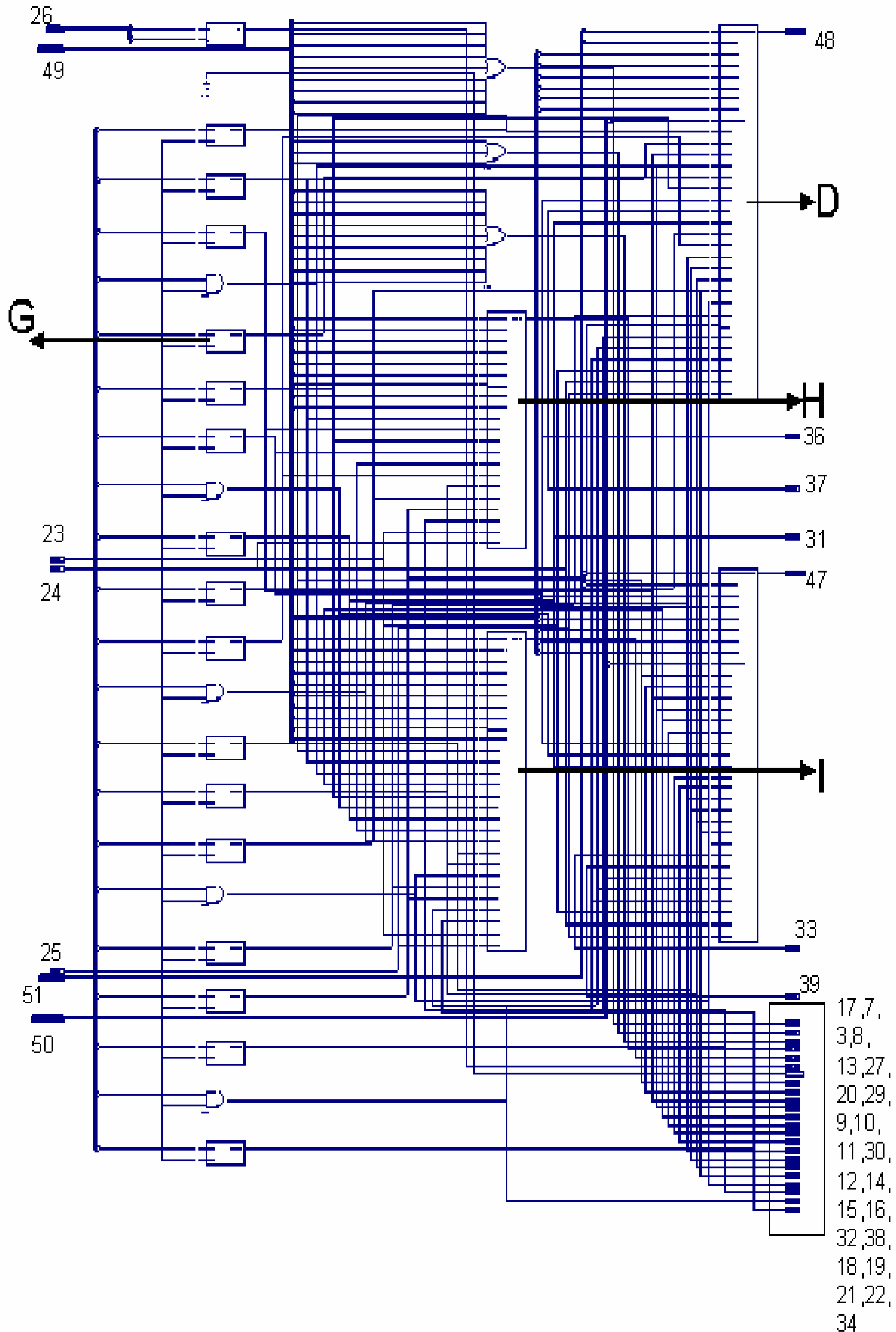
Table 6.2. Representation of symbols in square root module

Symbol	Representation
26	a(7:0)
20	temp(0)
2	clock
28	a(7)
35	reset
49	p_state(9:0)
51	output(7:0)
50	sub_output(7:0)
1,3-19,21-25,27,29-34,36-48	connectors connecting the output from view_2 and view_3 to inputs in view_1
29,9,10,11,30,12,36,37,31,14,15,16,32,38,18,19,33,39,21,22,34	connectors connecting the output from view_2 to inputs in view_3
23,24,25	connectors connecting the output from view_3 to inputs in view_2



1,2,3,4,5,6,7
8
9-28
29-39
40-48
49,50,51
A
B
C
D
E
F

Figure 6.11. Detailed View_1 of cosine



26
49
48
D
G
H
36
37
23
31
24
47
I
33
25
51
39
50
17,7,
3,8,
13,27,
20,29,
9,10,
11,30,
12,14,
15,16,
32,38,
18,19,
21,22,
34

Figure 6.12. Detailed View_2 of cosine

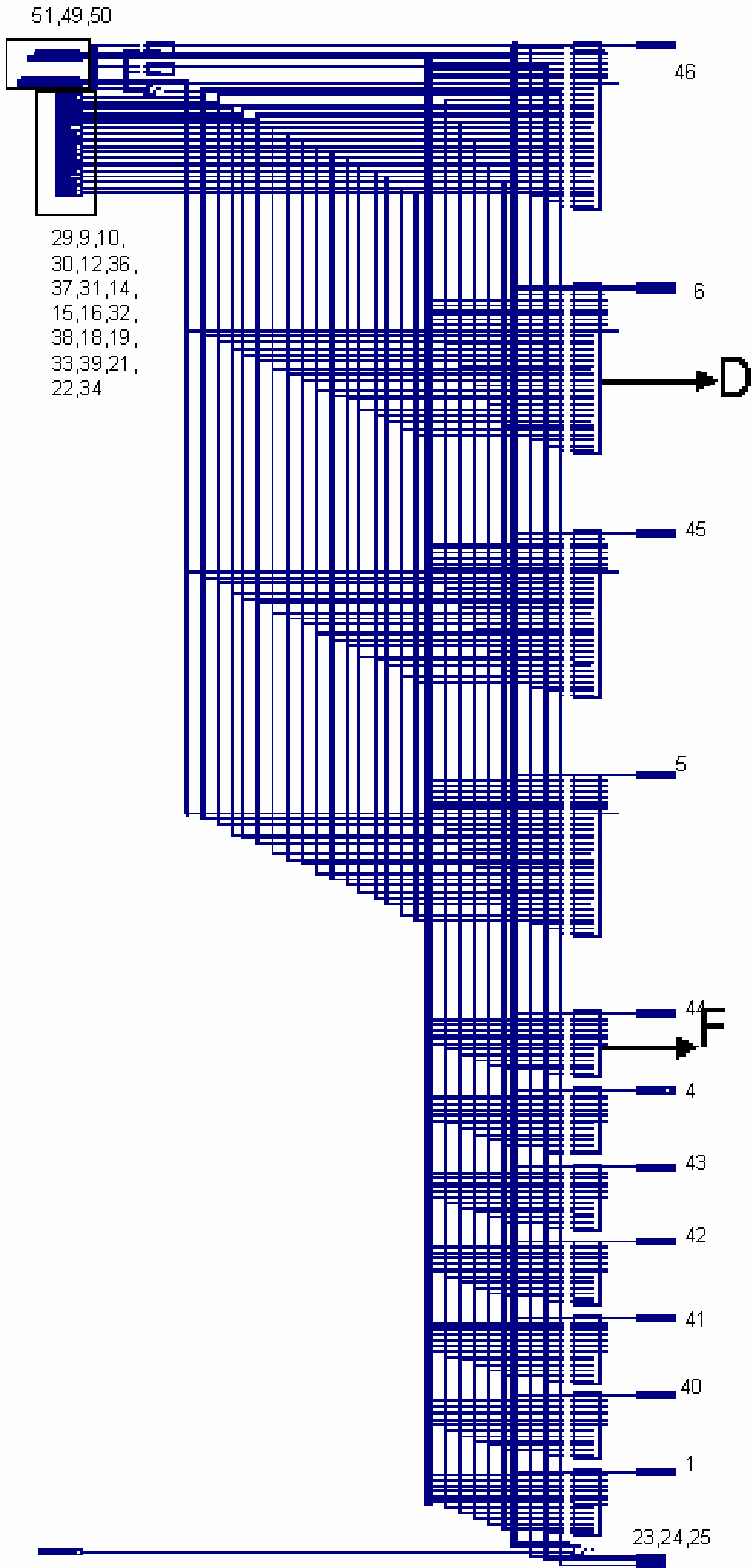

51,49,50
46
29,9,10,
30,12,36,
37,31,14,
15,16,32,
38,18,19,
33,39,21,
22,34
6
D
45
5
44
F
4
43
42
41
40
1
23,24,25

Figure 6.13. Detailed View_3 of cosine

6.1.5 Simulation results of Function generator module

The inputs to the function generator are one 8 bit number, one two bit number which act as select lines to select which function to perform, one bit clock and one bit start_function. The output from this module is one 8 bit output, one bit sign output and one bit integer output. The simulation result for an input of “00110111”, clock with a time period of 100ns and two bit function to do input equal to “01” (perform the sine of the input) and output “10011101”, sign output equal to 1 and integer output equal to 0 is shown in the Figure 6.15.

Design Statistics

# IOs	: 22

Device utilization summary:

Number of Slices:	124	out of	6912	1%
Number of Slice Flip Flops:	60	out of	13824	0%
Number of 4 input LUTs:	216	out of	13824	1%

Maximum combinational path delay: 14.197ns

Minimum period: 4.991ns (Maximum Frequency: 200.377MHz).

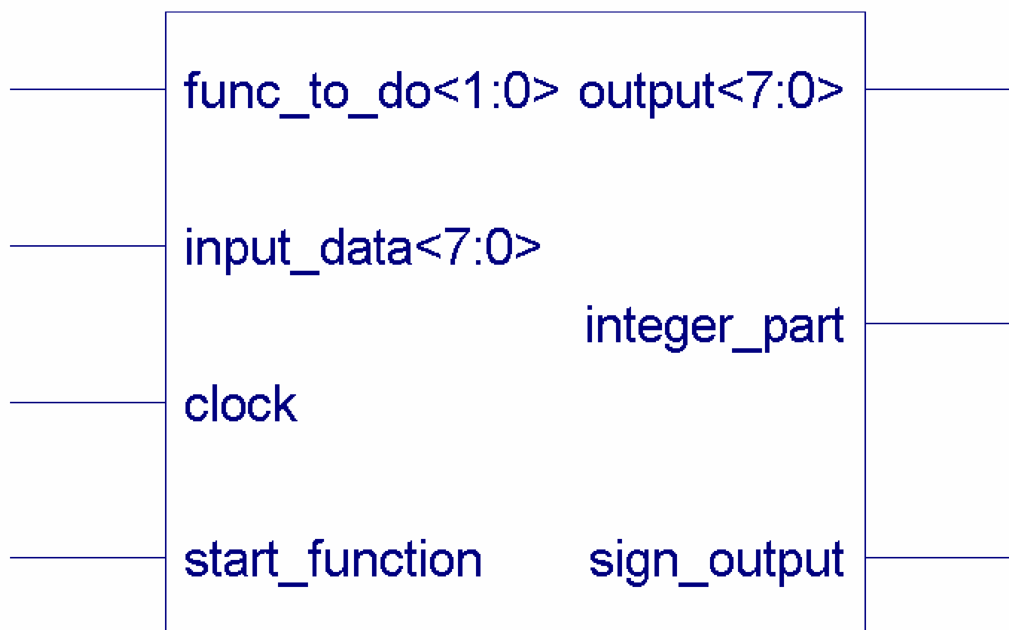


Figure 6.14. I/O view of function generator

I/O view of the Function generator module is shown in Figure 6.14.

The Detailed view drawing of Function generator is split horizontally into two sheets representing its two views: first one is shown in Figure 6.17 as view_1, and second one is shown in Figure 6.18 as view_2.

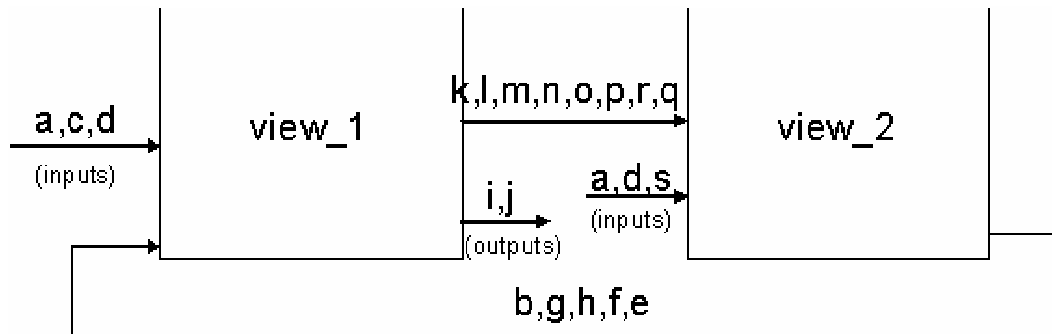


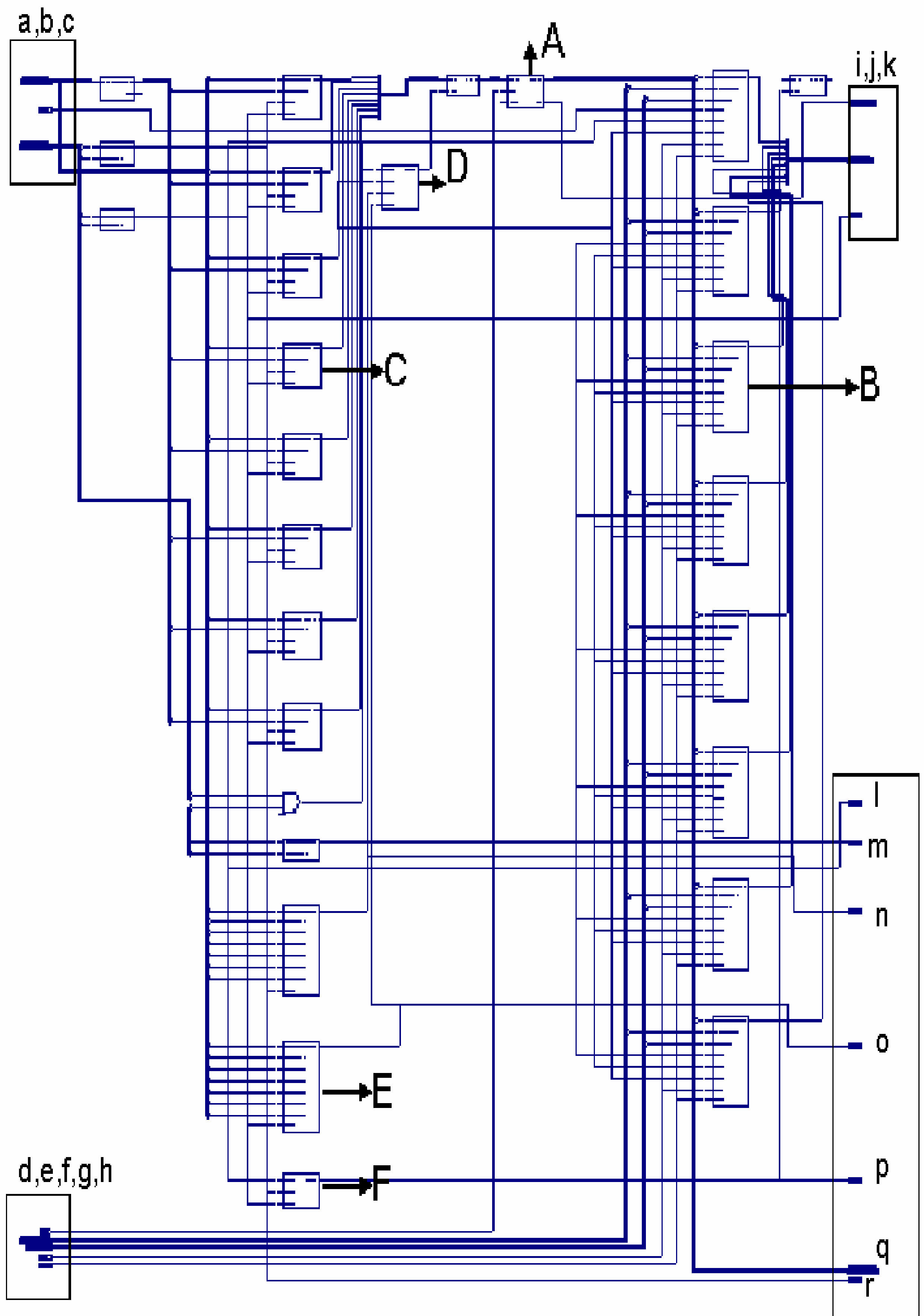
Figure 6.16. Splitted detailed view of function generator

The two views are connected with help of connecters as shown in the Figure 6.16 and as explained in the Table 6.3. The inputs and outputs in the Figures 6.17 and 6.18 are marked with the help of small alphabets and explained in the Table 6.3. The different instances in the Figure 6.17 and 6.18 are marked with help of capital alphabets.

With the help of alphabets S, T, U, V, W, X, Y, Z the functionality of the instance is explained by supposing them as the inputs in Figures 6.17 and 6.18. With the help of these supposed inputs the output of the instances are explained in the form of Boolean expression. In the detailed Figure 6.17 A is the cosine module, B has three 2-input And gates, two 3-input And gates, one 2-input Or gate and one 3-input Or gate, C has the functionality of ST + UV and F is 3-input Nor. In Figures 6.17 and 6.18 D is 4-input Nor and E has the functionality of STUVWXYZ’. In Figure 6.18 H is square root module, G is 7-input Nor and I has the functionality S’TU + S’TV.

Table 6.3. Representation of symbols

Symbol	**Representation**
a	input_data(7:0)
c	func_to_do(1:0)
d	clock
e	sqrt_output(7:0)
f	tan_out(7:0)
i	sign_cos_output
j	output(7:0)
q	cos_output(7:0)
s	start_function
b,g,h	connectors connecting the output from view_2 to inputs in view_1
k,l,m,n,o,p,r	connectors connecting the output from view_1 to inputs in view_2



a,b,c
A
i,j,k
D
C
B
l
m
n
o
E
d,e,f,g,h
F
p
q
r

Figure 6.17. Detailed View_1 of function generator

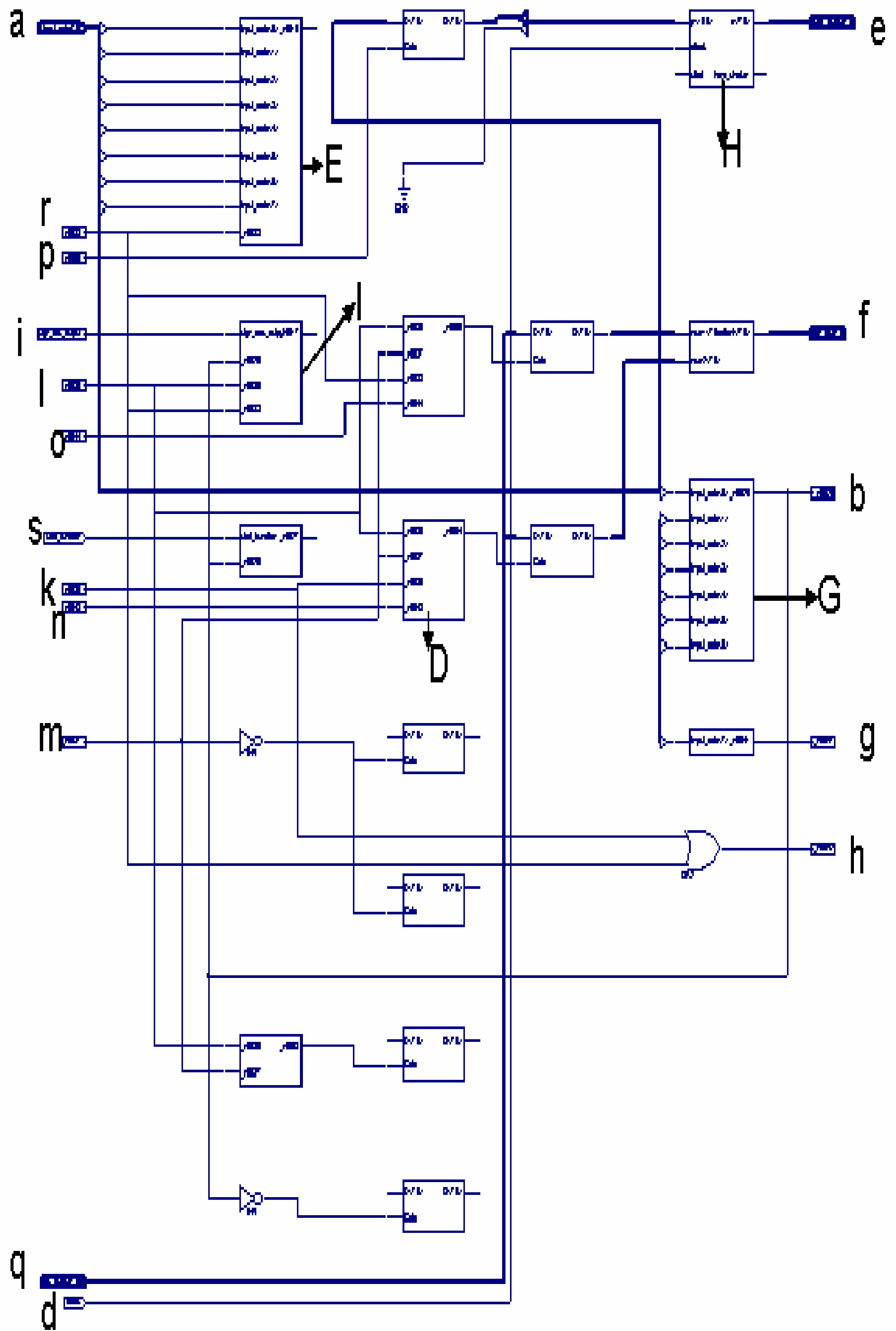

a
e
E
H
r
p
i
I
f
l
o
b
s
k
n
G
D
m
g
h
q
d

Figure 6.18. Detailed View_2 of function generator

For example:

The instance E in Figure 6.18 could be elaborated as shown in Figure 6.19.

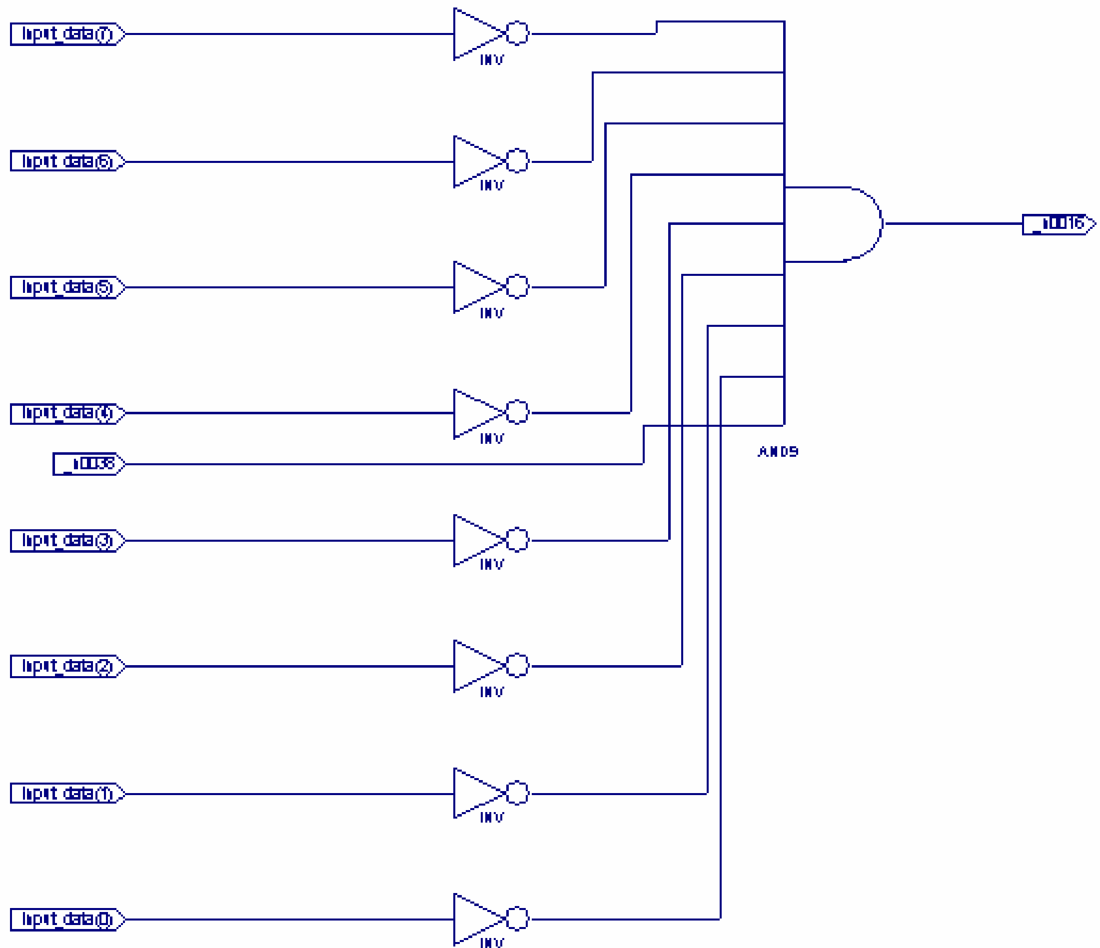


Figure 6.19. Elaborated instance E of Function generator

6.2 Graphs showing comparison with CORDIC

6.2.1 Comparison of square root

Comparison of square root values calculated from the restoring square root algorithm with calculator values is done as shown by the graph in Figure 6.20. Error varies from -2.9% to 2%. The error is high for low values up to input of 0.18 and then it is in the range of -0.5 to 0.0. The results show that this algorithm has high accuracy. If the number of bits are increased the accuracy is also increased.

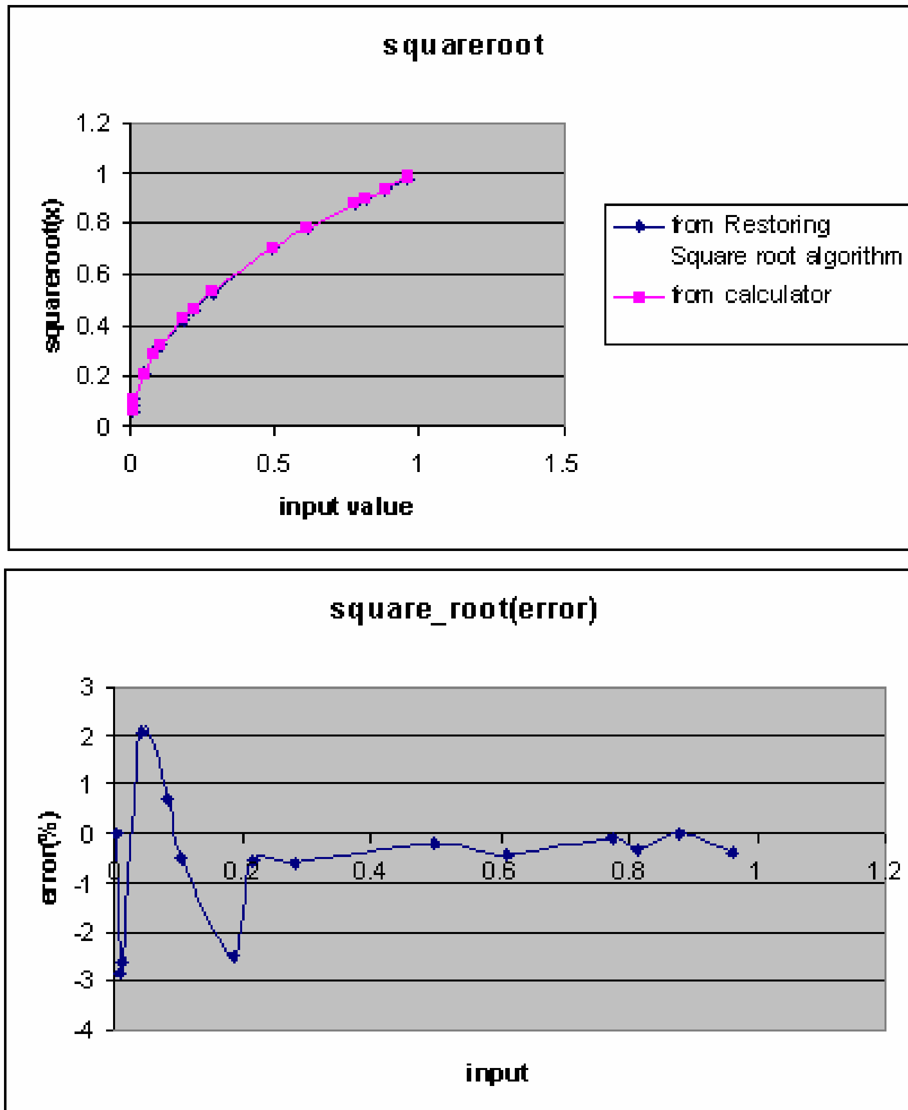


Figure 6.20. Error in square root calculation

6.2.2 Comparison of cosine values

Comparison of cosine values calculated from the Fowkes [23] algorithm with calculator values is done as shown by the graph in Figure 6.21. Error varies from -1.4% to 0.19%. The error is high for low values up to input of 38 and then it is in the range of -0.45 to 0.12. The results show that this algorithm has high accuracy.

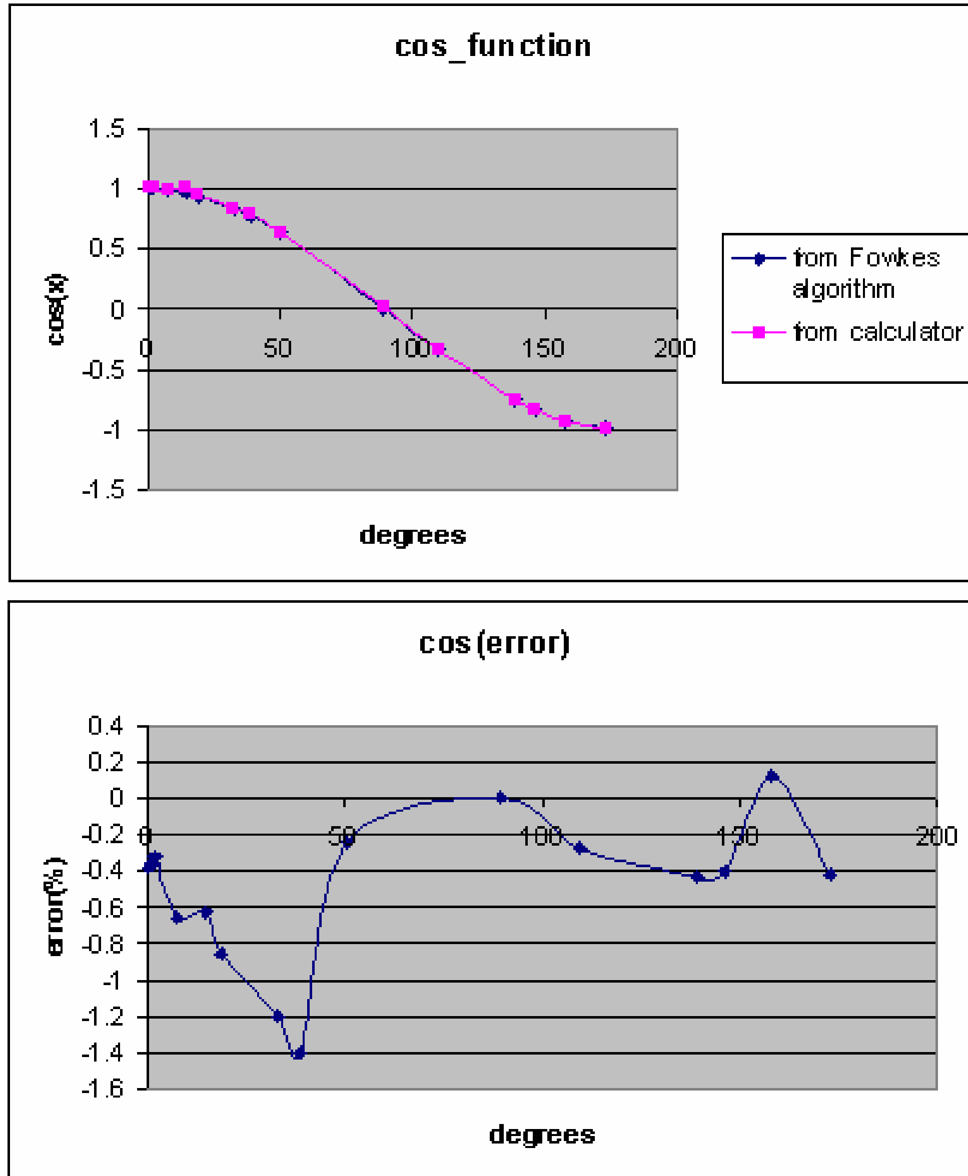


Figure 6.21. Error in cosine calculation

6.2.3 Comparison of sine values

Comparison of sine values calculated from the Fowkes [23] algorithm with calculator values is done as shown by the graph in Figure 6.22. Error varies from -6.5% to 1.9%. The error is high for low values up to input of 14 and then it is in the range of -1.8 to 1.8.

The results show that this algorithm has high accuracy. The accuracy of this is little less than cosine, which is because of a subtraction operation which is performed in the starting which leads to less accuracy.

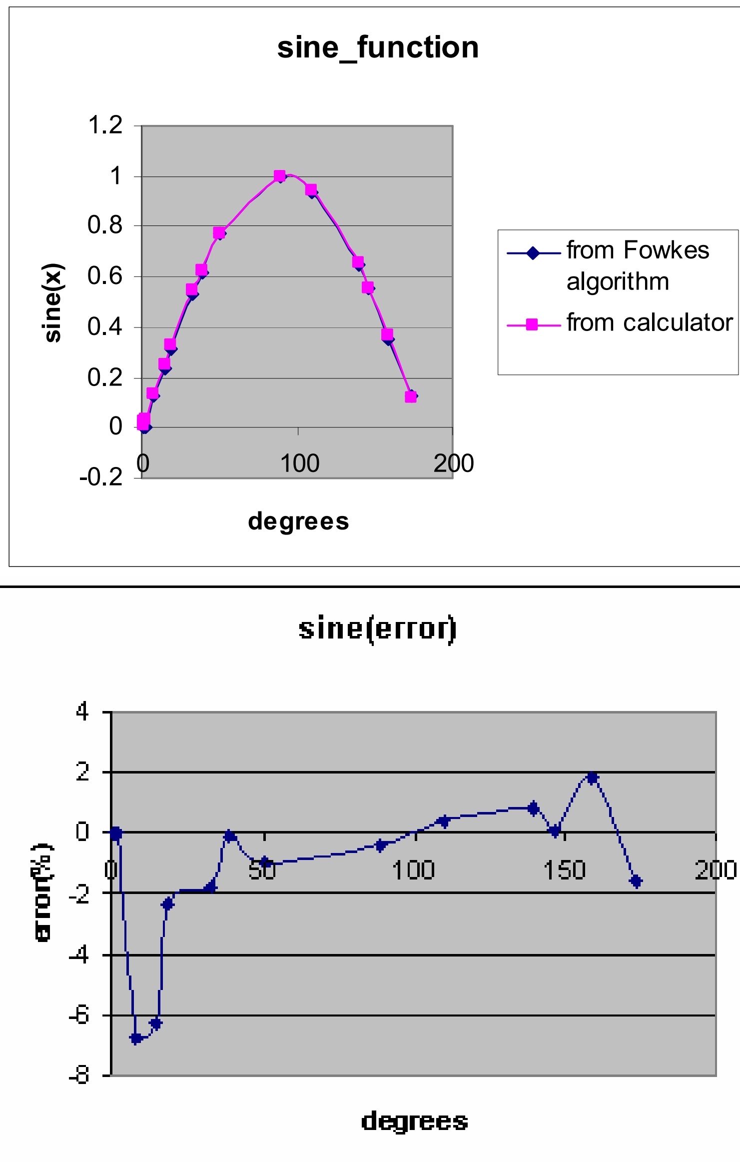


Figure 6.22. Error in sine calculation

6.2.4 Comparison with CORDIC

Comparisons of error in sine values and cos values calculated from Fowkes algorithm with error in CORDIC sine and cosine values is done as shown by the graph in Figure 6.23. Error in CORDIC varies from -65% to 16% in sine and from -11% to 1.5% in cos which is quite much in comparison to the error in algorithm suggested by Fowkes.

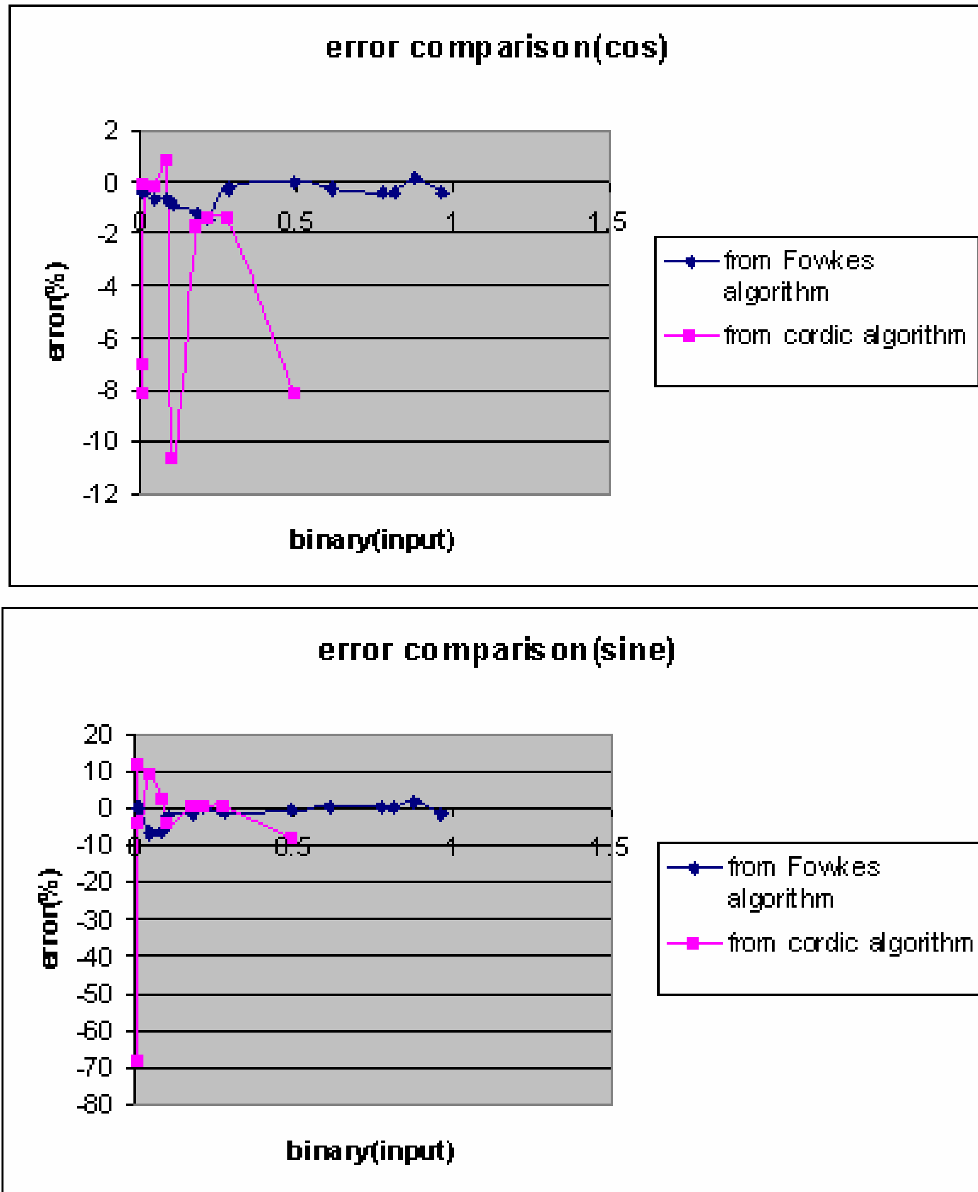


Figure 6.23. Comparison with CORDIC

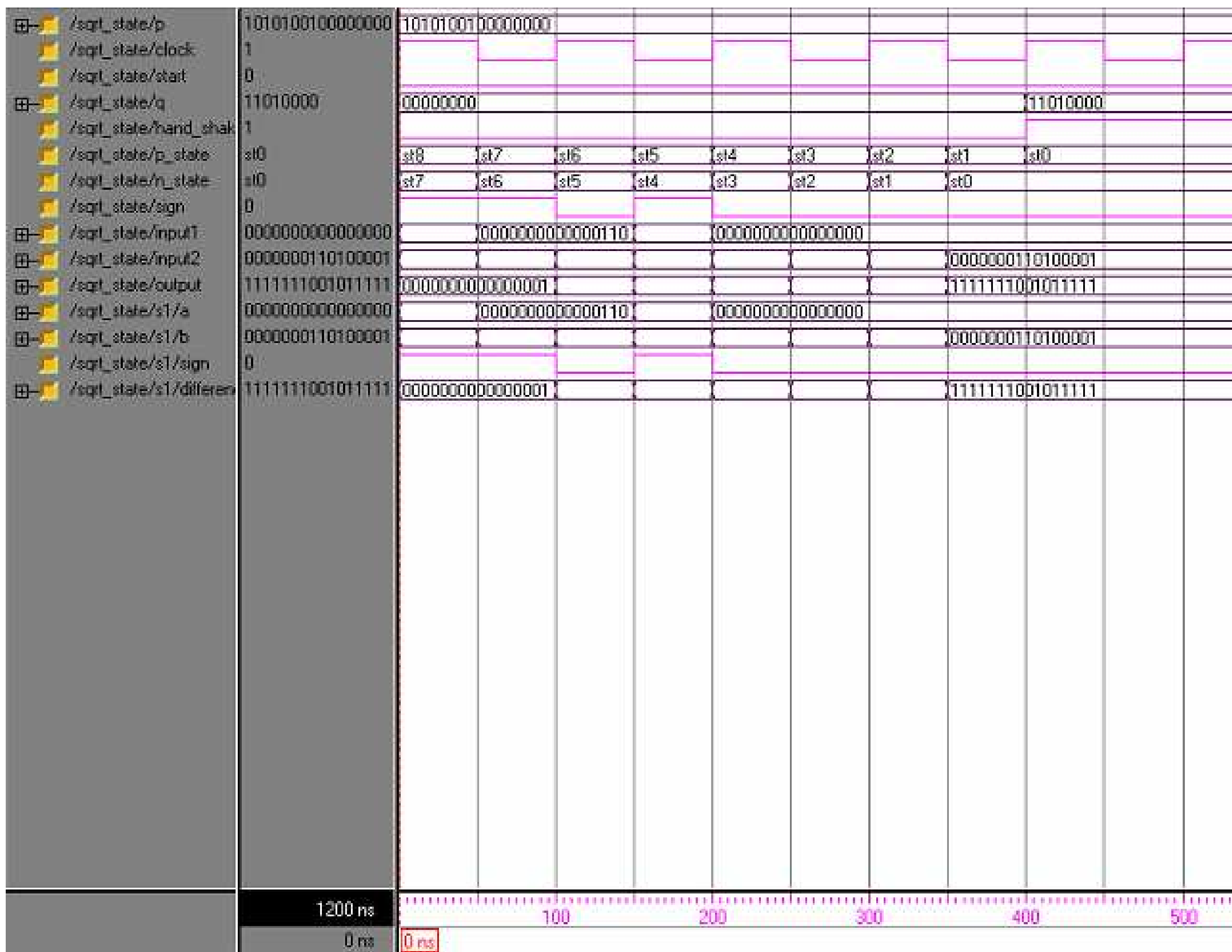


Figure 6.3. Simulation of square root

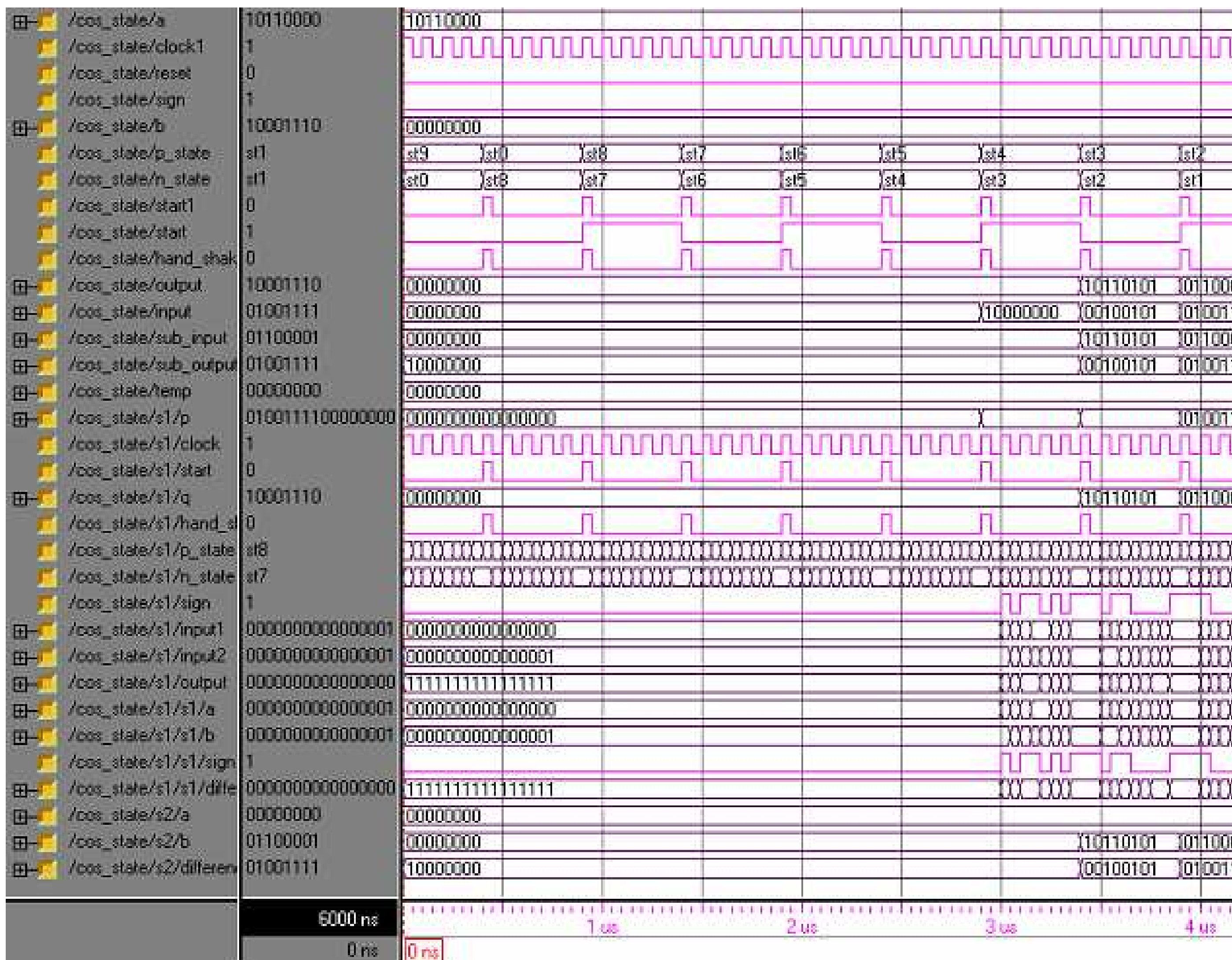


Figure 6.9. Simulation of cosine

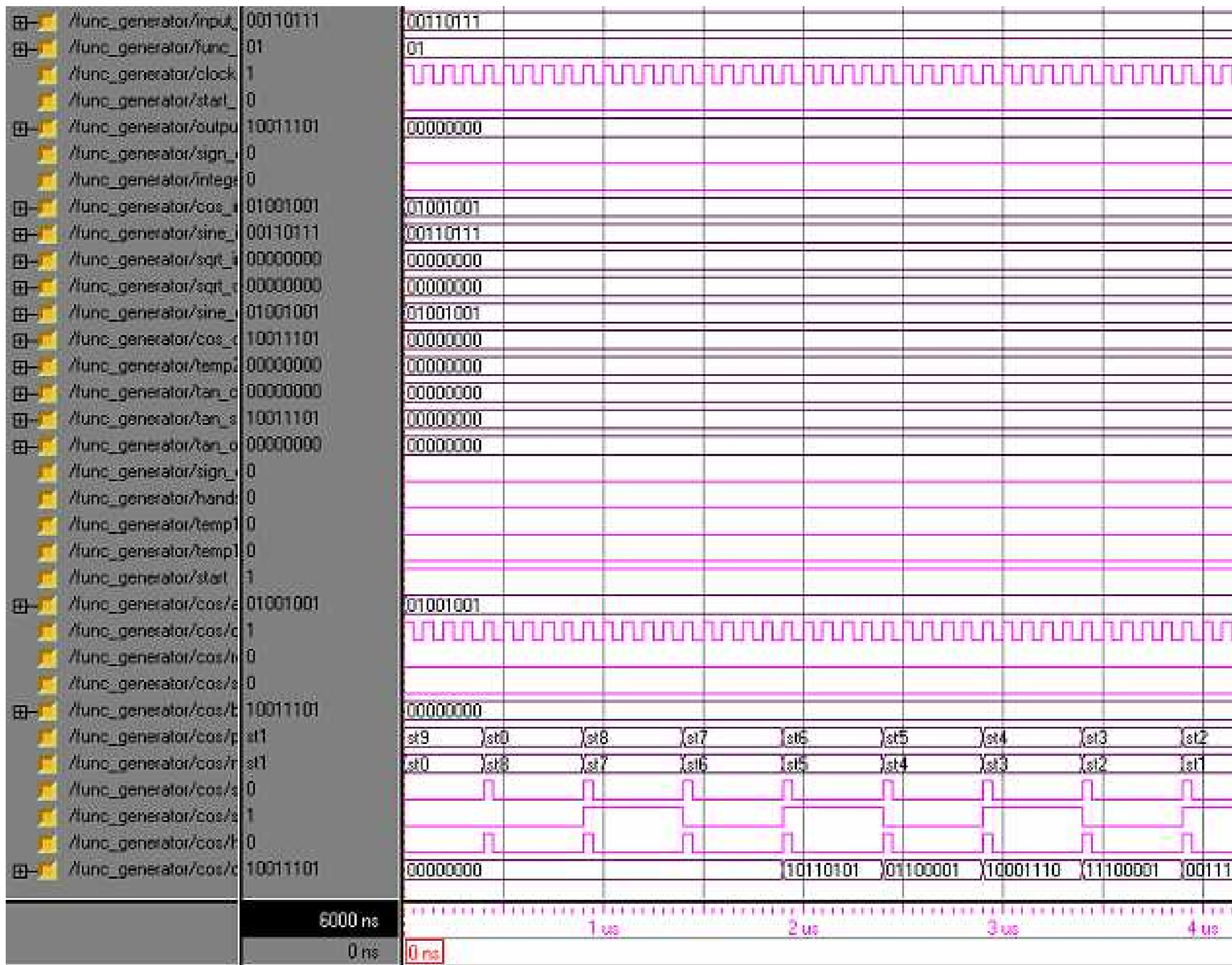
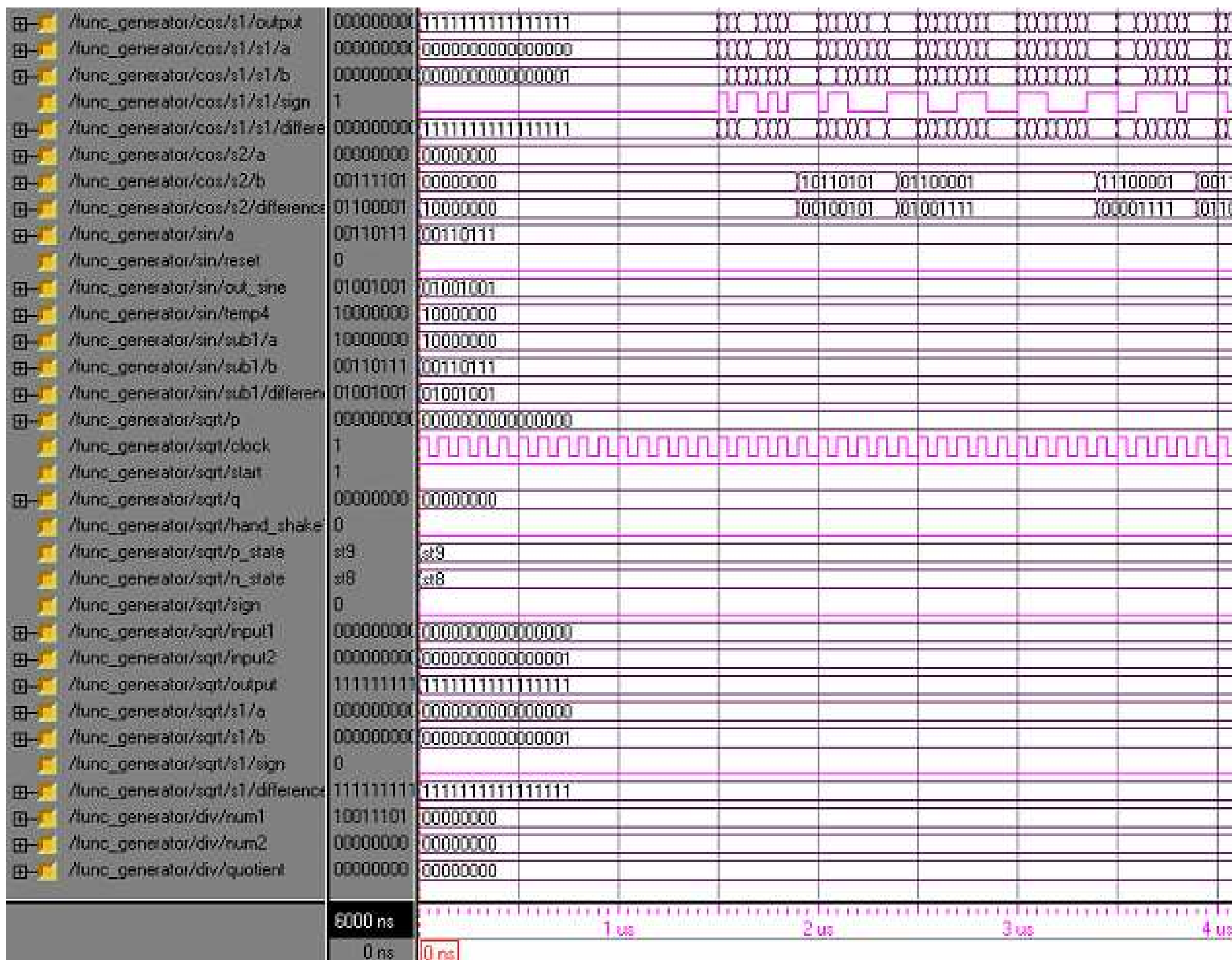


Figure 6.15. Simulation of Function Generator

(Contd….) Figure 6.15. Simulation of Function Generator



(Contd….) Figure 6.15. Simulation of Function Generator

Equation 2

CHAPTER 7
Conclusion And Future Scope

Conclusion

In this thesis a method suggested by Fowkes [23] for calculating some common trigonometric functions is explored and compared with extensively used CORDIC method.
The method is better than CORDIC in many aspects which are:

- More accurate
- Requires relatively little area to implement
- No tables of pre-stored constants

There are some drawbacks also when compared to CORDIC. It has a latency of (n x n) cycles whereas the latency of CORDIC is just n cycles. Its limitations are such that it is not likely to be useful neither in designs which require floating-point arithmetic with large word sizes, nor in cases where a simple ROM mapping scheme would suffice.

This algorithm quality make it ideal for integration as a subsystem in a VLSI circuit, for adding sine and cosine functionality to a design which already requires square root, or for otherwise providing an alternative to the many CORDIC-like methods or other more area-intensive methods which exist.

It has its applications in biomedical such as speech recognition, coding, sound processing and medical imaging.

Future Scope

Currently, our main disadvantage is speed, the speed could be increased if some parallelism could be implemented in it. Future work could be done in extending it for designs which require floating-point.

References

[1] A.A.J.de Lange, A J van der Hoeven, E.F.Deprettere, P.Dewilde, J.Bu, “The Design of a 50MFLOP Arithmetic Chip for Massively Parallel Pipelined DSP Algorithms – The Floating Point Pipeline CORDIC Processor”, 410–414, 2003.

[2] D. A. Patterson, and J. L. Hennessy, “Computer organization and design: The hardwired software interface, Morgan Kauffman, San Francisco, 1994.

[3] Dhananjay S. Phatak, “Double Step Branching CORDIC: A New Algorithm for Fast Sine and Cosine Generation”, IEEE Transactions on computers, vol. 47, no. 5, may 1998.

[4] D. R. Morrison, “A method for computing certain inverse functions”, Math. Tables and Other Ai& to Comp., vol. 10, no. 56, pp. 202-208, 1956.

[5] Elisardo Antelo, Julio Villalba, “Low Latency Pipelined Circular CORDIC”, Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH’05), 2005.

[6] Elisardo Antelo, Tom’as Lang, Javier D. Bruguera, “Very–High Radix CORDIC Vectoring with Scalings and Selection by Rounding”, IEEE, 1999.

[7] I. Flores, “The Logic of Computer Arithmetic”, Englewood Cliffs, N.J., Prentice-Hall, pp. 409-417, 1963.

[8] I. Koren, “Keynote talk”, IEEE Transactions on computers, vol. 54, no. 3, March 2005.

[9] I. Koren, “Computer arithmetic algorithms”, Prentice Hall, Englewood Cliffs, N. J., 1989.

[10] J. Bhasker, “A VHDL primer”, Third edition, Pearson education, 1999.

[11] J. Duprat and J. Muller, “The CORDIC Algorithm: New Results for Fast VLSI Implementation,” IEEE Trans. Computers, vol. 42, no. 2, pp. 920-930, Feb. 1993.

[12] J.E. Volder, “The CORDIC Trigonometric Computing Technique”, IRE Trans. Electronic Computers, vol. 8, pp. 330-334, Sept. 1959.

[13] John L Hennessy and David A Patterson, “Computer architectute and quantitative approach”, Second edition, Morgan Kauffman, San Francisco, 1996.

[14] J.S. Walther, “A Unified Algorithm for Elementary Functions”, Proc. 28th Spring Joint Computer Conf., pp. 379-385, 1971.

[15] M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, J. Barnes, “Developing the WRL3170/3171 SPARC floating-point coprocessors”, IEEE Micro 10:1, 55-64, 1990.

[16] M.D. Ercegovac and T. Lang, “Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD”, IEEE Trans. Computers, vol. 39, no. 6, pp. 725-740, June 1990.

[17] M. Ghairiani, N. Masmoudi, M. W. Khariut, L. Kamoun, “Design and Chip Implementation of Modified CQRDIC Algorithm for Sine and Cosine functions Application: PARK Transformation”, IEEE, December 1998.

[18] Naofumi Takagi, Tohru Asada, and Shuzo Yajima, “Redundant CORDIC Methods with a Constant Scale Factor For Sine and Cosine Computation”, IEEE Transactions on computer, vol. 40, no. 9, September 1991.

[19] Neil Weste and Kamran Eshraghian, “Principles of CMOS VLSI Design: A Systems Perspective”, Second Edition, Pearson Education, New Dehi, 1993.

[20] Peter J. Ashenden, “The designer’s guide to VHDL”, Second edition, Morgan Kauffman, San Francisco, 1996.

[21] Pran kurup, taher abbasi, “Logic Synthesis using Synopsys”, Second edition, Kluwer Academic Publishers, 1997.

[22] Ray Andraka, FPGA '98, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998.

[23] Raymond E. Fowkes, “Hardware efficient algorithms for trigonometric functions”, IEEE Transactions on computers, vol. 42, no. 2, Feb 1993.

[24] R. E. Fowkes, “Minimal hardware algorithms for trigonometric functions’’, in WESCON ‘91 Con$ Rec., pp. 596-600, 1991.

[25] R. Naik, A. Srojcevski, V. Vibhute, I. Singh, “Implementation of Magnitude Estimation Algorithm for Hearing Aid”, IEEE International Workshop on Biomedical Circuits & Systems, 2004.

[26] Shen-Fu Hsiao, Yu-Hen Hu, Tso-Bing Juang, "A Memory-Efficient and High-Speed Sine/Cosine Generator Based on Parallel CORDIC Rotations", IEEE Signal Processing Letters, vol. 11, no. 2, February 2004.

[27] The Quantization Effects of the CORDIC Algorithm, IEEE Trans. Signal Processing, Vol 40, No 4, 834–844, April 1992.

[28] V. Considine, "CORDIC trigonometric function generator for DSP", IEEE-89, International Conference on Acoustics, Speech and Signal Processing, pages 2381 - 2384, Glasgow, Scottland, May 1989.

[29] William Stallings, "Computer Organization and Architecture", Sixth edition, Pearson Education, New Dehi, 2003.

[30] W. J. Cody, J. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J.Palmer, F. N. Ris, and D. Stevenson, "Proposed radix- and word-length-independent standard for floating-point arithmetic", IEEE Micro 4:4, 86-100, 1984.

[31] www.wikipedia.org/wiki/Booth_algorithm, 2000.

[32] www.xilinx.com.

[33] www.peakfpga.com/vhdlref/refguide/language_overview.htm