

STUDY OF RECURSIVE DIVIDE ARCHITECTURES
IMPLEMENTATION FOR DIVISION AND
MULTIPLICATION

By

AMEY P PHADKE

Bachelor of Engineering in Electronics

University of Pune

Pune, India

2007

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2011

STUDY OF RECURSIVE DIVIDE ARCHITECTURES
IMPLEMENTATION FOR DIVISION AND
MULTIPLICATION

Thesis Approved:

Dr. James E. Stine

Thesis Adviser

Dr. Chris Hutchens

Dr. Louis G. Johnson

Dr. Sheryl A. Tucker

Dean of the Graduate College

TABLE OF CONTENTS

Chapter		Page
1.	I
	INTRODUCTION.....	1
2.	R
	REVIEW OF LITERATURE.....	5
	2.1. DIGIT RECURRENCE ALGORITHM	5
	2.2. QUOTIENT SELECTION TABLE.....	7
	2.3. CONTAINMENT CONDITION	8
	2.4. CONTINUITY CONDITION	9
	2.5. ON THE FLY CONVERSION	11
3.	M
	METHODOLOGY	16
	3.1 DIVISION	16
	3.2 RADIX2 DIVISION.....	17
	3.3 RADIX4 DIVISION.....	20
	3.4 RADIX16 DIVISION USING OPERAND SCALING	25
	3.5 RADIX 512 DIVISION	31
	3.5.1 <i>Recoding</i>	34
	3.5.2 <i>Adder</i>	36
	3.6 RECURSIVE DIVIDE/MULTIPLIER UNIT.....	40
4.	R
	RESULTS.....	45
	4.1 DESIGN FLOW:	46
	4.1.1 <i>Design Specifications</i>	47

4.1.2	<i>HDL Coding</i>	48
4.1.3	<i>Logic synthesis</i>	50
4.1.4	<i>Place and Route</i>	52
4.1.5	<i>Post-Layout Analysis:</i>	52

4.2	COMPARISON	53
5.	C
	ONCLUSION	57
	REFERENCES	59
	APPENDICES.....	60

LIST OF TABLES

Table	Page
Table 1 - Truth table for OTF logic	15
Table 2 - Signed Digit Conversion to Normal Representation	15
Table 3 - Truth table for Radix2 QST	19
Table 4 - Quotient bit encoding for Radix2	19
Table 5 - Selection Intervals and mk constants [4]	22
Table 6 – Quotient bit encoding for Radix4	24
Table 7 - Selection multiples for scaling	28
Table 8 - Quotient selection table for Radix16	29
Table 9 - Quotient recoding in Carry-Save format	35
Table 10 - Division cycle timing chart	36
Table 11 - Multiplication cycle timing chart	44
Table 12 - Radix2 Divide Results	53
Table 13 - Radix4 Divide Results	54
Table 14 - Radix16 Divide Results	54
Table 15 - Radix512 Divide Results	54
Table 16 - Radix512 Mult-Div Results	54
Table 17 - Relative area increase	56
Table 18 - Single Precision Comparisons for various multiplier architectures.....	56

LIST OF FIGURES

Figure	Page
Figure 1 - Distribution of FP instructions	2
Figure 2 - Distribution of unit stall time	3
Figure 3 - Robertson's PD Diagram	9
Figure 4 - On the fly conversion	13
Figure 5 - Radix2 hardware block diagram	20
Figure 6 - Hardware block diagram for QST	23
Figure 7 - Radix4 hardware block diagram	24
Figure 8 - Radix16 hardware block diagram	30
Figure 9 - SBTM implementation.....	32
Figure 10 - Block diagram for recoder.....	36
Figure 11 - Radix512 hardware block diagram	38
Figure 12 - Radix512 hardware block diagram using Carry-Save format	39
Figure 13 - Radix512 hardware block diagram for integrated Mult-Div Unit.....	43
Figure 14 - Design Flow	46
Figure 15 - ModelSim simulation snapshot	49
Figure 16 - Place and Route Design Flow	51
Figure 17 - Delay Comparisons	55
Figure 18 - Area Comparisons	55

CHAPTER I

INTRODUCTION

Due to advancement in VLSI technology many industrial and business related applications makes use of floating point operations as a result of which floating point unit has become critical component of modern processor design as well as dedicated embedded systems. Special purpose applications such as graphics rendering has further demanded need of high speed & efficient floating point units as performance of the processor is often limited by high latency of floating point unit.

Modern applications consist of several floating-point operations, which mainly include addition, subtraction, multiplication, division and square root [1]. These days focus is more on the implementation of high speed floating point adders and multipliers as a result of which latency of these operations has come down to 2 to 4 cycles, but at the same time latency associated with division and square root is large, in some cases latency for such operations is around 50-60 cycles. This ignorance towards implementation of high-speed divider is due to instruction mix of the benchmarks, which are used to validate performance of processors.

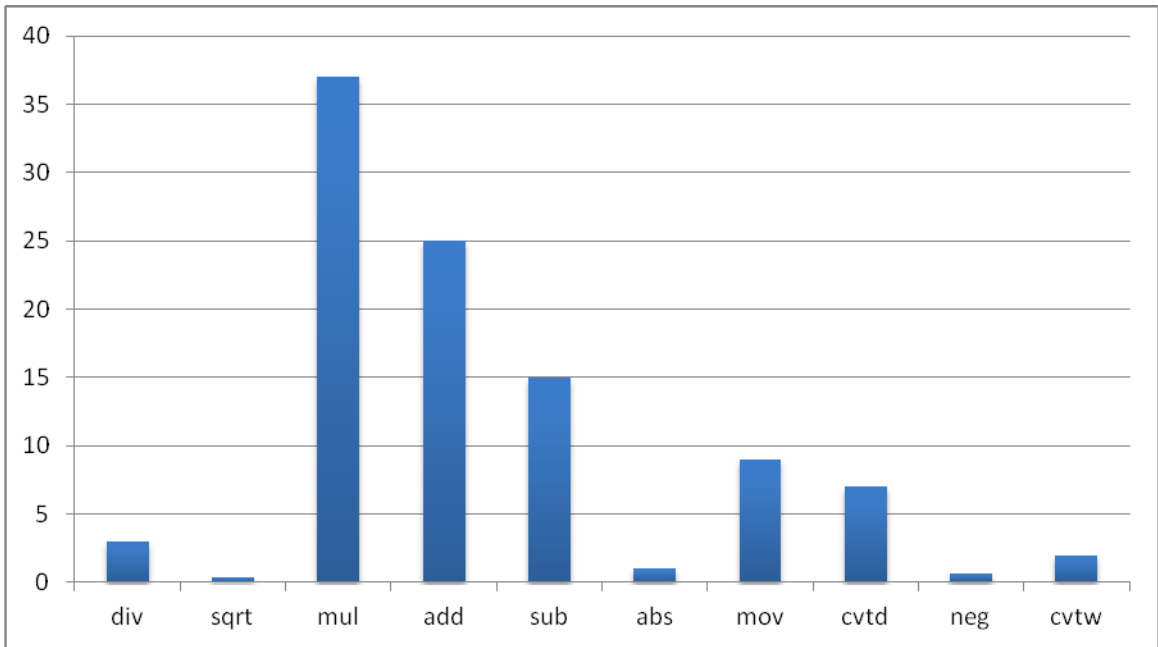


Figure 1 - Distribution of FP instructions

Figure1 shows average frequency of division and square root operations in benchmark suite as compared to other floating-point operations. All benchmark applications are compiled using O3 optimization. From figure1 we can observe that division and square root are very uncommon operations comprising only 3% of total floating point operations [2].

On the other hand floating point multiplication holds for 37% of the instructions & floating point addition accounts for 25% of the instructions. Due to this common view of division operation is that it is an infrequent operation which does not required high priority, but if we look into Figure2 which shows latency associated with floating point operations, it indicates that latency of divide operation is 20 cycle as compared to multiply and addition operations which has 3 cycle of latency each. Distribution of stall time due to FP hardware shows that FP division is responsible for 40% of the latency, FP addition accounts for 42 % and FP multiply results in remaining 18%. This shows that slow division operation results in overall degradation of system performance.

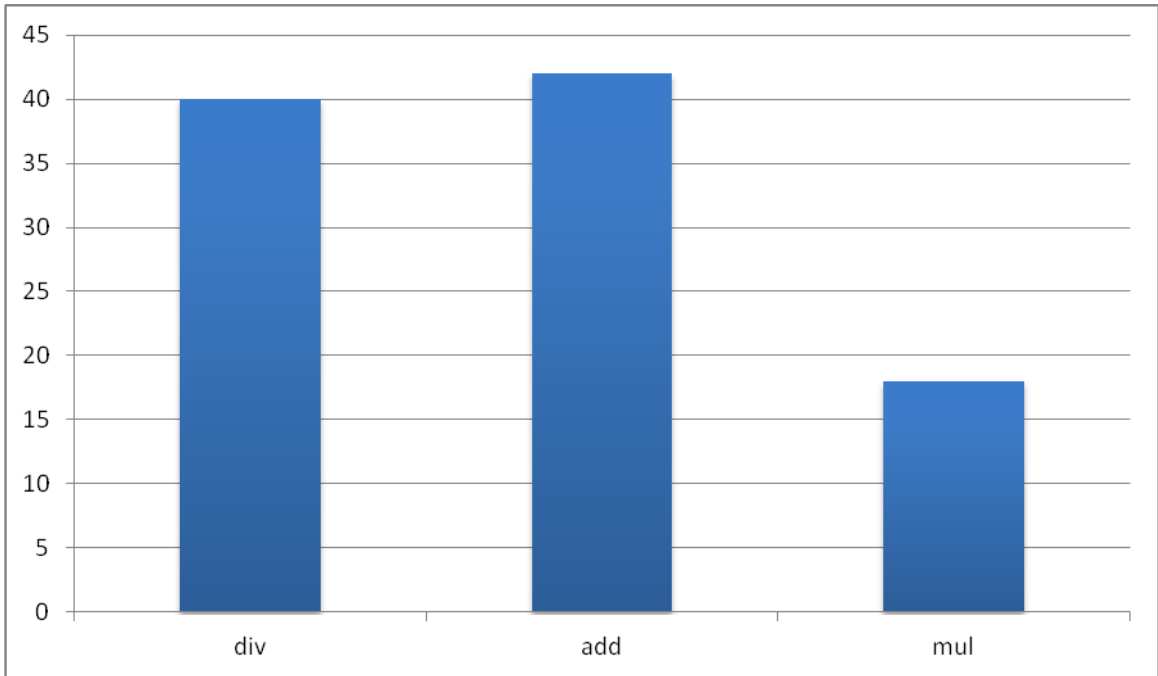


Figure 2 - Distribution of unit stall time

There is an extensive mathematical literature available for implementation of division algorithm but still implementation of these algorithms is considered as black box due to many parameters involved in it. Division algorithms can be divided into five classes such as digit recurrence, functional iteration, very high radix, table look up and variable latency.

Some division algorithms don't belong to any specific class; they make the use of more than one technique mentioned above to perform the operation. Out of algorithms mentioned above digit recurrence is the oldest class of high-speed division as a result of which lot of literature is published related to it. The most common and popular implementation of digit recurrence algorithm is SRT division algorithm, which has been named SRT division by Freiman taking its name from the initials of Sweeney, Robertson and Tocher, who developed the algorithm independently at the same time.

Implementation work related to SRT division algorithms have been done previously without an actual implementation, Ercegovac and Lang provide comparison between different

division algorithms in terms of delay and area using full adder units and NAND gate units. Objective of this work is to provide accurate comparison between different SRT algorithms by actual RTL implementation of all of them, followed by synthesis and PNR of the design in order to get area, delay and power numbers associated with it. This work includes implementation of SRT Radix2, Radix4, Radix16 and Radix512 algorithms. More emphasis is given on the implementation of high Radix512 division algorithms and how the same hardware can be utilized to perform high radix multiplication to save area and maximize performance.

CHAPTER II

REVIEW OF LITERATURE

Digit recurrence algorithm consists of n iterations, where each iteration produces one digit of the quotient. However, digit recurrence algorithms do have preprocessing as well as post processing steps, which are vital in order to, make sure that the dividend, divisor, quotient and remainder are presented correctly [3].

2.1. Digit Recurrence Algorithm

Division operation starts by taking divisor d , dividend x and computes quotient q , and its remainder, r . the basic equation for division is recursive as follows:

$$x = q \cdot d + r \quad \text{where } r < d \quad (2.1.1)$$

The quotient digit is one of the most interesting elements in the division process. In particular, quotients are most often implemented as redundant digit set. The reason behind doing so is that it simplifies the quotient digit selection. Unfortunately as radix increases, complexity of the quotient digit selection also increases.

Therefore as designers one of the challenges for division is to make right decision on which algorithm to chose for implementation and its circuit implications [4]. This tradeoff is noticeable in division due to variation in which division can be implemented.

Digit recurrence algorithm of division works iteratively using following equation where w_i is the partial remainder for iteration i , d is the divisor, r is the radix and q_i is the quotient digit for iteration i

$$w_{i+1} = r \cdot w_i - q_{i+1} \cdot d \quad (2.1.2)$$

Algorithm starts by assigning first partial remainder denoted by $w_{[0]}$ as dividend x . The quotient selection function is chosen based on comparison between the divisor and shifted partial reminder

$$q_{i+1} = \text{QST}(r \cdot w_i, d) \quad (2.1.3)$$

Where QST is the Quotient Selection Table. The QST can be implemented in various ways such as using ROM, PLA or combinational logic. All RTL implementation in this work use combinational logic to construct QST due to an obvious advantage of speed over other techniques as well as in order to avoid use of Macro block in the design.

The most challenging step in the division procedure is the comparison between the divisor and remainder to determine the quotient bit. If this is done by subtracting d from w_i , one has to be careful if the result is negative. If so, a correction operation occurs restoring the remainder to the previous interaction, this method is called restoring division. Non-restoring division is an alternative for sequential division by having specific logic for not correcting the quotient; this is achieved by allowing a correlation factor within the algorithm. Unfortunately, because non-restoring division requires a correction factor, there may be some post processing

that is required if the final remainder is negative. Accordingly, it is necessary to have a correction step that adjusts the quotient as follows where m is the final iteration of the recurrence relation and r^{-n} is an ulp

$$q = q_m \quad \text{if } w_i \geq 0 \quad (2.1.4)$$

$$q_m - r^{-n} \quad \text{if } w_i < 0 \quad (2.1.5)$$

Therefore, the process of division by recurrence can improve upon general division algorithms by taking advantage of floating elements

1. As Radix increases, it decreases the number of iterations assuming $r = 2^k$
2. Redundancy within the quotient digit set reduces and simplifies the QST
3. Partial remainder can be implemented using redundant notation, which simplifies the computation of the partial remainder using carry-free adder.

All the designs implemented in this work make the use of these choices and showing tradeoffs for these choices.

2.2. Quotient Selection Table

This section presents basic theoretical background required to design quotient digit selection function. Algorithm for division is challenging because the implementations for quotient digit selection vary from design to design. The basic idea of the QST is to choose the value of quotient digit q_{i+1} , based on comparison between shifted partial remainder and the divisor. A symmetric SD digit set is utilized where the range of quotient is

$$q_{i+1} \text{ belongs to } \{-a, -a+1, \dots, -1, 0, 1, \dots, a-1, a\}$$

Redundancy factor ρ is defined as

$$\rho = \frac{a}{r-1}, \frac{1}{2} < \rho < 1 \quad (2.2.1)$$

Even though choosing the right function for a QST is complex, it can be easily formulated into two conditions called containment and continuity. The containment condition identifies selection interval for each quotient digit q_{i+1} . On the other hand continuity condition details the range, which the quotient digit is selected.

2.3. Containment condition

Since the equation for recurrence involves subtraction and shifts from equation 2.1.2, it is important to note that the quotient digit selection becomes difficult. For example, if user decided to divide $300 / 5$ in radix 10 and chooses q_i to be 2, this will violate the bound available for next quotient digit making the computation cumbersome. In other words a quotient digit will need to be calculated for partial remainder of 200.

The containment condition sets up the selection intervals necessary for computing the subsequent quotient digit. For given quotient digit q_{i+1} to be chosen as k , there should be bounds on an interval of allowable partial reminders, these regions are defined by the interval $[L_k, U_k]$ such that L_k is the smallest value of partial remainder $r.w_j$ for which it is possible to choose $q_{i+1} = k$, whereas U_k is largest value of partial remainder $r.w_j$ for which it is possible to choose $q_{i+1} = k$.

In other words selection intervals are for quotient digit $q_{i+1} = k$ is given by

$$U_k = (k + \rho) \cdot d \quad (2.3.1)$$

$$L_k = (k - \rho) \cdot d \quad (2.3.2)$$

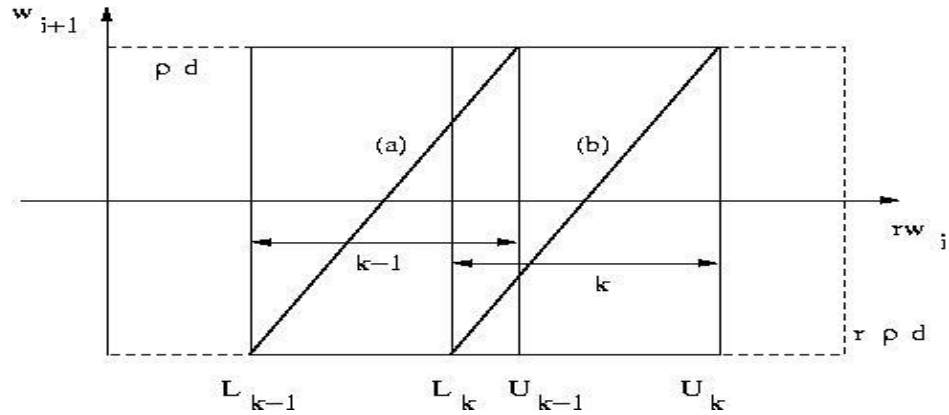


Figure 3 - Robertson's PD Diagram

Sometimes, this can be visualized by examining a graph of subsequent partial remainder, w_{i+1} versus the shifted partial remainder, $r \cdot w_i$. This visualization is represented in Robertson's diagram as shown in figure3. Robertson's plot the recurrence relationship for a given quotient digit q_{i+1} , assuming the user is varying the shifted partial remainder and plotting or computing the subsequent partial remainder. The axis of Robertson's diagram is bounded by axis ($[-rp.d ; rp.d;=p.d;p.d]$) where the axis function defines the range of function such that the argument is defined as $\{[x_{min}; x_{max};y_{min}; y_{max}]\}$. Interestingly, the redundancy introduced by using SD digit set imposes an overlap between quotient digits. For example, in figure3 there is an overlap between $q_{i+1} = k-1$ and $q_{i+1}=k$. This overlap will be useful in defining the continuity equation.

2.4. Continuity Condition

Since the containment condition defines range of subsequent partial remainder, choosing the correct quotient digit from this region is job of continuity condition. To satisfy the containment condition, the minimum value of x-axis of the Robertson's diagram is chosen such that $q_{i+1}=k$ is our quotient digit. This can be defined as the following inequality where s_k is the minimum value that user chooses before an implementation is devised.

$$L_k \leq s_k \leq U_k \quad (2.4.1)$$

Unfortunately, because the overlap that occurred in the containment condition, the quotient digit may be chosen as either k or $k-1$. For example, in figure 3 an overlap exists between L_k , which is Lower bound for quotient digit k , and U_{k-1} , which is the largest bound on partial remainder for a quotient digit to be $k-1$ such that s_k can either be $k-1$ or k . Since the containment equations are defined, it is easy to measure this overlap as

$$U_{k-1} - L_k = (k - 1 + \rho \cdot d) - (k - \rho \cdot d) = (2 \cdot \rho - 1) \cdot d \quad (2.4.2)$$

The simplest selection function is to make s_k constant and do a comparison on the constant. Thus many implementations for QST's resort to ROM or PLA elements. The constants should satisfy the following equation

$$\max(L_k) \leq m_k \leq \min(U_{k-1}) + ulp$$

A second requirement of the selection in the continuity condition states that every value of $r.w[i]$ must belong to at least one selection interval. This can be expressed as

$$U_{k-1} \geq L_k - r^{-n} \quad (2.4.3)$$

As shown in Robertson's diagram the term r^{-n} reflects granularity of representable values, since $r.w[i]$ is represented with n fractional digits. To simplify the derivations and to be independent of iteration i , the more conservative bound is

$$U_{k-1} \geq L_k$$

By substituting the values of bound in above equation we get

$$k - 1 + \rho \cdot d \geq k - \rho \cdot d \quad (2.4.4)$$

which results in $\rho \geq 2^{-1}$

This proves that minimum bound on the redundancy factor ρ is 1/2

2.5. On the fly conversion

The use of the redundant representation complicates the use of SRT division. In most radix system that most digital devices employ, the digit set is restricted to 0, ..., r-1. One of the benefits of using SD number system is that it simplifies the QST. Although SD numbering is useful, it unfortunately is cumbersome to convert from SD notation back to a conventional representation.

Fortunately division algorithm implemented in this work calculates quotient bit with Most Significant Digit First. Arithmetic performed in this fashion is sometimes referred as online arithmetic. Since quotient is calculated as a fraction

$$q_i = \sum q_m r^{-m} \quad (2.5.1)$$

Therefore, using the correction factor and plugging it into the equation above results in the following form

$$q_{i+1} = q_i + q_{i-1} r^{-(i+1)} \quad q_{i+1} \geq 0 \quad (2.5.2)$$

$$q_{i+1} = q_i - r^{-j} + (r - |q_{i+1}|) r^{-(i+1)} q_{i+1} < 0 \quad (2.5.3)$$

The later equation is formed since the quotient for that iteration is negative; hence subtraction is required for the conversion. If we substitute a variable for the correction factor, qm_i , the equation shown above is presented more efficiently as

$$q_{i+1} = q_i + q_{i-1} r^{-(i+1)} \quad q_{i+1} \geq 0 \quad (2.5.4)$$

$$q_{i+1} = qm_i + (r - |q_{i+1}|) r^{-(i+1)} \quad q_{i+1} < 0 \quad (2.5.5)$$

With simple manipulation we can also convert the equation above into an equation for qm_i such that $qm_i = q_i - r^n$. In other words, if the final remainder is negative, subtraction of an *ulp* from the quotient is performed to adjust the correction factor. Then qm_i is calculated as follows

$$qm_i = q_i - (q_{i+1} - 1)r^{-(i+1)} \quad q_{i+1} > 0 \quad (2.5.6)$$

$$qm_i = qm_i - (r - 1 - |q_{i+1}|)r^{-(i+1)} \quad q_{i+1} \leq 0 \quad (2.5.7)$$

Fortunately there is an easy algorithm to convert back redundant quotient into conventional representation for online algorithms. It is called as on-the-fly conversion. The basic idea behind on the fly conversion is to produce the conversion as the digits of the quotient are produced by performing a concatenation instead of any carries or borrows within a carry propagate adder. One element keeps tracks of the quotient, whereas other element keeps track of normal quotient-*ulp*.

Since on-the-fly conversion involves concatenations, the MSDF enables the appropriate quotient digit to be converted by simple combinational logic and shifting as opposed to utilizing CPA. The algorithm can be summarized as follows

$$q_{i+1} = \{q_i, q_{i+1}\} \quad \text{if } q_{i+1} \geq 0 \quad (2.5.8)$$

$$q_{i+1} = \{qm_i(r - |q_{i+1}|)\} \quad \text{if } q_{i+1} < 0 \quad (2.5.9)$$

&

$$qm_i = \{q_{i+1} - 1\} \quad \text{if } q_{i+1} > 0 \quad (2.5.10)$$

$$qm_i = \{qm_i - (r - 1 - |q_{i+1}|)\} \quad \text{if } q_{i+1} \leq 0 \quad (2.5.11)$$

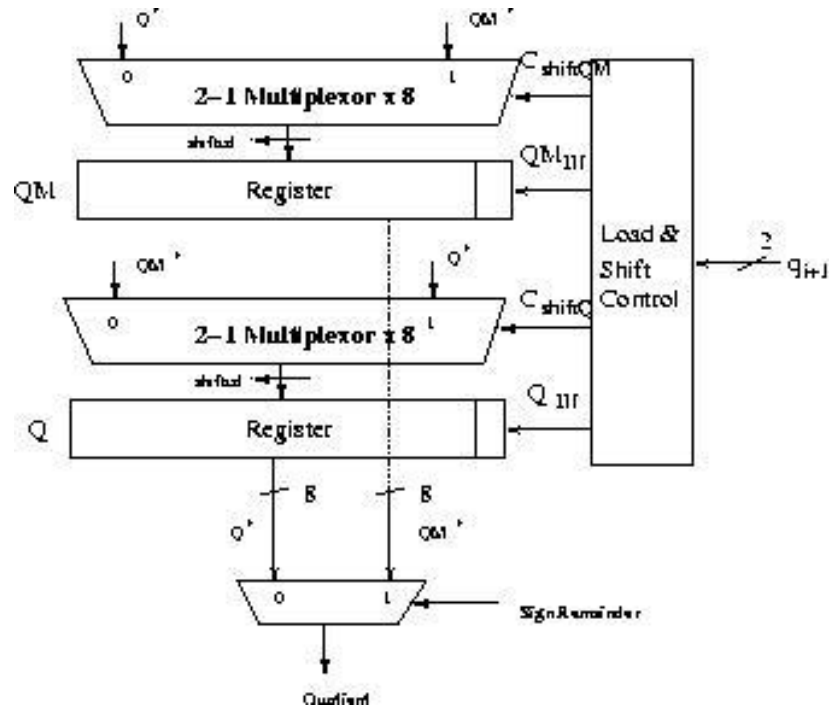


Figure 4 - On the fly conversion

In order to implement on-the-fly conversion, it requires 2 registers to hold q_i and qm_i . These registers are shifted one digit left with insertion into the least-significant digit, depending on the value of q_{i+1} . In other words, depending on what the subsequent quotient digit, the register either chooses q or qm and concatenates the current converted quotient digit into the least significant digits. The figure4 shows on the fly conversion technique implemented for Radix2 division. In all the implementations, structure of on-the-fly technique remains the same except number of bits of quotient shifted in every iteration changes; number of bits shifted into 2 registers depends on the radix of the divider.

Two multiplexers are utilized to select either q or qm and combinational logic is used to select q_{in} or qm_{in} . In order to handle shifting after every cycle, the outputs of the multiplexor are shifted by one (multiplied by 2, which changes for other radix dividers) and either q_{in} or qm_{in} are inserted into the least significant bit during each load.

The final multiplexer choose the correct quotient once the final remainder is known. If the sign of final remainder is 1, it will choose q_m since this register contains the proper corrected quotient. Finally q_{star} and q_{mstar} are designated to q_{i+1} and $q_{m_{i+1}}$ respectively.

For Radix2 on the fly conversion, the registers are updated according to the table1. The values in this table are computed utilizing the equations above for q_m and q_i . The quotient digit set used for this example is $\{-1, 0, 1\}$. The values of C_{shiftq} and $C_{shiftqm}$ are used to control the multiplexers. The values of q_{in} and q_{min} is the concatenation element input into the register. The quotient is utilized as input to compare C_{shiftq} , $C_{shiftqm}$, q_{in} and q_{min} . In order to simplify the logic, the quotient utilizes one hot logic encoding.

(One hot logic is chosen to simplify the digital logic design; this doesn't mean it is mandatory to use the same technique)

Boolean two-level simplified expressions for control signal are as follows

$$\begin{aligned}
 C_{sh\ i} &= q_{i+1}[0] \\
 C_{sh\ i} &= q_{i+1}[1] \\
 q_i &= q_{i+1}[0] + q_{i+1}[1] \\
 q_m &= (q_{i+1}[0] + q_{i+1}[1])
 \end{aligned}$$

For example suppose conversion is required for the following Signed Digit number 11010 to a conventional representation using on-the-fly conversion. Table2 shows the same example. At step $i=0$, the values of both registers are reset which is accomplished by using a flip-flop with reset capabilities. In addition since division is online algorithm, on-the-fly conversion works from most significant bit to last significant bit. The last value in the register is the final converted value assuming a fractional number for q_i and q_m , which is 0.78125 and 0.77334375 respectively

It should be obvious that both these numbers are one *ulp* from each other (i.e. an *ulp* is 2^{-7} or 0.0078125) and 0.78125 is the conventional representation.

q_{i+1}	q_{in}	C_{shiftq}	q_{i+1}	qm_{in}	$C_{shiftqm}$	qm_{i+1}
1	1	1	$\{q_i, 1\}$	0	0	$\{q_i, 0\}$
0	0	1	$\{q_i, 0\}$	1	1	$\{qm_i, 1\}$
-1	1	0	$\{qm_i, 1\}$	0	1	$\{qm_i, 0\}$

Table 1 - Truth table for OTF logic

i	q_i	q	qm
0	-	0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	0	0.110	0.101
4	1	0.1101	0.1100
5	-1	0.11001	0.11000
6	0	0.110010	0.110001
7	0	0.1100100	0.1100011

Table 2 - Signed Digit Conversion to Normal Representation

CHAPTER III

METHODOLOGY

In this section initially we describe how lower radix divider were built and problems associated with the techniques involved in it. It is followed by an explanation of operand scaling technique, and its implementation in higher radix division

3.1 Division

The division algorithm performs division between two double precision floating point numbers x and d , that produces quotient q

$$q = \frac{x}{d} \quad (3.1.1)$$

In the algorithm described in this work only the mantissa is calculated since sign and exponent calculation is pretty straightforward. The ranges of the operands are

$$\frac{1}{2} \leq d \leq 1 \quad (3.1.2)$$

$$\frac{1}{2} \leq x \leq 1 \quad (3.1.3)$$

So for the quotient we have -

$$\frac{1}{2} \leq q \leq 1 \tag{3.1.4}$$

The IEEE double precision standard defines a format of 64 bits for this representation. These 64 bits are divided into 3 fields sign, exponent and mantissa consists of 1, 11 and 52 bits respectively. Because number in this format are normalized with the mantissa in the range $1 < x < 2$ with most significant bit is always 1 and can be omitted. Thus real mantissa consists of 52 bit with 1 in most significant position, giving total of 53 bits, as our algorithms requires both the operands to be in the range of $[1/2, 1]$ we divide the IEEE mantissa by 2 and increase their exponent respectively. This is similar to shifting the mantissa right by one bit, algorithm also requires $d > x$ so when this condition is not satisfied mantissa is further divide by 2 due to which x looks like: $x = 0.0x_{[-1]}x_{[-2]}x_{[-3]} \dots x_{[-53]}$ number of bits of both operands are extended by 1 bit from 53 to 54 bits to make sure it don't lose the precision due to right shift [4].

3.2 Radix2 Division

The Radix2 division SRT algorithm is most easy to implement. It produces one bit of quotient every iteration, requiring 25 or 54 clock cycles for single and double precision floating point respectively. This algorithm is an extension of non-restoring division with a quotient digit set of $\{-1, 0, 1\}$. The equation used for this division is as follows

$$w_{i+1} = 2 \cdot w_i + q_{i+1} \cdot d \tag{3.2.1}$$

SRT division was named after Sweeney, Robertson and Touher. Main objective of this algorithm is to speed up the division by allowing 0 as a quotient digit. This eliminates need of subtraction or addition when the value of quotient selected is 0

Now we will derive the containment and continuity conditions for Quotient Selection table. Using the equation of containment and $\rho = 1$, the following condition exists

$$L1 = 0 \quad U1 = 2 \cdot d \quad (3.2.2)$$

$$L0 = -d \quad U0 = d \quad (3.2.3)$$

$$L-1 = -2d \quad U-1 = 0 \quad (3.2.4)$$

To maximize the selection of quotient digit with value 0, most straight forward quotient selection table

$$q_{i+1} = 1 \text{ if } d < 2w[i] \quad (3.2.5)$$

$$q_{i+1} = 0 \text{ if } -d < 2w[i] \leq d \quad (3.2.6)$$

$$q_{i+1} = -1 \text{ if } -2d < 2w[i] \leq -d \quad (3.2.7)$$

From the equation expressing number of bits of divisor required for QST we get

$$2^{-\delta} \leq \frac{2\rho-1}{2(a-\rho)} = \frac{1}{2(1-1)} = \infty$$

Consequently, a single set of selection constants can be used for whole range of divisor (it means quotient selection is independent of value of the divisor. So the equation becomes

$$q_{i+1} = 1 \text{ if } \frac{1}{2} < 2w[i] \quad (3.2.8)$$

$$q_{i+1} = 0 \text{ if } -\frac{1}{2} < 2w[i] \leq \frac{1}{2} \quad (3.2.9)$$

$$q_{i+1} = -1 \text{ if } 2w[i] < -\frac{1}{2} \quad (3.2.10)$$

So we require 3 bits of partial remainder to find the quotient as sign bit, integer bit and first fraction bit. Truth table for implementation is shown in Table3

Sign	Int	f0	Result	Quotient
0	0	0	$<1/2$	0
0	0	1	$\geq 1/2$	1
0	0	0	$\geq 1/2$	1
0	0	1	$\geq 1/2$	1
0	1	0	$<-1/2$	-1
0	1	1	$<-1/2$	-1
0	1	0	$<-1/2$	-1
0	1	1	$\geq 1/2$	0

Table 3 - Truth table for Radix2 QST

Each quotient bit is recoded into 2 bits as q_+ and q_- to simplify logical implementation.

Hardware implementation of Radix2 divide is shown below in figure5

Steps involved in Radix2 SRT divide are as follows

- On reset operand x is loaded in. 2 bits are added to it at the start which serves as sign and integer bit. It acts as first partial remainder $w[0]$
- Partial remainder is shifted by 2 every iteration
- First 3 bits of partial remainder identifies correct quotient which is recoded into 2 bits as q_+ and q_-
- Selected quotient chooses appropriate value of d from $\{d, 0, -d\}$
- When $-d$ is selected multiplexor chooses inverted version of d & additional 1 required to produce its 2's complement is added as carry input to the CPA adder.
- Redundant quotient bit is sent to OTF (On The Fly) converter.
- When algorithm requires input operands to be of size greater than double precision, delay due to carry propagate adder becomes the critical path, CPA can be replaced by carry save format adder.

Quotient	q_+	q_-
0	0	0
1	0	1
-1	1	0

Table 4 - Quotient bit encoding for Radix2

Detailed RTL implementation of Radix2 divide is attached in appendix A.

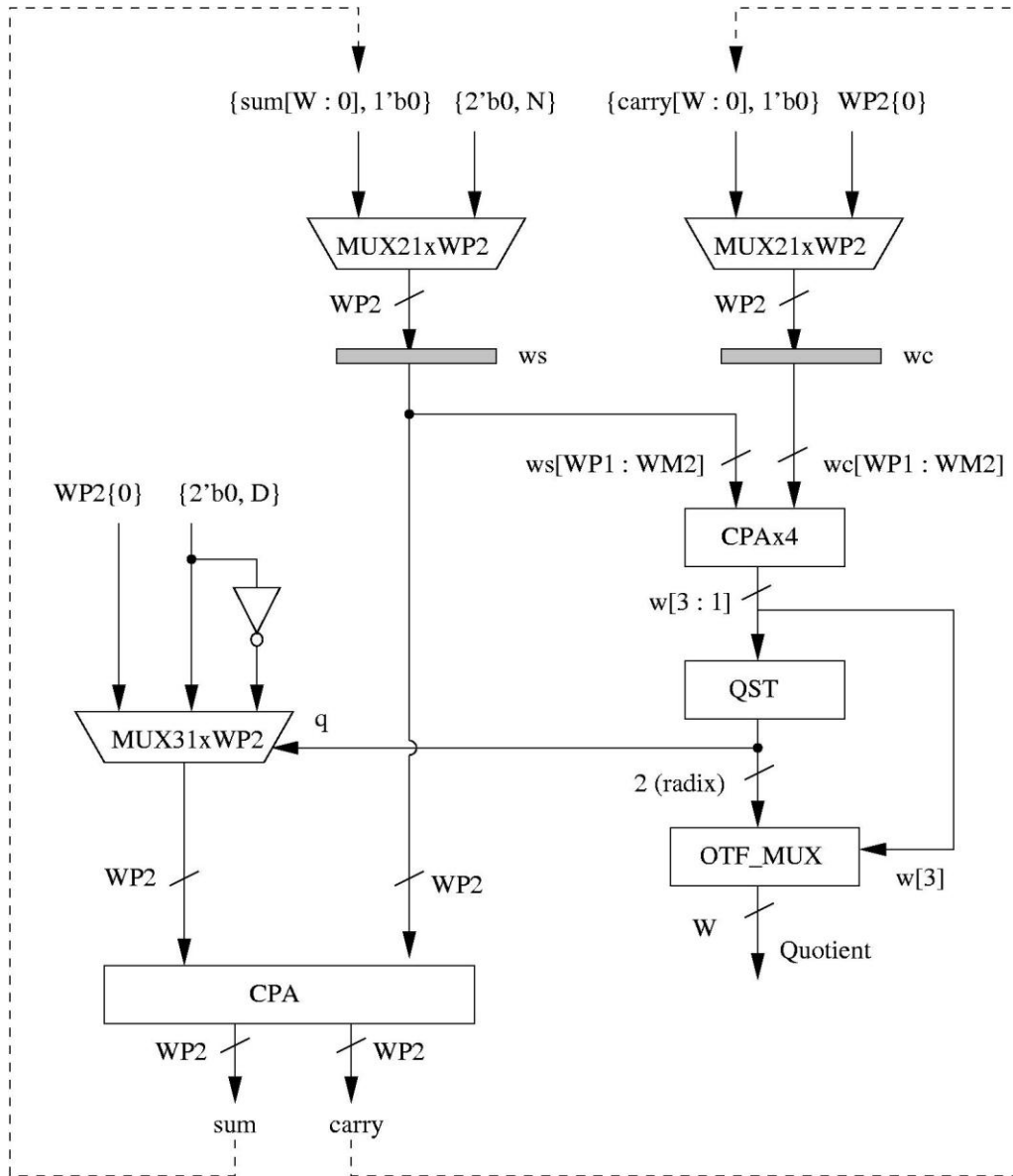


Figure 5 - Radix2 hardware block diagram

3.3 Radix4 Division

In Radix4 divider algorithm two possibilities exist for redundant quotient digit set with digit set can be $\{-2, 1, 0, 1, 2\}$ or $\{-3, -2, -, 1, 0, 1, 2, 3\}$. The case with $a=2$ has an advantage that

multiple of d required are easy to generated whereas with $a=3$ multiple of $3d$ is required which can not be generated using shifting of operand d and it needs to be split into $(d$ and $2d)$ which requires an extra adder in the critical path.

But for case $a = 3$ from equation providing condition on number of bits of divisor

$$2^{-\delta} \leq \frac{2^{\rho-1}}{2(a-\rho)} = \frac{2(1-1)}{2(3-1)} = \frac{1}{4} \quad (3.3.1)$$

Number of bits of the divisor required are 2 as compared to the case with $a= 2$ where bits of divisor required are 3.

$$2^{-\delta} \leq \frac{2^{\rho-1}}{2(a-\rho)} = \frac{2(2/3)-1}{2(2-1)} = \frac{1}{8} \quad (3.3.2)$$

Again similar to containment and continuity conditions derived for radix2, bounds on partial remainder in order to select correct quotient digit set are as follows

$$U_2 = 8/3.d \quad L_2 = 4/3.d$$

$$U_1 = 5/3.d \quad L_1 = 1/3.d$$

$$U_0 = 2/3.d \quad L_0 = -2/3.d$$

$$U_{-1} = -1/3.d \quad L_{-1} = -5/3.d$$

$$U_{-2} = -4/3.d \quad L_{-2} = -8/3. d$$

As shifted partial remainder can reach maximum value of $8/3$, it requires 2 bits of integer, 1 bit of sign and 4 bits of fractions, In addition to this 3 bits divisor are required to identify correct quotient digit.

Table5 shows how value of quotient chosen based on partial remainder and bits of divisor.

Table is derived from Roberson's diagram for the continuity relationship [5]

$[d_i, d_{i+1})^a$	[8,9)	[9, 10)	[10,11)	[11,12)
$L_2(d_{i+1}), U_1(d_i)^b$	36, 40	40,45	44,50	48,55
$m_2(i)$	3/4	7/8	1	1
$L_1(d_{i+1}), U_0(d_i)^b$	9, 16	10, 18	11,20	12, 22
$m_1(i)$	1/4	1/4	1/4	1/4
$L_0(d_{i+1}), U_{-1}(d_i)^b$	-16, -9	-18, -10	-20, -11	-22, -12
$m_0(i)$	-1/4	-1/4	-1/2	-1/2
$L_{-1}(d_{i+1}), U_{-2}(d_i)^b$	-40, -36	-45, -40	-50, -44	-55, -48
$m_{-1}(i)$	-3/4	-7/8	-1	-1
$[d_i, d_{i+1})^a$	[12,13)	[13, 14)	[14,15)	[15,16)
$L_2(d_{i+1}), U_1(d_i)^b$	52, 60	56, 65	60, 70	48,55
$m_2(i)$	5/4	5/4	1	1
$L_1(d_{i+1}), U_0(d_i)^b$	13, 24	14, 26	15, 28	16, 30
$m_1(i)$	1/2	1/2	1/2	1/2
$L_0(d_{i+1}), U_{-1}(d_i)^b$	-24, -13	-26, -14	-28, -15	-30, -16
$m_0(i)$	-1/2	-1/2	-1/2	-1/2
$L_{-1}(d_{i+1}), U_{-2}(d_i)^b$	-60, -52	-65, -56	-70, -60	-75, -64
$m_{-1}(i)$	-5/4	-5/4	-5/4	-3/2

Table 5 - Selection Intervals and mk constants [4]

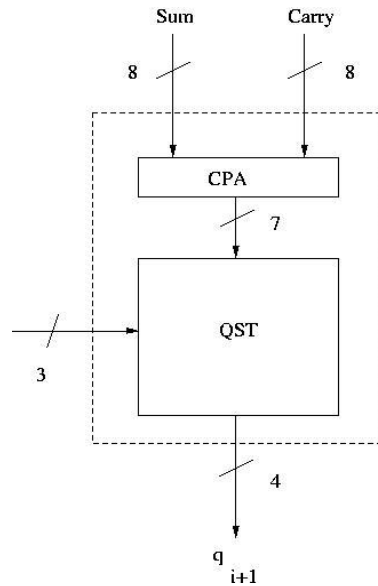


Figure 6 - Hardware block diagram for QST

Steps involved in Radix4 SRT divide are as follows

- a. On reset operand x is loaded. 3 bits are added to it at the start which serves as sign and two integer bits. It acts as first partial remainder $w[0]$
- b. Partial remainder is shifted by 4 (left shift by 2) every iteration & producing 2 bits of the quotient according to recurrence formula.
- c. First 7 bits of partial remainder and 3 bits of divisor identifies correct quotient which is recoded into 4 bits as q_{2+} , q_{+} , q_{2-} and q_{-} as shown in table6.
- d. Selected quotient chooses appropriate value of d from $\{2d, d, 0, -d, -2d\}$
- e. When $-d$ or $-2d$ is selected multiplexer chooses inverted version of d & additional 1 required to produce its 2's compliment is added as carry input to the CPA adder.
- f. Redundant quotient bit is sent to OTF (On-The-Fly) converter.
- g. When algorithm requires input operands to be of size greater than double precision, delay due to carry propagate adder becomes the critical path, CPA can be replaced by carry save format adder

Quotient	q^{2^+}	q^+	q^-	q^{2^-}
-2	0	0	0	1
-1	0	0	1	0
0	0	0	0	0
1	0	1	0	0
2	1	0	0	0

Table 6 – Quotient bit encoding for Radix4

Detailed RTL implementation of Radix4 divide is attached in appendix A. We observed that as we jump from Radix2 to Radix4 complexity of the Quotient Selection Table increase as no of inputs bits to increases from 3 to 10. As calculation of quotient from QST is in critical path of the circuit, which in turn decides the frequency at which divider unit can work. Hardware implementation of Radix4 divide is shown in Figure 7.

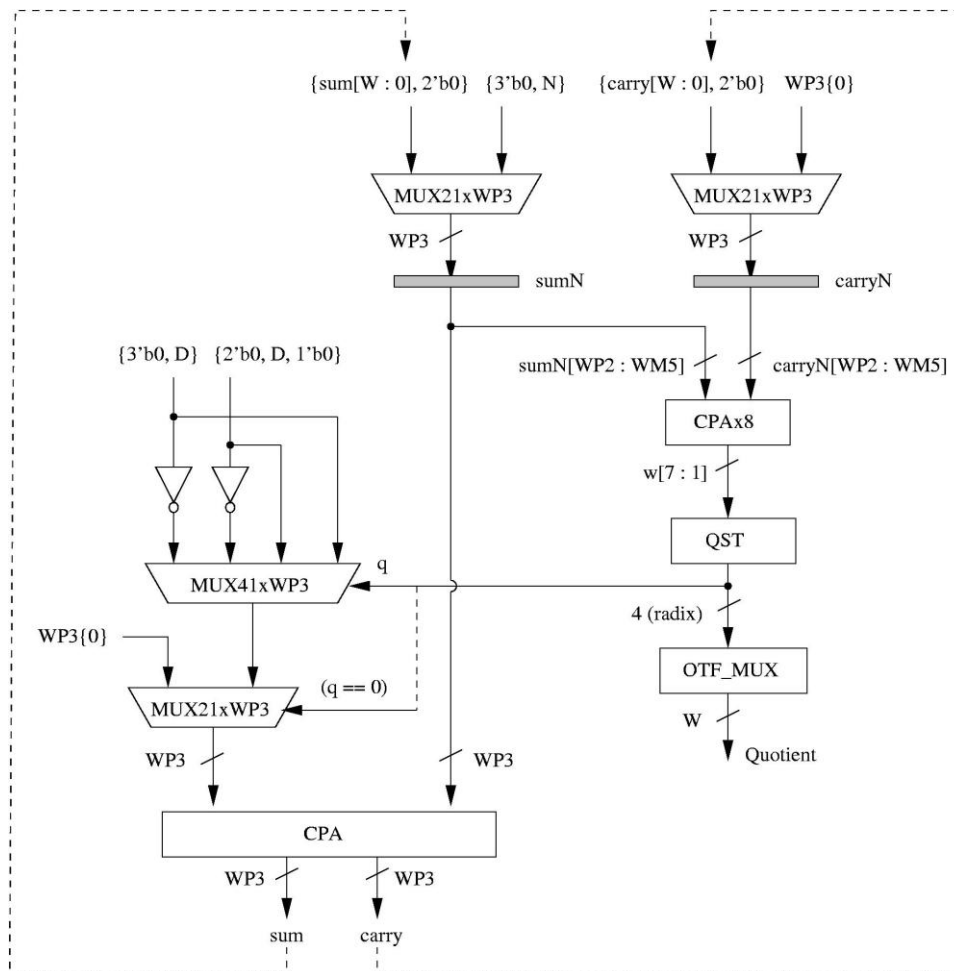


Figure 7 - Radix4 hardware block diagram

3.4 Radix16 division using operand scaling

From the analysis of Radix2 & Radix4 division algorithm it is apparent that the quotient digit selection is more costly to implement as we make jump from Radix2 to Radix4. We have seen that even Radix4 decreases number of clock cycles by half as compared to Radix2, this advantage gets nullified by the fact that quotient selection table becomes more complex resulting in increase in cycle time, this trend continues as we move to higher radix division, which can be proved from substituting the value of redundancy factor and quotient digit set in the equation from section 3.2 & 3.3, for e.g. Radix8 number of bits of partial remainder and divisor required are 9 and 5 bits respectively.

We need way of speeding up the algorithm; one of the ways of speeding up the algorithm is to use prediction of quotient digit. This allows simultaneous calculation of quotient as well as residual, but problem associated with this technique is that this prediction requires even more complex quotient selection table, making this approach impractical to use unless the method is simplified.

One way of simplifying this approach is to reduce the complexity of quotient selection table is by restricting the range of divisor. Since the overlap is largest close to $d = 1$, it is convenient to restrict the range of divisor close to 1. Moreover in order to preserve the precision of quotient both operands need to prescale [6].

It is possible to apply the scaling techniques to Radix2 as well as Radix4 division. However, reduction in execution time is more noticeable if the radix is increased to a value where using conventional method of quotient selection table incurs heavy penalty. This work shows the Radix16 division algorithm using scaling techniques. The basic requirements of the algorithm are a simple scaling technique to make quotient digit selection function independent of divisor and using small number of bits from partial remainder.

Since there are many conflicting parameters involved in the scaling, quotient digit selection functions are discrete some trial and error is necessary to achieve trade-offs.

Steps involved in the scaling algorithms for radix 16 are as follows

- a. Quotient digit set taken into consideration is from $\{-10, -9, \dots, 9, 10\}$ it is decomposed into two components as $qh = \{-8, -4, 0, 4, 8\}$ and $ql = \{-2, -1, 0, 1, 2\}$ respectively. Redundancy factor ρ for this algorithm is calculated as

$$\rho = \frac{a}{r-1} = \frac{10}{15} = \frac{2}{3} \quad (3.4.1)$$

Thus, the value of ρ is within bounds of $[1/2, 1]$. Number of bits of divisor required could be calculated from the 3.2.1 as $\delta = 6$ so for scaling all divisors are shifted to a range, which is above value of 1 from $[6/6, 7/6]$.

- b. The scaling should be obtained by two pass through adder as recursive algorithm requires 2 redundant adders to add ql and qh components. The scaling factor can consists of up to four multiples of unscaled divisor d of the form $2^i \cdot d$. More specifically we want to transform the unscaled divisor into scaled divisor such that

$$Z = M \cdot d \text{ and } 1 - \alpha \leq |Z| \leq 1 - \beta$$

$$M = 1 + b \cdot 2^{-6} \quad \text{with } b \text{ integer, } 0 \leq b \leq 64$$

- c. Since the range of divider is now small, selection function is independent of divisor.

$$q_{j+1} = \text{SEL}(\hat{y}) \text{ for } 1 - \alpha \leq Z \leq 1 - \beta$$

where, \hat{y} is the redundant partial remainder

Consequently,

$$m_k \geq m \cdot \alpha_k (1 - \alpha) L_k (1 - \beta) \quad (3.4.2)$$

$$m_k + 2^{-t} \leq m \cdot U_k \alpha_k (1 - \alpha) U_{k-1} (1 - \beta) \quad (3.4.3)$$

Table7 is used to find out the value of M by which operand needs to be multiplied to get scaled operand. Maximum value that M can have is 127/64 where as 64/64 is the least value M can have in order to scale the divisor.

Bits of divisor .1(d-2d-3d-4d-5d-6d-7) As divider is in range 0.5 to 1	Value of M	Lookup table	Multiplication by MUX
000000	127/64	1.98438	(64/64+32/64+16/64+8/64)D
000001	125/64		(64/64+32/64+16/64+8/64)D
000010	123/64		(64/64+32/64+16/64+8/64)D
000011	121/64		(64/64+32/64+16/64+8/64)D
000100	120/64		(64/64+32/64+16/64+8/64)D
000101	118/64		(64/64+32/64+16/64+8/64)D
000110	116/64		(64/64+32/64+16/64+4/64)D
000111	115/64		(64/64+32/64+16/64+2/64)D
001000	113/64		(64/64+32/64+16/64+1/64)D
001001	112/64		(64/64+32/64+16/64+0/64)D
001010	110/64		(64/64+32/64+16/64+0/64)D
001011	109/64		(64/64+32/64+8/64+4/64)D
001100	107/64		(64/64+32/64+8/64+4/64)D
001101	106/64		(64/64+32/64+8/64+2/64)D
001110	104/64		(64/64+32/64+8/64+0/64)D
001111	103/64		(64/64+32/64+8/64+0/64)D
010000	102/64		(64/64+32/64+4/64+2/64)D
010001	101/64		(64/64+32/64+4/64+1/64)D
010010	99/64		(64/64+32/64+2/64+1/64)D
010011	98/64		(64/64+32/64+2/64+0/64)D
010100	97/64		(64/64+32/64+0/64+1/64)D
010101	96/64		(64/64+32/64+0/64+0/64)D
010110	95/64		(64/64+32/64+0/64+0/64)D
010111	94/64		(64/64+32/64+0/64+0/64)D
011000	93/64		(64/64+16/64+8/64+4/64)D
011001	92/64		(64/64+16/64+8/64+4/64)D
011010	91/64		(64/64+16/64+8/64+4/64)D
011011	90/64		(64/64+16/64+8/64+2/64)D

011100	89/64		(64/64+16/64+8/64+1/64)D
011101	88/64		(64/64+16/64+8/64+0/64)D
011110	87/64		(64/64+16/64+8/64+0/64)D
011111	86/64		(64/64+16/64+4/64+2/64)D
100000	85/64		(64/64+16/64+4/64+1/64)D
100001	84/64		(64/64+16/64+4/64+0/64)D
100010	83/64		(64/64+16/64+2/64+1/64)D
100011	82/64		(64/64+16/64+2/64+0/64)D
100100	82/64		(64/64+16/64+2/64+0/64)D
100101	81/64		(64/64+16/64+1/64+0/64)D
100110	80/64		(64/64+16/64+0/64+0/64)D
100111	79/64		(64/64+16/64+0/64+0/64)D
101000	78/64		(64/64+32/64+4/64+2/64)D
101001	78/64		(64/64+8/64+4/64+2/64)D
101010	77/64		(64/64+8/64+4/64+1/64)D
101011	76/64		(64/64+8/64+4/64+0/64)D
101100	76/64		(64/64+8/64+4/64+0/64)D
101101	75/64		(64/64+8/64+2/64+1/64)D
101110	74/64		(64/64+8/64+2/64+0/64)D
101111	74/64		(64/64+8/64+2/64+0/64)D
110000	73/64		(64/64+8/64+1/64+0/64)D
110001	72/64		(64/64+8/64+0/64+0/64)D
110010	72/64		(64/64+4/64+0/64+0/64)D
110011	71/64		(64/64+4/64+2/64+1/64)D
110100	70/64		(64/64+4/64+2/64+0/64)D
110101	70/64		(64/64+4/64+2/64+0/64)D
110110	69/64		(64/64+4/64+1/64+0/64)D
110111	69/64		(64/64+4/64+1/64+0/64)D
111000	68/64		(64/64+4/64+0/64+0/64)D
111001	67/64		(64/64+2/64+1/64+0/64)D
111010	67/64		(64/64+2/64+1/64+0/64)D
111011	66/64		(64/64+2/64+0/64+0/64)D
111100	66/64		(64/64+2/64+0/64+0/64)D
111101	65/64		(64/64+1/64+0/64+0/64)D
111110	65/64		(64/64+1/64+0/64+0/64)D
111111	64/64		(64/64+0/64+0/64+0/64)D

Table 7 - Selection multiples for scaling

Steps involved in radix 16 SRT divide are as follows

- a. On reset operand x is loaded in. 5 bits are added to it at the start which serves as sign and four integer bit.
- b. Multiple of d and x are created which are shifted versions of operands

q_{total}	q_h	q_l	$U_{k-1}(8288/8192)$	$L_k(8288/8192)$	m_k
10	8	2	9.566	9.44	9.5
9	8	1	8.57	8.43	8.5
8	8	0	7.58	7.42	7.5
7	8	-1	6.59	6.40	6.5
6	4	2	5.60	5.39	5.5
5	4	1	4.61	4.38	4.5
4	4	0	3.62	3.37	3.5
3	4	-1	2.63	2.36	2.5
2	4	-2	1.64	1.34	1.5
1	0	1	0.66	0.33	0.5
0	0	0	-0.33	-0.66	-0.5
-1	0	-1	-1.34	-1.64	-1.5
-2	0	-2	-2.36	-2.63	-2.5
-3	-4	1	-3.37	-3.62	-3.5
-4	-4	0	-4.38	-4.61	-4.5
-5	-4	-1	-5.39	-5.60	-5.5
-6	-4	-2	-6.40	-6.59	-6.5
-7	-8	1	-7.41	-7.58	-7.5
-8	-8	0	-8.43	-8.57	-8.5
-9	-8	-1	-9.44	-9.56	-9.5

Table 8 - Quotient selection table for Radix16

- c. First 6 bits of divisor are used to probe selection table to find appropriate multiples of d which are used to generate scaled divisor $Z=M.d$
- d. Then multiples of x are chosen to generate scaled dividend $M.x$ which serves as the initial partial remainder $w_{[0]}$, at the same time Z is assimilated into standard form using carry propagate adder.
- e. According to the quotient selection table (Table8) shown above maximum bounds on partial remainder to get correct quotients are $9.5 < \hat{y} < -9.5$ so sign bit, 4 integer bits and 1 fractional bit are sent to qst to chose correct values of q_l and q_h
- f. Selected q_l and q_h chooses appropriate value of scaled divisor $Z = M.d$ from pre-computed multiple of Z { -8, -4, 0, 4, 8} and {-2, -1, 0, 1, 2}
- g. When negative multiple of scaled divisor is selected multiplexer chooses inverted version of d & additional 1 required to produce its 2's compliment is added in *ulp* position of carry save adder.

- h. Redundant quotient bit is sent to OTF (On The Fly) converter.
- i. When algorithm requires input operands to be of size greater than double precision, delay due to carry propagate adder, which assimilates scaled, divisor becomes the critical path, CPA can be replaced by carry save adder which results in increase in number of inputs to the carry save path in recursive path.

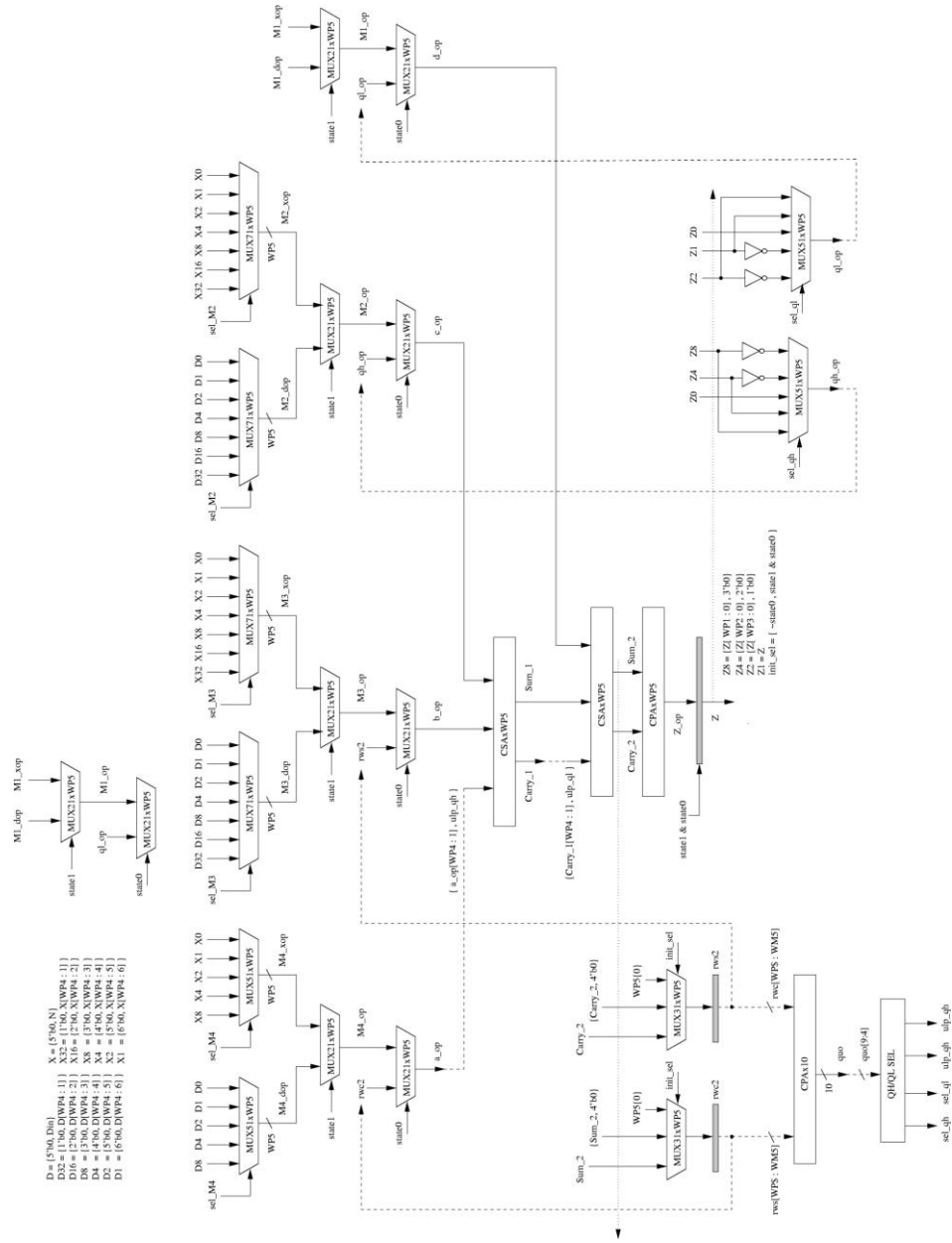
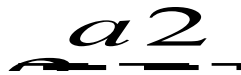


Figure 8 - Radix16 hardware block diagram

3.5 Radix 512 division

Previous algorithm showed that operand scaling is useful technique to reduce the complexity of quotient selection table by making selection function independent of bits of the divisor by scaling the operands. Operand scaling techniques multiplies both dividend and divisor by prescaling factor. When we go to radix, which are higher than radix 16 (for e.g.: radix 512) maintaining condition of minimum number of quotient digits in the set to satisfy bounds on redundancy factor requires quotient digit set to be $\{-255, 256\}$.



And if we use similar technique to radix 16 we require 8 different multiples of d , x and quotient needs to be split into 8 components as compared 2 components in radix 16. Moreover 8 pre-computed multiples of d and x not only increases the redundant hardware, it also makes Carry Save Adder tree in the recursive path to grow big. As a result of which using same technique in radix 512 nullifies the advantage gained by computing 9 bits of quotient every iteration.

There are several methods to accomplish the pre-scaling function. Some designs actually incorporate linear approximations to the function, so that for all values inside a given interval same scaling factor is used [7]. Although this method is useful, it requires extra iterations to acquire the approximation. Instead of using linear approximations this work uses table lookup that gets the best approximation to the function for given interval. This approximation called the Symmetric Bipartite Table Method (SBTM). Number of bits of divisor for which we need approximation are 14. There are various ways in which number for which approximation is required can be split up. In this setup 16 bits of divisor is split into 3 components as $\{5,4,5\}$. First two components form number which acts as in input to table a_0 and first 5 and last 5 bits acts as an input to table 2 as shown in fig below.

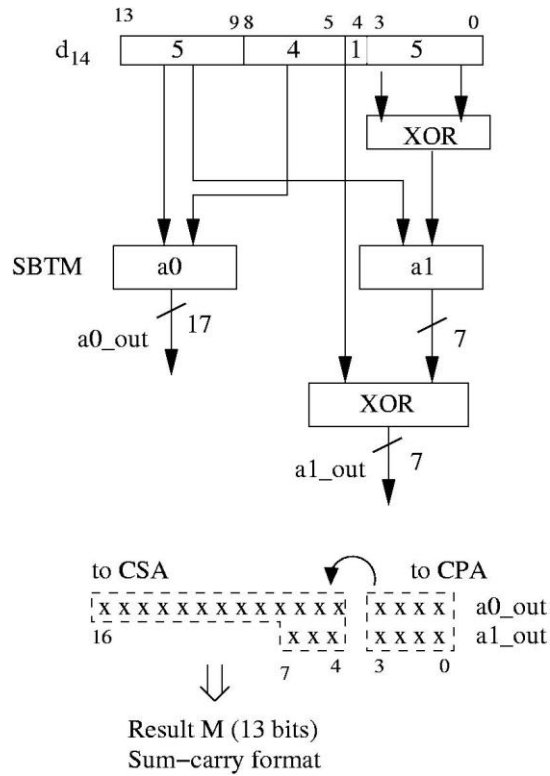


Figure 9 - SBTM implementation

Generally values generated by SBTM lookup table a_0 and a_1 needs to be added together however initial pre-scaling value can be stored in carry-save mode, so there is no need to incorporate the carry propagate adder saving clock cycle time.

According to recursive algorithm for SRT 512 division

$$w_{[i+1]} = 512. w_{[i]} + q_{[i+1]} * d \quad (3.5.1)$$

With quotient digit selection:

$$q_{i+1} = \lceil \hat{y} + 1/2 \rceil \quad (3.5.2)$$

Where, $w[i]$ is the residual after iteration i

$$w_{[0]} = M.x$$

q_{i+1} is the quotient digit generated in the iteration $q_i = \{-511, \dots, 0, \dots, 511\}$

$\hat{y} = \{rw[j]\}_2$ is partial remainder restricted till 2 fractional bits

At the end, q must be rounded according to the sign of the residual of the last iteration. If last residual is positive we have to add 1 in the least significant position after rounding. We do not add anything if it is negative.

To execute the recurrence two multiplications and one addition is required. The partial remainder $w[i]$ and the quotient digit q_{i+1} both are in carry save presentation to avoid carry propagation adder in critical path. Multiplying a number by $r = 512 = 2^9$ is equivalent to the shifting of its binary representation by 9 positions to the left. Shifting the scaled divisor by quotient value can't perform the other multiplication as its not strictly in multiples of 2; so recoding one of the operands is used to perform other multiplication. Recoding the multiplier into radix 4 representation reduce the number of partial products and make the operation faster. The recoded operand is in Signed Digit representation and each digit can assume values $\{-2, -1, 0, 1, 2\}$.

In carry save representation of M both MSB's are going to be 0 as we know the value of M is positive, but q_s and q_c can be positive or negative, since the sign is not known, the only solution is to extend partial remainder y by one bit the carry save representation of q_i . This also requires appending 3 integer bits at the start.

3.5.1 Recoding

In recursive algorithm concept of recoding is used. This is a standard radix 4 multiplication. Every partial product is 2 positions left shifted with the respect to previous one, because of the radix 4 representations of the digits. Sign extension is required to get correct result.

```
SSSSSSSXXXXXXXXXXXXXXXXXXXXX
SSSSSXXXXXXXXXXXXXXXXXXXXX00
SSSXXXXXXXXXXXXXXXXXXXXX0000
```

When a negative digit is encountered, we bit compliment the partial product (before the shift) and we put a 1 in the next product in correspondence of least significant bit of the actual product. If digit is -1 we have

```
SSSSSSSXXXXXXXXXXXXXXXXXXXXX
SSSSSCCCCCCCCCCCCCCCCC00
SSSXXXXXXXXXXXXXXXXXXXXX0100
```

And if the digit is -2 we have

```
SSSSSSSXXXXXXXXXXXXXXXXXXXXX
SSSSSCCCCCCCCCCCCCCCCC00
SSSXXXXXXXXXXXXXXXXXXXXX1000
```

Putting 1 for negative multiples of operand doesn't change the complexity of the circuits. As shown in the following diagram these bits in the first and second shift extension are simple through M2 and M1.

But this method cannot be used if the last partial product is negative; this problem will never occur during scaling as value of M is positive, but during calculation of quotient as well as during high radix multiplier last partial product can be positive or negative. In such case we bit compliment shifted partial product and add 1 in the least significant position

```

SSSSSSSSXXXXXXXXXXXXXXXXXXXXX
SSSSSXXXXXXXXXXXXXXXXXXXXX00
SSSXXXXXXXXXXXXXXXXXXXXX0000
SXXXXXXXXXXXXXXXXXXXXX111111
00000000000000000000000000001

```

As shown in the figure10 input to the each recoder pair is v_i is in carry save format, so maximum value of v_i can be 6, when both sum and carry bit of the input are 11. Input to the each recoder is v_i is in the range of $\{0,6\}$. Recoding of the carry save operands is shown below.

v_i	$h_i=0$			$h_i=1$		
	h_{i+1}	g_{i+1}	t_i	h_{i+1}	g_{i+1}	t_i
0	0	0	0	0	0	0
1	0	0	1	0	1	-3
2	1	0	-2	1	0	-2
3	1	0	-1	1	0	-1
4	1	0	0	1	0	0
5	1	0	1	1	1	-3
6	1	1	-2	1	1	-2

Table 9 - Quotient recoding in Carry-Save format

Final recoded output is $R_i = t_i + h_i + g_i$

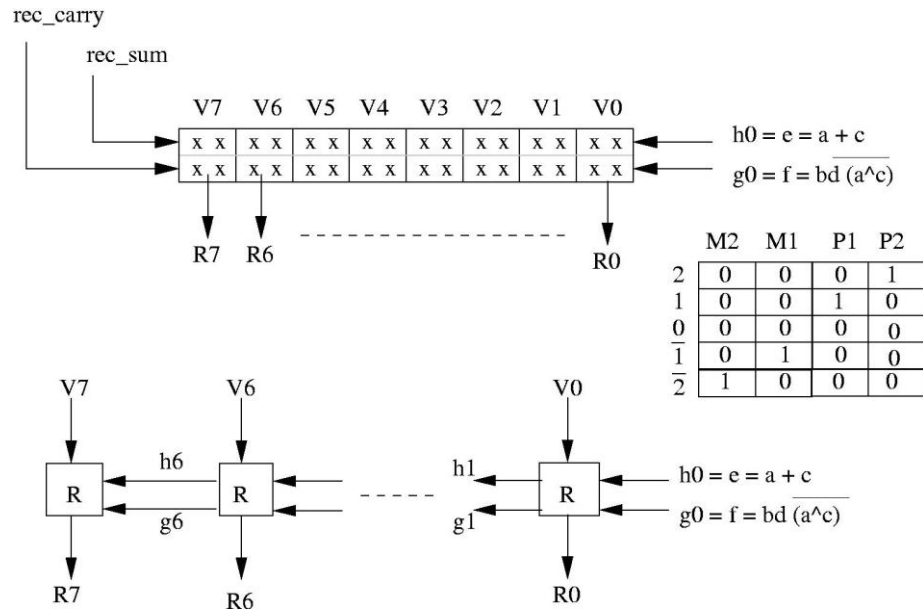


Figure 10 - Block diagram for recoder

3.5.2 Adder

Adder employed in Radix512 design can be split into components, first one calculates partial product depending on the output of recoder, and second component adds them. During calculations of scaled divisor and dividend 8 partial products are generated as pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7. During recursive algorithms as only 6 partial products are generated by quotient q_{i+1} . Upper 2 partial products pp6, pp7 are replaced by shifted partial residual $512.ws[i]$ and $512.wc[i]$.

Cycle	1	2	3	4	5
Operation	Prescaling	Prescaling	Recurrence	Recurrence	Recurrence
SBTM Register	M	M	M	M	M
W (Carry/Sum)	M.d =Z	M.x=W[0]	W[1]	W[2]	W[3]
Z	-	M.d	M.d	M.d	M.d

Table 10 - Division cycle timing chart

Steps involved in divide512 algorithm are as follows

- a. In cycle 1 that is termed as Prescaling, SBTM Register sends pre-scaling factor M in carry save format to one of the multiplexors, which in turn sends those value to the recoder.

- Operand selector multiplexor chooses operand d , recoder outputs chooses correct multiple of operand (d in this cycle), which are added in CSA tree to form $M.d$ in carry save format.
- b. In cycle 2 that is also termed as Prescaling, SBTM Register sends prescaling factor M in carry save format to one of the multiplexers, which in turn sends those values to the recoder. Operand selector multiplexor chooses operand x , recoder outputs chooses correct multiple of operand (x in this cycle) which are added in CSA tree to form $M.x$ in carry save format which is the initial partial remainder $w[0]$, and in same cycle $M.d$ is assimilated using carry propagate adder.
- c. From cycle 3 onwards which is termed as Recurrence operation quotient q_{i+1} is sent to the recoder and operand selected is Z scaled divisor

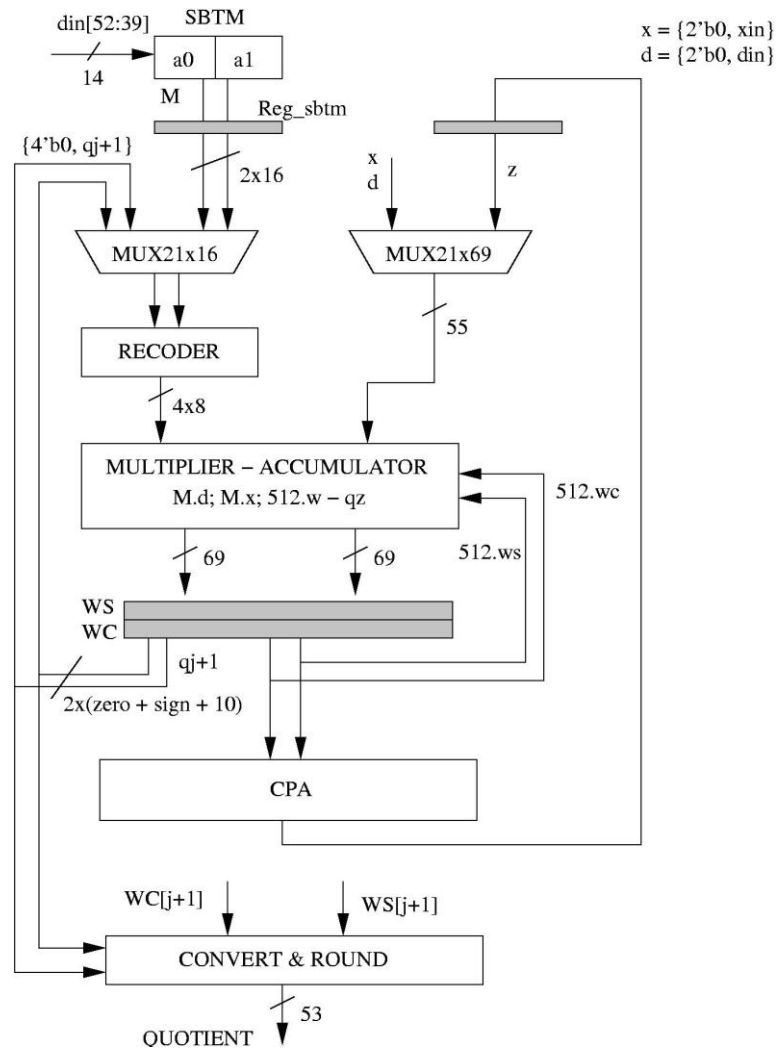


Figure 11 - Radix512 hardware block diagram

Quotient produced by recurrence algorithm is in Signed Digit representation, which is converted into standard representation using on the fly converter block.

- d. While implementing radix 512 division for quad precision or some other industry floating point formats with large range if carry propagate adder involved in cycle 2 which assimilates scaled divisor $M.d$ becomes critical path of the circuit, scaled operand can be stored in carry save format. (Modification are shown in figure12)

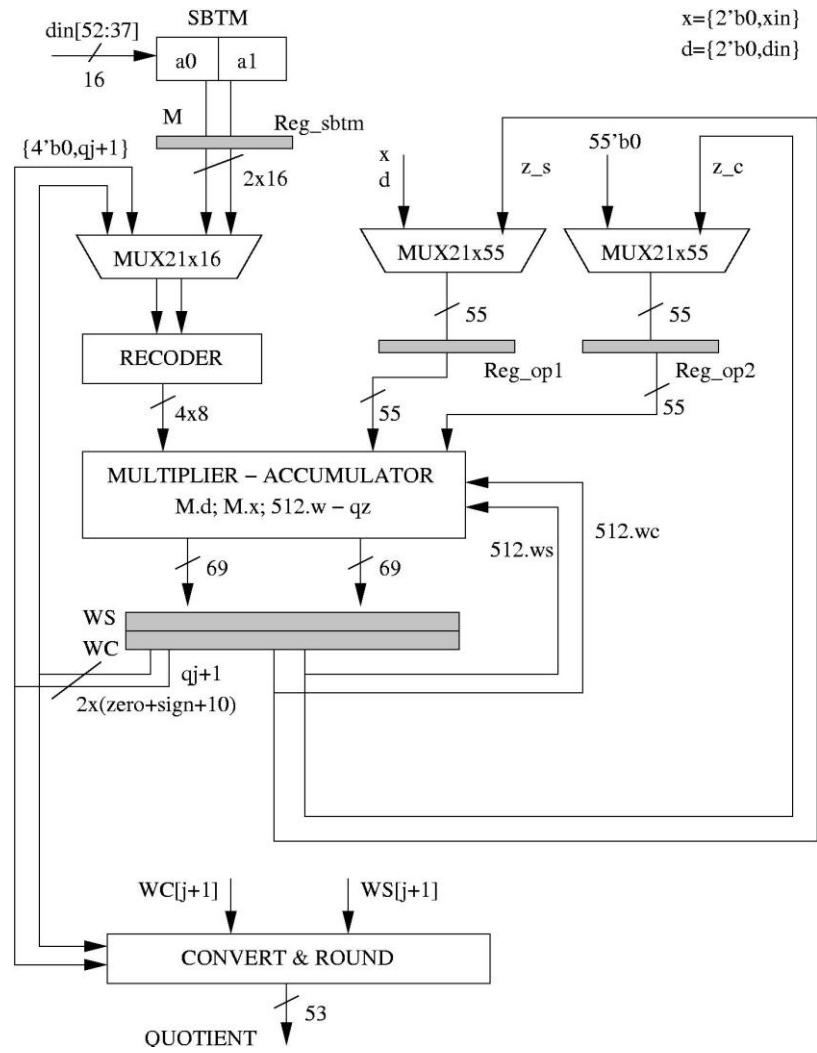


Figure 12 - Radix512 hardware block diagram using Carry-Save format

- e. In such case where prescaled divisor is stored in carry save format, number of partial products getting added in Carry Save Adder tree gets doubled as each partial product will be have its sum and carry version.

3.6 Recursive divide/multiplier unit

This section describes modifications in the radix 512-division algorithm to accommodate high radix multiplier. . This unit reduces its overall area and power footprint by utilizing a low area sequential architecture, but still maintains high amounts of computation power by employing high radix and redundant digit encoding for each digit.

Multipliers have been key and critical components for most application specific and general-purpose computer architectures. However, these architectures have been transitioning towards multiple cores that process large amount of data through parallel approaches to computation. Unfortunately, traditional arithmetic functional units that worked well for dingle core architectures have the side effect of incurring large amounts of area and power. Consequently, multi-core architecture needs new ways of thinking about increased throughput to handle large amounts of data. This section presents modification in the recursive high radix divide unit that is modified to handle both multiplication and division targeted at multi-core architectures.

Multiplication involves the use of addition in some way to produce product $p = x \cdot y$ from a multiplicand x and multiplier y . Multipliers and even division have been workhorse of scientific computing [8]. High-speed multipliers are typically classified into two categories. The first known as parallel multiplication, involves the use of hardware to multiply an m bit number by an n bit number to completely produce an $n+m$ product. On the other hand, serial or sequential multipliers compute the product sequentially usually utilizing storage elements so that hardware of the multiplier is reused during an iterative process.

Although multipliers are important for processing scientific data; they tend to consume a significant amount of area and delay within computer architectures. Today's architectures require significant amount of area to integrate multiple cores on a chip to satisfy Moore's Law.

For example, the Niagara 2 architecture has significant resources allocated for supporting multiple threading [9]. Subsequently Sun Microsystems utilized architecture to reduce area in order to allow the integrating of floating point and graphics unit. Decisions such as these will continue to dominate as the number of threads and cores increase and multi-core architecture becomes more prevalent. Consequently, designers must design new computational blocks that still maintain the same level of algorithmic complexity, but also have smaller area and power footprints.

Multipliers are an integral part in scientific computing, however they consume a large amount of resources. To reduce the size of the computation, it is common to have architectures that are designed to serially instead of parallel. Serial multipliers are typically computed by repetitively adding partial products to form the final product as follows:

$$p[j+1] = (p[j] + x[j].y).r^{-1} \quad (3.6.1)$$

where p is the product,

y is the multiplicand

x is the multiplier

r is the radix.

Although serial multiplication units are fairly simple to build in digital hardware, they tend to be significantly slower than the larger parallel multiplier architectures. More complex architectures, such as Goldschmidt and Newton Raphson algorithms, allow faster and more efficient multiplication and division algorithms to occupy the same functional unit. Although these quadratically converging algorithms are efficient, they have the disadvantage of using a significant amount of area, mainly for the parallel multiplier. In addition, traditional parallel multipliers also consumes large amount of power with their spurious switching activity [9].

Although serial multiplication units are fairly simple to build in digital hardware, they tend to be significantly slower than larger parallel multiplier architectures. More complex architectures, such as Goldschmidt and Newton Raphson algorithms, allow faster and more efficient multiplication and division algorithms to occupy the same functional unit. Although these quadratically converging algorithms are efficient, they have the disadvantage of using a significant amount of area, mainly for the parallel multiplier. In addition, traditional parallel multipliers also consumes large amount of power with their spurious switching activity [9].

Long standing argument that diminishing feature sizes justify larger area for more functionality is starting to lack any substance in today's multicore environment. This is because many multi-core architecture needs additional blocks for other on chip and off chip communications. Therefore, multi-core architectures require substantial area for the use of local and off-chip routers and reducing the size of key and critical functional units such as multipliers and dividers, is paramount to promoting multi-core paradigms. On the other hand, the reduction in size of these arithmetic circuits should be tempered with not decreasing the efficiency of the unit too dramatically.

One potential improvement for serial architectures is to compensate for the low algorithmic complexity of serial recursive multipliers by using higher radix and Signed Digit encoding. The advantage to this approach is that a designer can obtain lower power functional units while still maintaining high throughput. Recursive division algorithms have long had success in promoting high amounts of computing power by using a higher radix as shown by the following equation

$$w_{[i+1]} = r. w_{[i]} - d. q_{i+1} \quad (3.6.2)$$

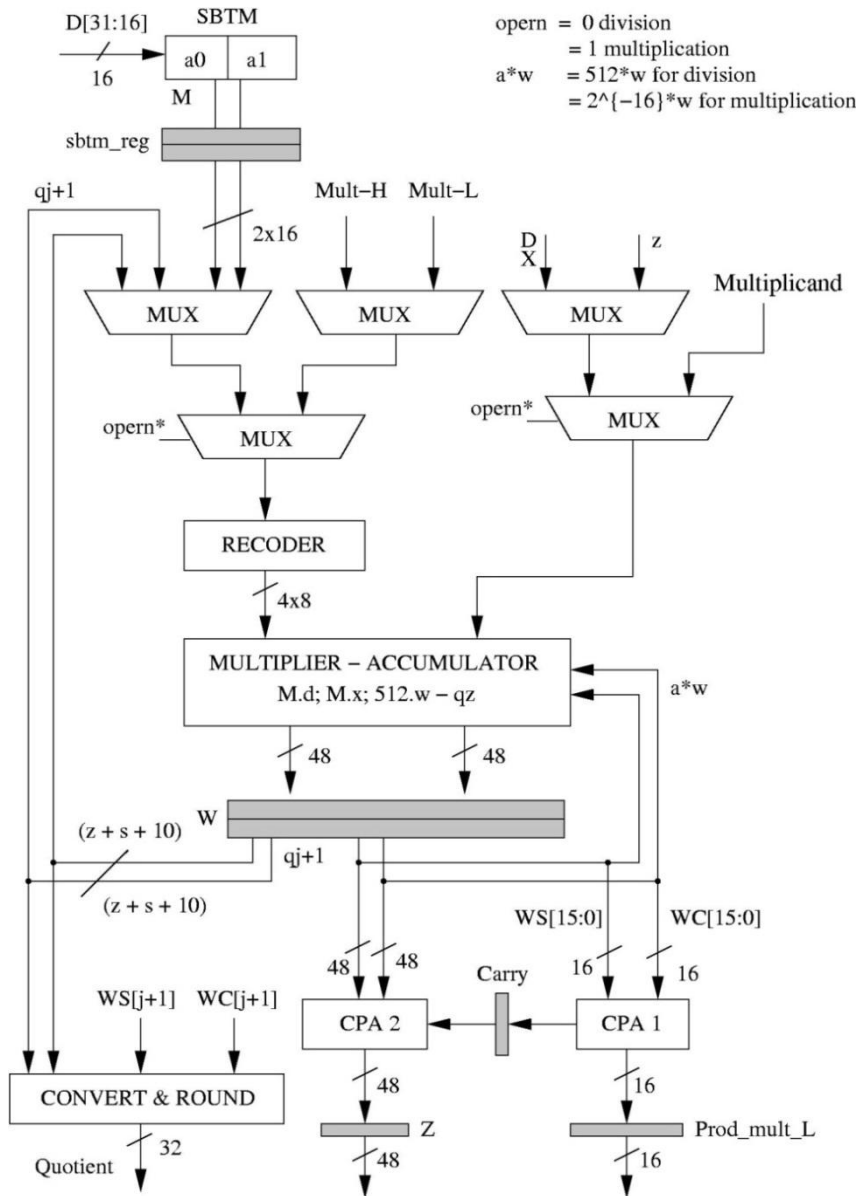


Figure 13 - Radix512 hardware block diagram for integrated Mult-Div Unit

The division architecture can easily be expanded into one that computes both division and multiplication by adding an extra data flow such that the prescaling operation is concatenated for a selection of multiplier. This is, since only a limited number of bits are utilized with lookup table for the scaled divisor, it can easily be modified to handle the lower elements of product, given the dividend is changed to the multiplicand [10].

For the $r=512$ unit paper, 16 bit is assumed for the approximation within the SBTM unit, half of the product or 32 bits is produced after 1 cycle, That is the following operation is computed:

$$M_{\text{plier}} \text{ 16 bits} \cdot M_{\text{cand}} \text{ 32 bits} = \text{Partial Product 48 bits}$$

A subsequent iteration can then incorporate into the datapath so that upper 16 bits of the product or the complete 64 bits of the product is produced. Since the multiplication algorithm is based closely on the radix 512-division algorithm, it is conceivable to easily modify the architecture to handle both multiplication as well as division within one unit. The major difference is for 32 bit multiplication requires 3 cycles, whereas, division requires 5 cycles. Moreover since multiplication has to shift multiplier by 16 bits, whereas, division shifts each partial remainder by 9 bits. Figure13 displays the combined architecture for 32-bit division and multiplication.

Cycle	1	2	3
Operation	Iteration	Iteration	Iteration
SBTM Register	M	M	M
W (Carry/Sum)	Mcand.MplierL	Mcand.MplierH	-
Z	-	-	Mcand.MplierH
Prod_mult_L		Mcand.MplierL	Mcand.MplierL
Carry	-	C	C

Table 11 - Multiplication cycle timing chart

CHAPTER IV

RESULTS

In order to quantify the architectures within the paper, the implementations were analyzed using the IBM cmos10lpe 65nm CMOS technology[11]. The standard cell library from Virage is utilized for the synthesis and place and route of the all architecture discuss in this work. All designs are coded in RTL Verilog and synthesized and place and routed using Synopsys tool chain. The power dissipation is calculated by running 1,000 different test vectors for each design.

All radix dividers are compared against each other to get accurate estimate of power as well as area and speedup comparison. To compare the recursive divide/multiplication structure versus traditional multipliers, a 32-bit carry-save array and Dadda multiplier are designed and analyzed. Parallel multipliers are chosen for this comparison, because they are typically employed within multiplicative-divide architectures and consequently can give a good comparison to the proposed architecture within the paper.

4.1 Design Flow:

Flowchart in figure14 shows steps involved in the design and implementation of architecture mentioned in this work [12]

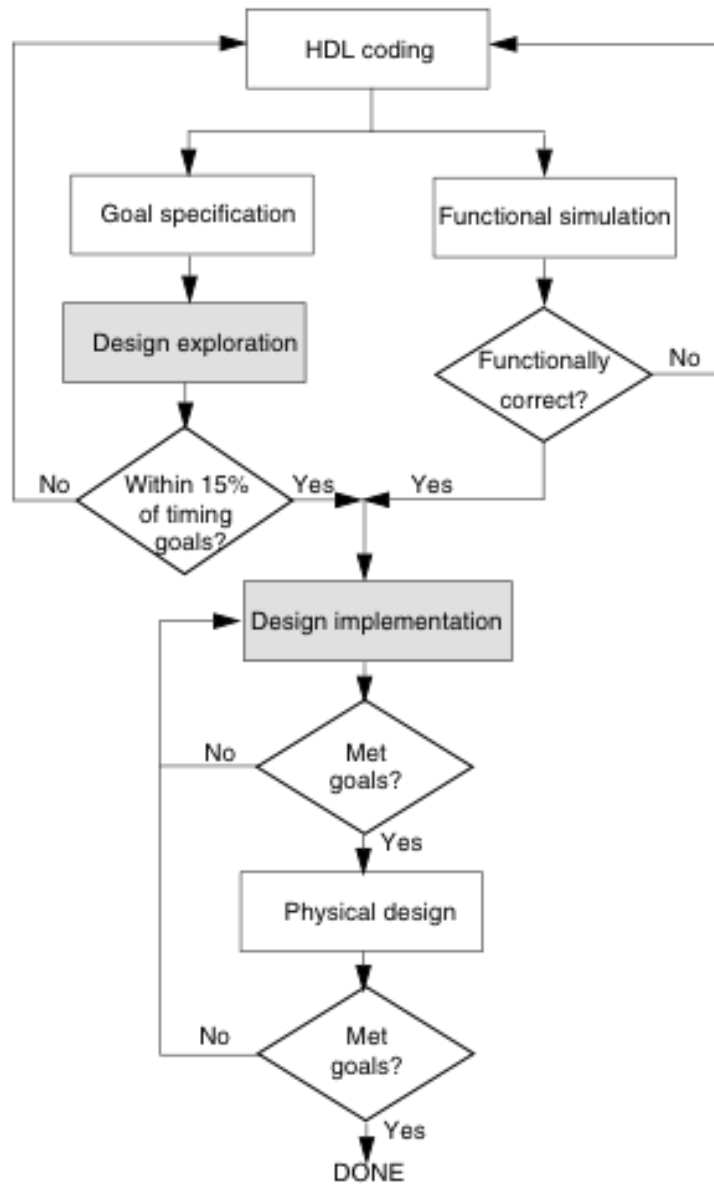


Figure 14 - Design Flow

4.1.1 Design Specifications

There are lots of articles available on SRT division algorithm, which does nice job of explaining the concept and theory behind it. In the initial phase of the design algorithms are implemented in C language it get better understanding of concept. This is followed by paper calculations for small operands range of 8 bits and results obtained which are compared against C program to validate the result. A sample paper calculation for radix-512 division is shown below

```
X = 0.84163E2
D = 0.F7755D7
Z   = M.D = 0000.FFFDF5A
w[0] = M.X = 0000.88A4527
```

1st Iteration:	512.WS[0]	0111.48A4E00	q = + 273
	512.WC[0]	0000.0000000	
	-273. Z	FFFF.022C706	
	WS[1]	FFFC.4AC1506	
	WC[1]	0004.0010000	
2nd Iteration:	512.WS[1]	F895.83E0C00	q = + 150
	512.WC[1]	0800.2000000	
	-150. Z	FF6B.01300DE	
	WS[2]	1FFE.12CE8DE	
	WC[2]	E002.9242400	
3rd Iteration:	512.WS[2]	FC25.9D1BC00	q = + 37
	512.WC[2]	0524.8480000	
	-37. Z	FEB7.029F55A	
	WS[3]	0FFE.D23095A	
	WC[3]	F002.520A800	

```
Quotient = 0.88A5695
```

4.1.2 HDL Coding

In next stage of implementation, designs are described at register transfer level using Verilog hardware description language. Hardware implementation is done at structural level to avoid potential problems in the synthesis of the design. All these design use MUX, registers, and adders as their basic building blocks. To reduce design efforts Perl scripts are developed which will generate building block of desired size.

ModelSim is used for the simulation of Verilog netlist. It's a very powerful tool that allows user to graphically analyze the design. Complex modules such as quotient selection table, recoder are tested independently to verify the their functionality against the truth table. Once individual unit testing is done, all the units are compiled to test the entire module. Control logic is built using state machine, which gets triggered from external world by start signal so that these design can be easily incorporated into bigger designs such as processor systems [13].

Testbench used for the testing for division algorithm instantiated two 1 dimensional array which were filled with user defined values to test the functionality of the design, all corner cases are taken into consideration in this mode of testing.

For exhaustive testing system task functions in-built into verilog like \$random () are used to generated vector for dividend and divisor. Output generated are written in .out file using \$fopen, \$fdisplay task functions.

ModelSim is also used at pre-layout and post layout stage in order to check the functionality the design after synthesis and place route. It is used in conjunction with sdf annotation to make sure the design is meeting timing constraints set in design planning stage of the design. This part will be covered in detail at post-layout section.

ModelSim allows user to create .do file in which you can specify the constraints on the design like verilog files to compiled, adding waveforms of the design which needs to be analyzed as well as window configuration such as WaveRestoreZoom, name column width, signal name width, value column width etc.

Commands to run the simulation: - vsim -do divide.do

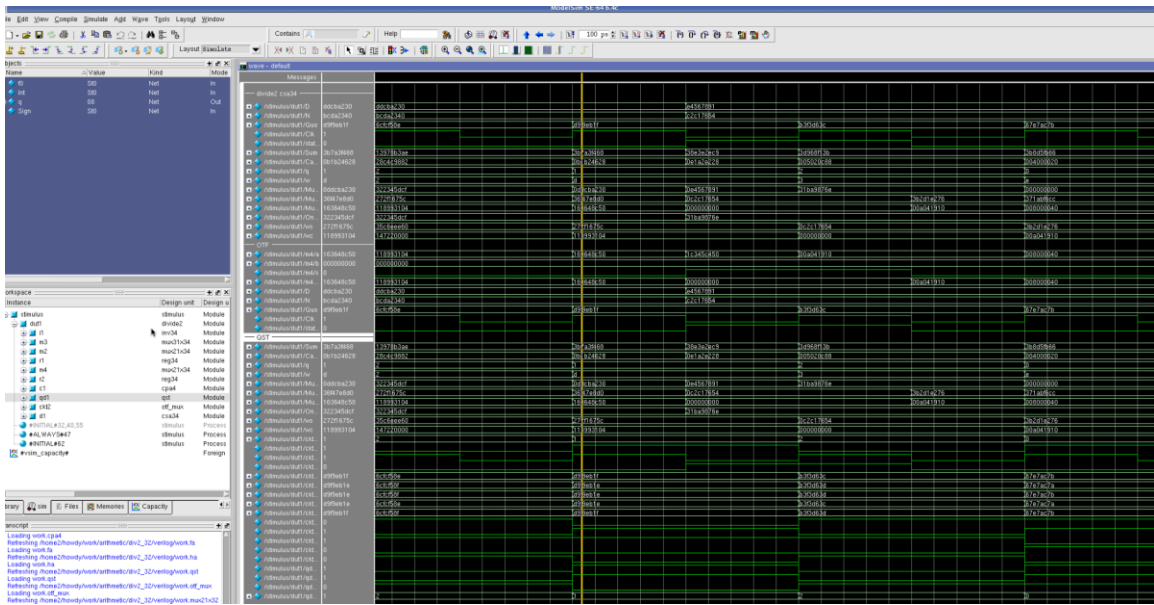


Figure 15 - ModelSim simulation snapshot

4.1.3 Logic synthesis

Synthesis is the process of transforming hardware description at RTL level into structured netlist using cells from standard cell library. Behavior of the design remains the same after synthesis but now design is described as a collection of cells from standard cell library. Design compiler is the tool from Synopsys used for this process, which comes in two flavors as `dc_shell`, which is command line interface, which is preferred by most of the EDA designers, whereas other flavor is `design_vision`, which gives graphical representation of the design to inspect critical path of the design, generating schematic etc.

For the successful synthesis of the design, design compiler required RTL description of the design as an input as well as `.sdc` file, which is (Synopsys Design Constraint) file. Synopsys design constraint file specifies information related to clock like Clock period, skew, latency, transition. Design compiler also allows you to put design constants such as `don't use` attribute to prevent compiler from using some of the standard cells, delay associated with input and output ports, driving cell, output load etc. Design compiler generates following file formats and report:

vh	Synthesized netlist of the design
sdc	A constraint file that can be used to pass the constraints that we set in the synthesis script to place and route tools
sdf	It's a standard delay format file that can be used for back-annotation with extracted timings from synthesized design
ddc	A Synopsys formatted binary database that can be used in design vision for further processing like viewing critical path, analyzing timing reports etc.
rep	These are generated reports from the synthesis tool which provides information related to timing, power as well as estimated area of the design

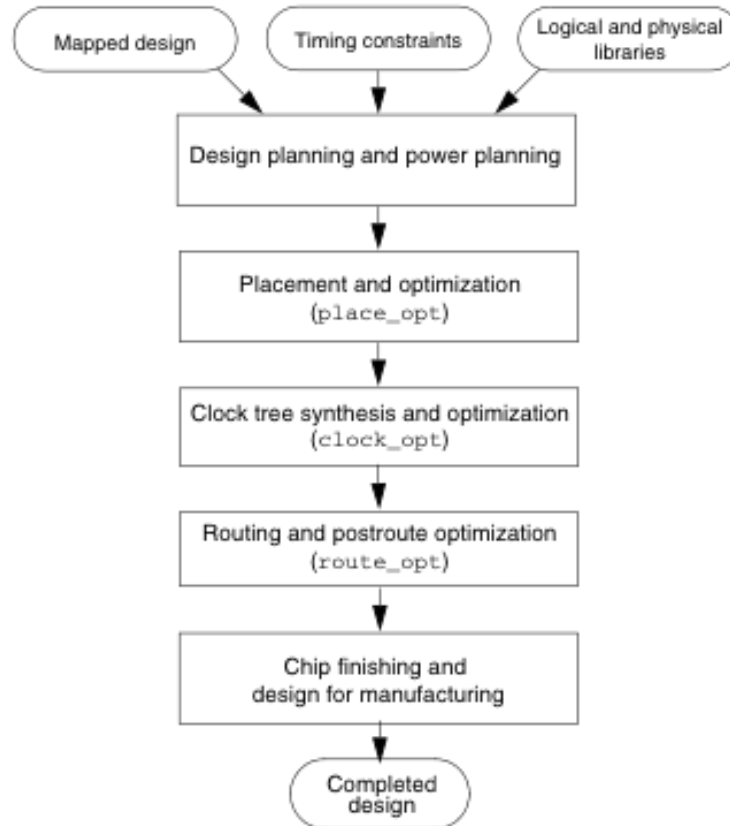


Figure 16 - Place and Route Design Flow

Synopsys design compiler used tcl environment to run the scripts. Command used for running design compiler is

```
dc_shell-xg-t -f compile_dc_ultra.tcl
```

Formality is logic equivalence tool, which is used to compare the synthesized netlist and RTL netlist for functionality matching.

4.1.4 Place and Route

After this stage Synopsys IC compiler tool is used to perform place and route of the design. This tool requires synthesized netlist, target libraries, Synopsys design Constraint file as an inputs. Figure 16 flow-chart shows design flow of IC compiler tool.

After the place and route is done. Dynamic simulation can be run on the post-layout to confirm functionality as well as timing constraints of the design. This is done by back annotating sdf file generated by place and route tool. While performing post-layout simulation slight modifications are required to be done in the testbench as shown below:

```
initial    begin
            $dumpfile(`divide2.vcd");
            $dumpvars(0, stimulus.dut1);
            $sdf_annotate(`divide2.sdf", dut1, , , "MAXIMUM", , );
        end
```

4.1.5 Post-Layout Analysis:

Primetime basic is a sign-off quality static timing analysis tool from Synopsys. Static timing analysis is without the doubt the most important step in the design flow. It determines whether the design works at required speed. PrimeTime analyzes the timing delays in the design and flags violation that must be corrected. Four types of analysis is performed on the design, as follows:

- a. From primary inputs to all flops in the design
- b. From flop to flop
- c. From flop to primary output of the design
- d. From primary inputs to primary outputs of the design

One of the critical parts in the timing analysis of the design is setup and hold time violations. PrimeTime allows you to generate timing reports for flop-to-flop path group using – delay_type option with max and min values to validate hold and setup time violations. If slow and fast timing libraries of the design are available then those libraries can also be preloaded in the PrimeTime tool to check corner cases of setup and hold time violations. Following commands are used to perform setup and hold time violations

```
report_timing -from [all_register-clock_pins] \
              -to [all_registers -data_pins] \
              -delay_type min
report_timing -from [all_register-clock_pins] \
              -to [all_registers -data_pins] \
              -delay_type max
```

It is also used for power estimation of the design using switching activity file .vcd dump file generated during dynamic simulation.

4.2 Comparison

In this section we perform comparisons of different divide architectures with their delay, power and area numbers. As mentioned before the evaluations in the [4] are given in terms of delay and area of 2 inputs NAND gate, whereas in [7] are given in terms of full-adder units. As mentioned before aim of this work is get an accurate evaluation of these metrics by actual hardware implementation. All the work is implemented is IBM65nm technology but these evaluations can be extended to any library.

	Timing (ns)	Iterations	#Cells	Area (um2)	Power	
					Static (uw)	Dynamic (mw)
SP	1.01	25	348	1884	0.629	1.277
DP	1.06	54	563	3220	1.087	2.066

Table 12 - Radix2 Divide Results

	Timing (ns)	Iterations	#Cells	Area (um2)	Total Power (mw)	
					Static (uw)	Dynamic (mw)
SP	1.54	13	442	2440	0.801	1.357
DP	1.65	27	681	3955	1.652	2.231

Table 13 - Radix4 Divide Results

	Timing (ns)	Iterations	#Cells	Area (um2)	Power	
					Static (uw)	Dynamic (mw)
SP	1.87	8	1545	5802	1.841	1.462
DP	1.93	15	2210	8450	3.521	2.687

Table 14 - Radix16 Divide Results

	Timing (ns)	Iterations	#Cells	Area (um2)	Power	
					Static (uw)	Dynamic (mw)
SP	1.96	5	705	9506	3.576	2.057
DP	1.98	8	1008	16326	6.172	3.515

Table 15 - Radix512 Divide Results

	Timing (ns)	Iterations	#Cells	Area (um2)	Power	
					Static (uw)	Dynamic (mw)
SP	1.96	5/3	836	11292	4.090	2.191
DP	1.98	8/5	1333	20143	7.403	4.733

Table 16 - Radix512 Mult-Div Results

Next page shows graph of the delay and area of the design. Delay is calculated by multiplying number of iterations by clock cycle time.

The graph shows clearly that as complexity of the algorithm increases area consumed by design also increase but at the same delay required for the computation decreases. So using higher radix divider when speed is the priority is feasible choice

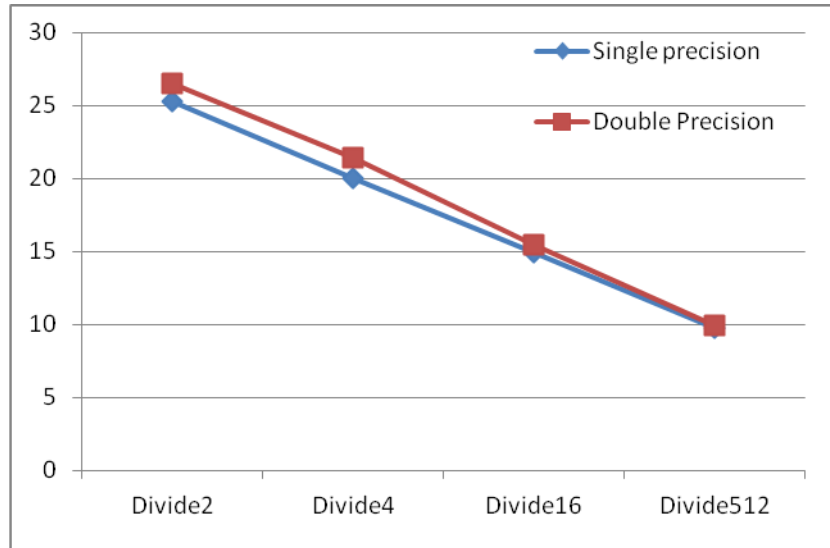


Figure 17 - Delay Comparisons

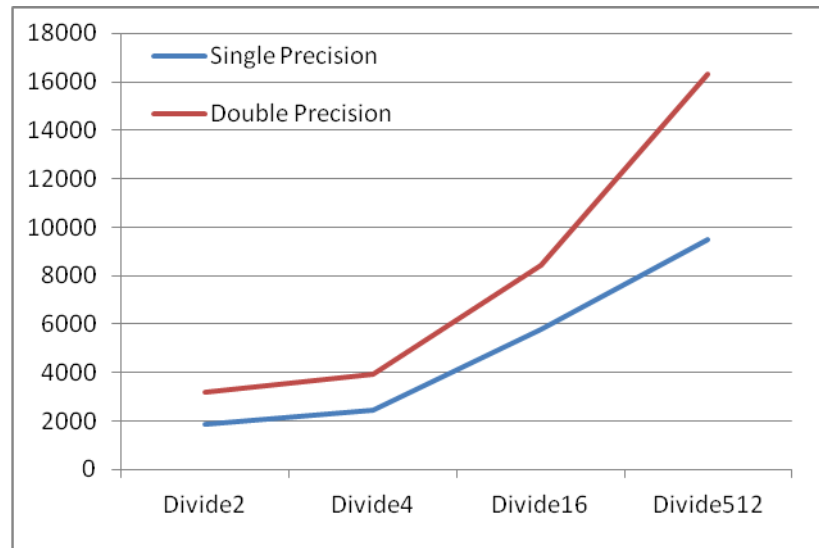


Figure 18 - Area Comparisons

Advantages gained by high radix divider are more visible in recursive multiply/divide structure. In recursive multiply/divide design increase in the area over conventional divide 512 design for single precision and double precision is shown below

Format	Divide512	Mult-Div Unit	% Area Increase
Single Precision	9506	11292	18
Double Precision	16326	20143	23

Table 17 - Relative area increase

Multiplier Arch	Timing (ns)	Iterations	#Cells	Area (um2)	Power	
					Static (uw)	Dynamic (mw)
Dadda	8.75	1	3250	12656	1.943	2.468
Array	9.16	1	2105	11424	1.513	2.513
Proposed	1.96	3	836	11292	4.090	2.191

Table 18 - Single Precision Comparisons for various multiplier architectures

Multiplier Arch	Timing (ns)	Iterations	#Cells	Area (um2)	Power	
					Static (uw)	Dynamic (mw)
Dadda	17.40	1	12325	50323	9.200	6.375
Array	18.98	1	8562	46876	4.079	7.553
Proposed	1.98	5	1333	20143	7.403	4.733

Table 18 & 19 shows that proposed combinational mult/div unit achieves smaller area, delay and power footprints as compared to traditional parallel multiplier architectures. In addition to this it gives more functional power by allowing same hardware design to be used for both division as well as multiplication.

CHAPTER V

CONCLUSION

This work shows the comparison between different architectures of SRT division algorithm in terms of performance, area and power. It shows that using high radix divider is a practical option when performance has important priority.

This work also shows modifications in high radix divider to come up with sequential architecture to handle both divide and multiplication. The proposed design utilizes multiplexers to select either the dividend/divisor or multiplicand/multiplier and employs a specialized shifter to allow the product to be shifted into the correct location. Results indicated the proposed design occupies less area and dissipates significantly less power than traditional multiplier. Moreover the proposed design integrates novel encoding methods to allow two's complement operands to be employed for wither division or multiplication. Units similar to the proposed architecture can impact multicore architectures by allowing more area for communication and graphic blocks.

This work also describes design flow that can be very efficient in the design of high-speed arithmetic modules, especially to get accurate evaluations of different schemes and algorithms.

In future, we will be analyzing the impact of new recoding schemes such as booth 3 and booth 4 on the existing architecture. It will be interesting to see tradeoff between complexity of recoder and number of partial product terms in Carry Save Adder tree. We will be integrating this multiplier/divide design in full function floating point core. Floating point unit core will be used in upcoming multi-core architecture project.

REFERENCES

- [1] S.F.Oberman, Michael Flynn “Division Algorithms and Implementations” in *IEEE Transactions on Computers* vol. 46 August 1997
- [2] S.F.Oberman, Michael Flynn “Design issues in Division and Other Floating Point Operations” in *IEEE Transactions on Computers* vol. 46 Feb 1997
- [3] J.E.Stine, *Digital Computer Arithmetic Datapath Design using Verilog HDL*. Springer 2004
- [4] M.D.Ercegovac, T.Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations* Kluwer 1994
- [5] A.Nannarelli, “Implementation of a Radix-512 divider.” Master’s thesis, University of California, Irvine, 1995.
- [6] M.D.Ercegovac, T.Lang “Simple Radix-4 Division with Operand Scaling” in *IEEE Transactions on Computers*, September 1990
- [7] M.D.Ercegovac, T.Lang, “Very High Radix Division Root with Selection by Rounding and Prescaling,” *IEEE Transactions on Computers*, Vol C-43, 1994.
- [8] S.F.Oberman, H.Al-twaijry, and M.J.Flynn, “The SNAP Project: Design of Floating Point Arithmetic Units” in proceedings of the 13th Symposium on Computer Arithmetic, 1997
- [9] E.de Angel and E.E.Swartzlander, Jr. “Low Power parallel multipliers” in *VLSI signal Processing, IX* [workshop on], 1996
- [10] J.E. Stine, A.Phadke, S.Tike “A Recursive-Divide Architecture for Multiplication and Division” in *ISCAS* 2011
- [11] Virage Standard Cell Book, Virage Logic Corporation
- [12] Design Compiler user guide, Synopsys
- [13] IC compiler design planning user guide
- [14] ModelSim Manual, Mentor Graphics

APPENDICES

Appendix A: It contains RTL implementation of different division architectures and proposed combinational multiplier/divide unit.

Appendix A

RTL implementation of Radix2 Division

```
module divide2 #(parameter WIDTH = 32, WIDTHHP2 = WIDTH + 2,
WIDTHM2 = WIDTH - 2) (
    input          clk, clear, state0,
    input [WIDTH-1:0]  D, N,
    output         [WIDTH-1:0]  Quo,
    output         [WIDTH+1:0]  sum, carry,
    output         [1:0]        q);

    wire [3:0]          w;
    wire [WIDTH+1:0]   Mux3Out, Mux2Out_ws, Mux2Out_wc,
    OnesCompD, ws, wc;
    inv #(WIDTHHP2) i1 (OnesCompD, {2'b00,D});
    mux31 #(WIDTHHP2) m3 (Mux3Out, {{WIDTHHP2}{1'b0}}, {2'b00,
    D}, OnesCompD, q);
    mux21 #(WIDTHHP2) m2 (Mux2Out_ws, {sum[WIDTH:0], 1'b0},
    {2'b00, N}, state0);
    reg_pos #(WIDTHHP2) r1 (ws, Mux2Out_ws, clk, clear);
    mux21 #(WIDTHHP2) m4 (Mux2Out_wc, {carry[WIDTH:0], 1'b0},
    {{WIDTHHP2}{1'b0}}, state0);
    reg_pos #(WIDTHHP2) r2 (wc, Mux2Out_wc, clk, clear);
    cpa #(4) c1 ( .sum(w), .a(ws[WIDTH+1:WIDTHM2]),
    .b(wc[WIDTH+1:WIDTHM2]));
    qst qd1 (q, w[3], w[2], w[1]); // q = {+1, -1} else q = 0
    otf_mux #(WIDTH) ckt2 (.Q(Quo), .quot(q),
    .SignRemainder(w[3]),.clk(clk),
    .state0(state0)); // negedge
    csa #(WIDTHHP2) d1 (.sum(sum), .carry(carry), .a(ws),
    .b(Mux3Out), .c({wc[WIDTH+1:1],q[1]}));
endmodule
```

RTL implementation of Radix4 Division

```

module divide4 #(parameter WIDTH = 32, WIDTHHP3 = WIDTH + 3)
(
    input          clk, clear, state0,
    input  [WIDTH-1:0]  N, D,

    output  [WIDTH+2:0]  sum, carry,
    output  [3:0]        q,
    output  [WIDTH-1:0]  Qstar);

    wire [WIDTH-1:0]      QMstar;
    wire [WIDTH+2:0]      sumN, carryN, sumN2, carryN2,
                        divi1, divi2, divi1c, divi2c, divel,
                        mdivi_temp, mdivi;
    wire [7:0]            qtotal;

    assign divi1 = {3'b0, D};
    assign divi2 = {2'b0, D, 1'b0};
    assign divel = {3'b0, N};

    inv #(WIDTHHP3) inv1 (divi1c, divi1);
    inv #(WIDTHHP3) inv2 (divi2c, divi2);
    mux21 #(WIDTHHP3) mux1 (sumN, {sum[WIDTH:0], 2'b0},
                        divel, state0);
    mux21 #(WIDTHHP3) mux2 (carryN, {carry[WIDTH:0], 2'b0},
                        {{WIDTHHP3}{1'b0}}, state0);
    reg_pos #(WIDTHHP3) reg1 (sumN2, sumN, clk, clear);
    reg_pos #(WIDTHHP3) reg2 (carryN2, carryN, clk, clear);
    cpa #(8) cpa1 (.sum(qtotal), .a(carryN2[WIDTH+2:WIDTH-
                        5]), .b(sumN2[WIDTH+2:WIDTH-
                        5]));
    qst4 pd1 (q, qtotal[7:1], divi1[WIDTH-2:WIDTH-4]);
    // q = {+2, +1, -1, -2} else q = 0
    otf_mux #(WIDTH) otf1 (.Qstar(Qstar), .QMstar(QMstar),
                        .q(q), .clk(clk),
                        .state0(state0)); // negedge

    assign ulp = q[2] | q[3];
    nor n1 (zero, q[3], q[2], q[1], q[0]);
    mux4lhx #(WIDTHHP3) mux3 (.y(mdivi_temp), .a(divi2c),
                        .b(divi1c), .c(divi1),
                        .d(divi2), .sel(q));

```

```

mux21 #(WIDTHHP3) mux4 (mdivi, mdivi_temp,
                        {{WIDTHHP3}{1'b0}}, zero);
csa #(WIDTHHP3) csal (.sum(sum), .carry(carry),
                    .a(mdivi), .b(sumN2),
                    .c({carryN2[WIDTHHP3-1:1],
                        ulp}));
endmodule

```

RTL implementation of Radix16 Division

```

module dividel6 #(parameter WIDTH = 32, WIDTHHP5 = WIDTH +
5, WIDTHHP4 = WIDTH + 4, WIDTHM5 = WIDTH - 5) (
    input      [WIDTH-1:0]      N, Din,
    input      clk, clear, state0, statel,
    output     [WIDTH-1:0]      Qstar);

    wire [WIDTHHP4:0]      D, D32, D16, D8, D4, D2, D1, D0,
    M4_dop, M3_dop, M2_dop, M1_dop,
    X, X32, X16, X8, X4, X2, X1, X0,
    M4_xop, M3_xop, M2_xop, M1_xop,
    M4_op, M3_op, M2_op, M1_op,
    init_rws, init_rwc, rws2, rwc2,
    Z, Z_op, Z8, Z4, Z0, bar4Z,
    bar8Z, qh_op, Z2, Z1, bar1Z,
    bar2Z, ql_op, a_op, b_op, c_op,
    d_op, Sum_1, Sum_2, Carry_1,
    Carry_2, rws, rwc;

    wire [9:0]      quo;
    wire [2:0]      sel_qh, sel_ql, sel_M4, sel_M3,
    sel_M2;
    wire [1:0]      init_sel;
    wire      ulp_qh, ulp_ql;

    assign X = {5'b0,N};
    assign D = {5'b0,Din};

    assign D32 = {1'b0, D[WIDTHHP4:1]};
    assign D16 = {2'b0, D[WIDTHHP4:2]};
    assign D8  = {3'b0, D[WIDTHHP4:3]};
    assign D4  = {4'b0, D[WIDTHHP4:4]};
    assign D2  = {5'b0, D[WIDTHHP4:5]};
    assign D1  = {6'b0, D[WIDTHHP4:6]};
    assign D0  = {{WIDTHHP5}{1'b0}};

```

```

mux51 #(WIDTHHP5) mux_M4d (.y(M4_dop), .a(D8), .b(D4),
                          .c(D2), .d(D1), .e(D0),
                          .sel(sel_M4));
mux71 #(WIDTHHP5) mux_M3d (.y(M3_dop), .a(D32),
                          .b(D16), .c(D8), .d(D4),
                          .e(D2), .f(D1), .g(D0),
                          .sel(sel_M3));
mux71 #(WIDTHHP5) mux_M2d (.y(M2_dop), .a(D32),
                          .b(D16), .c(D8), .d(D4),
                          .e(D2), .f(D1), .g(D0),
                          .sel(sel_M2));

assign M1_dop = D;
assign X32 = {1'b0, X[WIDTHHP4:1]};
assign X16 = {2'b0, X[WIDTHHP4:2]};
assign X8  = {3'b0, X[WIDTHHP4:3]};
assign X4  = {4'b0, X[WIDTHHP4:4]};
assign X2  = {5'b0, X[WIDTHHP4:5]};
assign X1  = {6'b0, X[WIDTHHP4:6]};
assign X0  = {{WIDTHHP5}{1'b0}};

mux51 #(WIDTHHP5) mux_M4x (.y(M4_xop), .a(X8), .b(X4),
                          .c(X2), .d(X1), .e(X0),
                          .sel(sel_M4));
mux71 #(WIDTHHP5) mux_M3x (.y(M3_xop), .a(X32),
                          .b(X16), .c(X8), .d(X4),
                          .e(X2), .f(X1), .g(X0),
                          .sel(sel_M3));
mux71 #(WIDTHHP5) mux_M2x (.y(M2_xop), .a(X32),
                          .b(X16), .c(X8), .d(X4),
                          .e(X2), .f(X1), .g(X0),
                          .sel(sel_M2));

assign M1_xop = X;

//preparation of 2:1 MUX for all the M.d and M.x
//2:1_MUX for selection M4_op
mux21 #(WIDTHHP5) mux_M4 (.y(M4_op), .a(M4_dop),
                          .b(M4_xop), .sel(statel));
mux21 #(WIDTHHP5) mux_M3 (.y(M3_op), .a(M3_dop),
                          .b(M3_xop), .sel(statel));
mux21 #(WIDTHHP5) mux_M2 (.y(M2_op), .a(M2_dop),
                          .b(M2_xop), .sel(statel));
mux21 #(WIDTHHP5) mux_M1 (.y(M1_op), .a(M1_dop),
                          .b(M1_xop), .sel(statel));

//preparation of 5:1 MUX for qh
assign Z8 = {Z[WIDTH+1:0], 3'b0};
assign Z4 = {Z[WIDTH+2:0], 2'b0};
assign Z0 = {{WIDTHHP5}{1'b0}};
inv #(WIDTHHP5) i1 (.y(bar4Z), .in(Z4));
inv #(WIDTHHP5) i2 (.y(bar8Z), .in(Z8));

```



```

mux51 #(WIDTHHP5) mux_gh (.y(qh_op), .a(bar8Z),
                        .b(bar4Z), .c(Z0), .d(Z4),
                        .e(Z8), .sel(sel_gh));

//preparation of 5:1 MUX for ql
assign Z2 = {Z[WIDTH+3:0], 1'b0};
assign Z1 = Z;
assign Z0 = {{WIDTHHP5}{1'b0}};
inv #(WIDTHHP5) i3 (.y(bar1Z), .in(Z1));
inv #(WIDTHHP5) i4 (.y(bar2Z), .in(Z2));
mux51 #(WIDTHHP5) mux_ql (.y(ql_op), .a(bar2Z),
                        .b(bar1Z), .c(Z0), .d(Z1),
                        .e(Z2), .sel(sel_ql));

// Flow starts HERE
assign rws = {Sum_2[WIDTH:0],4'b0};
assign rwc = {Carry_2[WIDTH:0],4'b0};
assign init_sel = {~state0, state0 & state1};

mux21 #(WIDTHHP5) mux_a (.y(a_op), .a(rwc2), .b(M4_op),
                        .sel(state0));
mux21 #(WIDTHHP5) mux_b (.y(b_op), .a(rws2), .b(M3_op),
                        .sel(state0));
mux21 #(WIDTHHP5) mux_c (.y(c_op), .a(qh_op),
                        .b(M2_op), .sel(state0));
//preparation of 2:1 MUX for qh vs M2
mux21 #(WIDTHHP5) mux_d (.y(d_op), .a(ql_op),
                        .b(M1_op), .sel(state0));
//preparation of 2:1 MUX for ql vs M1
csa #(WIDTHHP5) csa1 (.sum(Sum_1), .carry(Carry_1),
                    .a(c_op), .b(b_op),
                    .c({a_op[WIDTHHP4:1],
                        ulp_gh}));
//CSA adder a_op, b_op, c_op
csa #(WIDTHHP5) csa2 (.sum(Sum_2), .carry(Carry_2),
                    .a(d_op), .b(Sum_1),
                    .c({Carry_1[WIDTHHP4:1],
                        ulp_ql}));
//CSA adder Sum_1, Carry_1, d_op
cpa #(37) cpa1 (.sum(Z_op), .a(Carry_2), .b(Sum_2));
//preparation of CPA for finding final value of Md
cpa #(10) cpa2 (.sum(quo), .a(rwc2[WIDTHHP4:WIDTHM5]),
                .b(rws2[WIDTHHP4:WIDTHM5]));
//preparation of CPA for the QST

mux31 #(WIDTHHP5) mux_init1 (.y(init_rws),
                            .a({{WIDTHHP5}{1'b0}}),

```

```

        .b(Sum_2), .c(rws),
        .sel(init_sel));

mux31 #(WIDTHHP5) mux_init2 (.y(init_rwc),
                            .a({{WIDTHHP5}{1'b0}}),
                            .b(Carry_2), .c(rwc),
                            .sel(init_sel));
reg_pos #(WIDTHHP5) reg1 (.y(rws2), .in(init_rws),
.clock(clk), .clear(clear));
//registers for rws, rwc
reg_pos #(WIDTHHP5) reg2 (.y(rwc2), .in(init_rwc),
.clock(clk), .clear(clear));
sel_m sel_m(sel_M4, sel_M3, sel_M2, D[WIDTH-2:WIDTHM5-
2]); //M selection table
qh_sel qh_sel (sel_qh, ulp_qh, quo[9:4]);
//qh/ql selection table
ql_sel ql_sel (sel_ql, ulp_ql, quo[9:4]);

reg_pos #(WIDTHHP5) reg3 (.y(Z), .in(Z_op),
.clock(state1 & state0), .clear(clear));
//Latching the value - z = M.d at falling edge of state1
otf_mux #(WIDTH) adj1 (.d(quo[9:4]), .clk(clk),
.clear(clear), .Qstar(Qstar));
//On the fly conversion
endmodule

```

RTL implementation of Radix512 division

```

module divide512 (p0, p1, Xin, Din, clk, state0, state1,
clear, clear2, ctrl, zctrl, Qstar,pre_out);

input      [27:0]      Xin, Din;
input      clk, state0, state1, clear,
clear2, ctrl;

input [1:0] zctrl;
wire [13:0] quo_in;
output [27:0] Qstar;
output [43:0] pre_out;
input [18:0] p0,p1;

wire [29:0] sel_op, mux_selop, pre_op;
wire [29:0] X, D;
wire [43:0] ws, wc;
// 1 sign, 9 integer, 23 fractional bits
wire [17:0] Mrec_ab, Mrec_cd;
wire [7:0] P2, P1, M1, M2;
wire [43:0] pprod0, pprod1, pprod2, pprod3,
pprod4, pprod5, pprod6, pprod7;

```

```

wire [43:0]      pre_cpaop, ws_reg, wc_reg, rwc_regip,
                rws_regip;
wire [17:0]     sum1, carry1;

wire [13:0]     sbtm_sr, sbtm_cr, sbtm_s, sbtm_c;

assign X = {2'b0, Xin};    assign D = {2'b0, Din};

// SBTM input and its register - positive clock

sbtm_add sbtm1 (sbtm_s, sbtm_c, p0, p1);
reg14 reg14_1 (sbtm_sr, sbtm_s, clk, clear);
reg14 reg14_2 (sbtm_cr, sbtm_c, clk, clear);

mux21x18 mux_rec1 (Mrec_ab, {4'b0, rws_regip[39:26]},
                 {2'b0, sbtm_sr, 2'b0}, state0);
mux21x18 mux_rec2 (Mrec_cd, {4'b0, rwc_regip[39:26]},
                 {2'b0, sbtm_cr, 2'b0}, state0);
recoder recoder(P2, P1, M1, M2, Mrec_ab, Mrec_cd);
//[7:0] Preamble      [4:0] Recursion
// Select other operand
mux31x30 mux_select (mux_selop, D, X, pre_op, zctrl);
//select D, X, or Z on pos edge | in sync with sbtm

reg30 reg1 (sel_op, mux_selop, clk & clear2, clear);

pprod_gen pprod_gen (pprod0, pprod1, pprod2, pprod3,
                    pprod4, pprod5, pprod6, pprod7, P2,
                    P1, M2, M1, sel_op, state0);
csa_tree csa_tree(ws, wc, pprod0, {pprod1[41:0],
M2[0], M1[0]}, {pprod2[39:0], M2[1],
M1[1], 2'b0},
{pprod3[37:0], M2[2], M1[2], 4'b0},
{pprod4[35:0], M2[3], M1[3], 6'b0},
{pprod5[33:0], M2[4], M1[4], 8'b0},
{pprod6[31:0], M2[5], M1[5], 10'b0},
{pprod7[29:0], M2[6], M1[6], 12'b0},
rwc_regip, rws_regip, state0, P2[5],
P1[5]);

reg44 regws (ws_reg, ws, clk, clear);
//Register for rws
reg44 regwc (wc_reg, wc, clk, clear);

// assign ws_out=ws;
// assign wc_out=wc;
// For Z
// assign pre_cpaop= wc_reg+ws_reg;

cpa44 cpa44 ( pre_cpaop, ws_reg,wc_reg);

```

```

assign pre_op = pre_cpaop[42:13];
assign pre_out=pre_cpaop;

mux21x44 mux1 (rws_regip, {{4{ws_reg[43]}}},
              ws_reg[43:13], 9'b0}, {ws_reg[34:0],
              9'b0}, ctrl);
mux21x44 mux2 (rwc_regip, {{4{wc_reg[43]}}},
              wc_reg[43:13], 9'b0}, {wc_reg[34:0],
              9'b0}, ctrl);
csa18 csa18(sum1, carry1, rws_regip[43:26],
           rwc_regip[43:26],
           {16'b0,1'b1,1'b0});
//csa to add rws, rwc,0.5
cpa14 cpa14_2 (quo_in, sum1[13:0], carry1[13:0]);
otf_mux otf_mux (quo_in[11:2], clk, Qstar, clear2);

endmodule // divide512

```

RTL implementation of Radix512 mult/div unit

```

module divide512 (Xin, Din, Mult, Mulcnd_in, clk, state0,
                 statel, clear, clear2, ctrl, zctrl, opern, Qstar, mul_op);

    input      [27:0]    Xin, Din;
    input [27:0]    Mult, Mulcnd_in;
    input      clk, state0, statel, clear, clear2,
ctrl, opern;
    input [1:0] zctrl;
    output     [27:0]    Qstar;
    output     [55:0]    mul_op;

    wire [13:0]    quo_in;
    wire [29:0]    operand_r, mux_selop, pre_op, operand;
    wire [29:0]    X, D, Mulcnd;
    wire [43:0]    ws, wc;
    wire [17:0]    Mrec_ab, Mrec_cd;
    wire [17:0]    Multrec_ab, Multrec_abr, Divrec_ab,
Divrec_cd;
    wire [7:0]    P2, P1, M1, M2;
    wire [43:0]    pprod0, pprod1, pprod2, pprod3,
pprod4, pprod5, pprod6, pprod7;
    wire [43:0]    pre_cpaop, ws_reg, wc_reg, rwc_regip,
rws_regip, mult_ws, mult_wc;
    wire [17:0]    sum1, carry1;
    wire [13:0]    sbtm_s, sbtm_c, sbtm_sr, sbtm_cr;

```

```

wire [15:0]      mul_l_r, cpa_mul_l;

assign X = {2'b0, Xin};      assign D = {2'b0, Din};

assign Mulcnd = {2'b0, Mulcnd_in};

// SBTM input and its register - positive clock
sbtm_add sbtm1 (Din[26:13], sbtm_s, sbtm_c);
reg14 reg14_1 (sbtm_sr, sbtm_s, clk, clear);
reg14 reg14_2 (sbtm_cr, sbtm_c, clk, clear);

// For Division Recoding
mux21x18 mux_div1 (Divrec_ab, {4'b0, rws_regip[39:26]},
                  {2'b0, sbtm_sr, 2'b0}, state0);
mux21x18 mux_div2 (Divrec_cd, {4'b0, rwc_regip[39:26]},
                  {2'b0, sbtm_cr, 2'b0}, state0);

// For MultL/ MultH Recoding
mux21x18 mux_mul (Multrec_ab, {Mult[15:0], 2'b0},
                 {4'b0, Mult[27:14]}, state1);
// Send lower half when state1=0
reg18 reg18_1 (Multrec_abr, Multrec_ab, clk, clear);
mux21x18 mux_rec1 (Mrec_ab, Divrec_ab, Multrec_abr,
                  opern);
// opern = 0   division
mux21x18 mux_rec2 (Mrec_cd, Divrec_cd, 18'b0, opern);
// opern= 1 multiplication

recoder recoder(P2, P1, M1, M2, Mrec_ab, Mrec_cd,
                state0, state1, opern, hi8, gi8);
// Select other operand
mux31x30 mux_select1 (mux_selop, D, X, pre_op, zctrl);
//select D, X, or Z on positive edge | in sync with sbtm
mux21x30 mux_select2 (operand, mux_selop, Mulcnd,
                    opern);

reg30 reg1 (operand_r, operand, clk & clear2, clear);

pprod_gen pprod_gen (pprod0, pprod1, pprod2, pprod3,
                    pprod4, pprod5, pprod6, pprod7,
                    P2, P1, M2, M1, operand_r, state0,
                    opern, hi8, gi8);

csa_tree csa_tree (ws, wc, pprod0, {pprod1[41:0],
                                    M2[0], M1[0]},
                  {pprod2[39:0], M2[1], M1[1], 2'b0},
                  {pprod3[37:0], M2[2], M1[2], 4'b0},
                  {pprod4[35:0], M2[3], M1[3], 6'b0},
                  {pprod5[33:0], M2[4], M1[4], 8'b0},

```

```

        {pprod6[31:0], M2[5], M1[5], 10'b0},
        {pprod7[29:0], M2[6], M1[6], 12'b0},
        rwc_regip, rws_regip, mult_ws, mult_wc,
        state0, P2, P1, M2, M1, opern,
        operand_r);

    reg44 regws (ws_reg, ws, clk, clear);
//Register for rws
    reg44 regwc (wc_reg, wc, clk, clear);
// For Z
    cpa16 cpa16 ( cpa_mul_l, carry, ws_reg[15:0],
                 wc_reg[15:0], carry_r);
    reg16 reg_mul_l (mul_l_r, cpa_mul_l, (clk & opern),
                   clear);
    dff dff (carry_r, carry, (clk & opern), clear);

    cpa44 cpa44 (pre_cpaop, cout, ws_reg, wc_reg,
                carry_r);

    assign pre_op = pre_cpaop[42:13];
    assign mult_ws = {{16{ws_reg[43]}}, ws_reg[43:16]};
    assign mult_wc = {16'b0, wc_reg[43:16]};

    mux21x44 mux1 (rws_regip, {{4{ws_reg[43]}},
                             ws_reg[43:13], 9'b0}, {ws_reg[34:0],
                             9'b0}, ctrl);
    mux21x44 mux2 (rwc_regip, {{4{wc_reg[43]}},
                             wc_reg[43:13], 9'b0}, {wc_reg[34:0],
                             9'b0}, ctrl);

    csa18 csa18(sum1, carry1, rws_regip[43:26],
                rwc_regip[43:26],
                {16'b0, 1'b1, 1'b0});
//csa to add rws, rwc, 0.5
    cpa14 cpa14_2 (quo_in, sum1[13:0], carry1[13:0]);
    otf_mux otf_mux (quo_in[11:2], clk, Qstar, clear2);

    assign mul_op = {pre_cpaop[39:0], mul_l_r};

endmodule

```

VITA

Amey P Phakde

Candidate for the Degree of

Master of Science

Thesis: **STUDY OF RECURSIVE DIVIDE ARCHITECTURES
IMPLEMENTATION FOR DIVISION AND MULTIPLICATION**

Major Field: Electrical Engineering

Biographical:

Education: Received Bachelor of Engineering degree in Electronics from University of Pune, Pune, India in August 2007. Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in Dec, 2011.

Experience: Employed by Oklahoma State University, Department of Electrical and Computer Engineering as a graduate teaching assistant in Fall'09, Spring'10 and Fall'10, and as a graduate research assistant ;Oklahoma State University, Department of Electrical and Computer Engineering, Spring'11 and Fall'11. Worked as Hardware Design Engineer, Aftek, Pune, India

Professional Memberships: Student Member,

Name: Amey P Phadke

Date of Degree: December, 2011

Institution: Oklahoma State University

Location: Stillwater, Oklahoma

Title of Study: STUDY OF RECURSIVE DIVIDE ARCHITECTURES
IMPLEMENTATION FOR DIVIDE AND MULTIPLICATION

Pages in Study: 69

Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Multipliers have been key and critical components for most application-specific and general-purpose computer architectures. However, these architectures have been transitioning towards multiple cores that can process large amounts of data through parallel approaches to computation. Unfortunately, traditional arithmetic functional units that worked well for single-core architectures have the side effect of incurring large amounts of area and power. Consequently, multi-core architecture need new ways of thinking about increased throughput to handle large amounts of data. This work discusses implementation of different divider algorithms and presents a recursive high radix divide unit that is modified to handle both multiplication and division targeted at multi-core architectures. Results are obtained with a 65nm technology and show a significant decrease in area and power while still maintaining a low total latency by utilizing high radix encoding within the functional unit.

ADVISER'S APPROVAL: Dr. James Stine
