Oklahoma State University

Department of Electrical and Computer Engineering

Spring 2014
ECEN 4233 - High Speed Computer Arithmetic
Instructor - Dr. James Stine

Final Project Report
May 03 2014

Student - Srinivas Sambaraju
CWID - 11461894

Abstract

The project discusses digital logic implementation for division, square root and complex division using Goldschmidt's iterative method of multiplicative division and square root. The end result is a very efficient data path with less expensive iterations compared to other iterative division methods. Since it is important to perform an error analysis when making such a bold statement, a comparison of the results with results from a java software program is also performed. The mathematics behind the method is explained and the data path and control logic for the hardware is also thoroughly discussed. The results are illustrated using various diagrams, tables and System Verilog code.

Introduction

In this project digital logic for Goldschmidt's division and square root is implemented. By using the multiplication and division functions available in the data path, complex division is also implemented. Since division is very expensive, most hardware devices implement this using software. The significance of the Goldschmidt's method is that it converts division into an equivalent multiplication problem which can be solved inexpensively. Goldschmidt's method is based on Newton Raphson's iterations and has quadratic convergence which means that after every iteration, the precision is doubled and also eliminates a final multiplication involved in Newton Raphson's method.

The data path and control logic is illustrated using a diagram. Muxes are used to select the desired signals (data) and a CSAM (carry save array multiplier) is used to perform 2's complement signed multiplication using a Baugh-Wooley multiplication formula. The result is rounded to nearest even, and the new value of Ki is computed. Registers are controlled through the control logic to read and write as desired. There is sequential and combinational logic involved to obtain an iterative solution to the division and square root. The Combinational logic involves multiplication, addition/subtraction and storing results in register. The Sequential logic is clock driven and multiple iterations are performed using the previously stored results.

For complex division, three operations are performed in a sequence to achieve the results. First a division followed by three multiplications and three addition's/subtraction's and finally two more divisions are performed. All the control logic is driven through the test bench using the appropriate delays so that data is stored to registers on the positive edge (@posedge) of a clock and read at the negative edge (@negedge) of the clock. The clock goes high for 10 ns and then low for 10 ns. This synchronization is used in the test bench to read and write the registers.

Different test benches are used for division, square root and complex division to keep things simple. Finally the data path, control logic, gate areas and delays, error analysis, and timing diagrams are discussed in that order.

Background

The multiplicative divide is discussed in this section first and then improvements are shown when Goldschmidt's method is used.

$$Q = \frac{N}{D} = N.\widetilde{\left(\frac{1}{D}\right)} \qquad (1)$$

Initial approximation for numbers in the range (a, b] is computed as

$$\frac{a+b}{2ab} \qquad (2)$$

Division is very expensive to calculate using hardware, a multiplicative division algorithm was proposed by Michael J Flynn.

$$Q = \frac{N}{D} \Rightarrow N.\frac{1}{D} \qquad (3)$$

Since calculating $\frac{1}{D}$ in hardware is very difficult, an approximation is used along with an iterative Newton Raphson's method to arrive at the solution. Newton Raphson's iteration for division is

$$X_{i+1} = X_i(2 - D.X_i) \qquad (4)$$

The division involves three steps and has a quadratic convergence, i.e., with every iteration the number of bits of approximation doubles. The steps are as follows

- multiplication $D.X_i$
- 2's complement $(2 - D.X_i)$
- multiplication $X_i * (2 - D.X_i)$

Error which also quadratically converges is calculated as

$$\varepsilon = \frac{1}{D} - D\varepsilon^2 \qquad (5)$$

Goldschmidt's Division

To further simplify this process, Goldschmidt in a 1961 thesis at MIT came up with a novel method that avoids a final multiplication with X_i. Goldschmidt method is a computationally efficient way of implementing Newton-Raphson's method. Below are the formulas for the Goldschmidt division algorithm.

$$Q = \frac{N.(K_0.K_1.K_2....K_n)}{D.(K_0.K_1.K_2....K_n)} \qquad (6)$$

These are the steps involved in Goldschmidt's division.

0) Get Initial Approximation (IA) for $\frac{1}{D}$
1) $N.K0$ … this becomes the quotient as we progress $(\frac{N}{D})$
2) $D.K0 = r0$ …this becomes 1.
3) $K1 = 2 - r0$ .. this is the 2's complement
4) Repeat steps 1 to 3 using the new value of Ki for required iterations to get the desired accuracy

Goldschmidt's Square Root

In this part, Goldschmidt's solution is extended to calculate square root and inverse square root.

$$\sqrt{x} = \frac{N}{D} = x.\widetilde{\left(\frac{1}{\sqrt{x}}\right)} \qquad (7)$$

The Newton Raphson's iteration $X_{i+1} = \frac{X_i\left(3-D.X_i^2\right)}{2}$ involves four steps

- Multiplication X_i^2
- Multiplication $D.X_i^2$
- 3's complement $(3 - D.X_i^2)$
- multiplication $X_i.(3 - D.X_i^2)$

In the project to avoid the division by 2, the implementation was modified to save one additional cycle of computation as below.

Iteration $X_{i+1} = \frac{X_i\left(3-D.X_i^2\right)}{2}$ involves three steps

- Multiplication X_i^2
- Multiplication $D.X_i^2$
- Multiplication $0.5.D.X_i^2$
- 1.5's complement $(1.5 - D.X_i^2)$
- multiplication $X_i.(1.5 - 0.5.D.X_i^2)$

The steps involved in the Goldschmidt square root algorithm are as follows

$$Q = \frac{N.(K_0.K_1.K_2....K_n)}{D.\left(K_0^2.K_1^2.K_2^2.....K_n^2\right)} \qquad (8)$$

0) Get Initial Approximation (IA) for $\frac{1}{\sqrt{x}} => \frac{\sqrt{a}+\sqrt{b}}{2\sqrt{ab}}$
1) $N.K0$ … this becomes the quotient as we progress ($\frac{N}{D}$)
2) Calculate K_0^2
3) $D.K_0^2 = r0$ …this becomes 1.
4) $K1 = (3 - r0)/2$ .. this is the new multiplicand
5) Repeat steps 1 to 3 using the new value of Ki for required iterations to get the desired accuracy

Applying Goldschmidt's formulas to calculate complex division

With some modifications in the data path used to calculate a division, a Complex division can also be performed as illustrated.

$$\frac{(a+ib)}{(c+id)} = \frac{a+b*\left({}^{d}/_{c}\right)}{c+d*\left({}^{d}/_{c}\right)} + i * \frac{b-a*\left({}^{d}/_{c}\right)}{c+d*\left({}^{d}/_{c}\right)} \qquad (9)$$

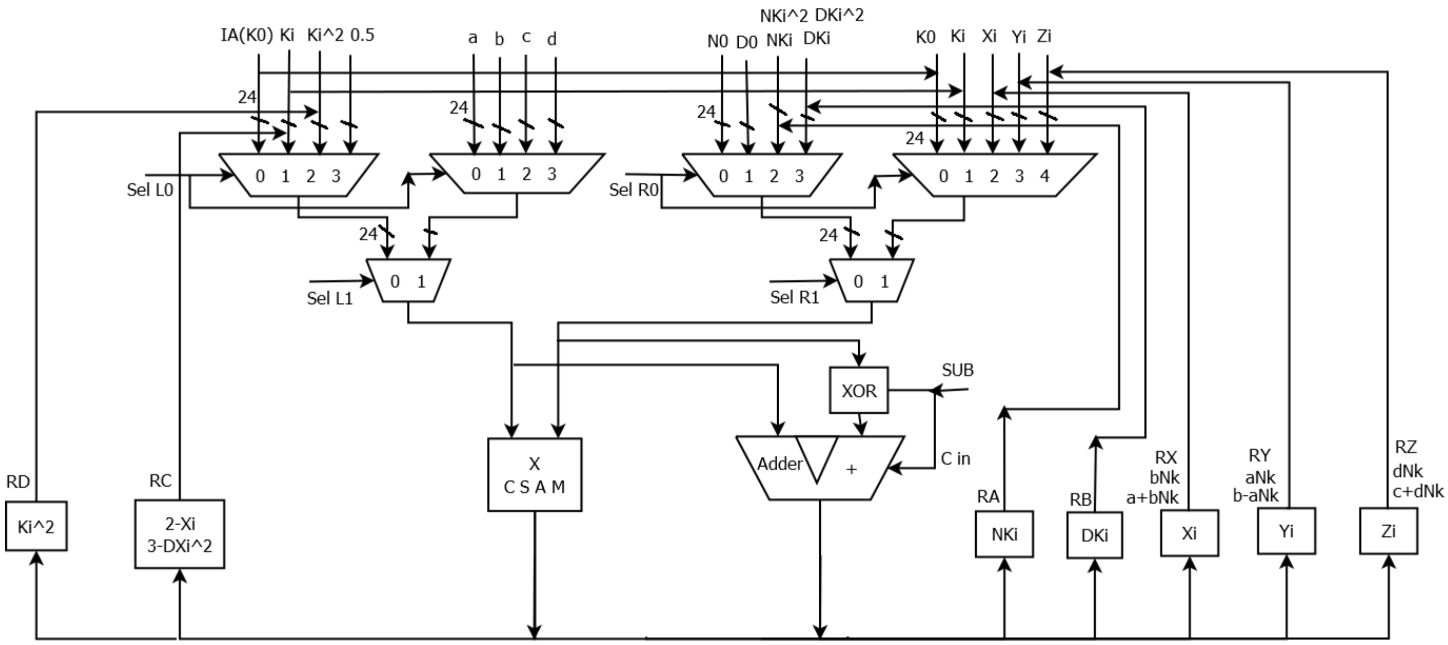
This problem can be solved by breaking down the operations as follows.

1) Calculate $\left({}^{d}/_{c}\right) = x_1$ N0 = d, D0 = c
2) $b * x_1$ = b.Nk = Rx
3) $a * x_1$ = a.Nk= Ry
4) $d * x_1$ = dNk= Rz
5) $a + b * x_1 = x_2$ = a+bNk = = Rx
6) $b - a * x_1 = x_3$ = b- a.Nk= Ry
7) $c + d * x_1 = x_4$ = c+da.Nk= Rz
8) ${}^{x_2}/_{x_4} = y_1$ = N0 = a+bNk, D0 = c+da.Nk ; Rx/Rz
9) ${}^{x_2}/_{x_4} = y_2$ N0 = b-aNk, D0 = c+da.Nk ; Ry/Rz
10) Finally write the result as $y_1 + i * y_2$

To compute complex division, a multiplier, an adder and a data path to compute Goldschmidt's division is required.

Results

The Data Path designed in the project is illustrated below.



Data Path for Goldschmidt Division, Square Root and Complex division

The muxes are used to select the desired signal based on control logic. CSAM computes the multiplication using Baugh-Wooley 2's complement multiplication. Adder is used for addition and subtraction with the help of XOR gate. The registers are read on negedge and data is stored on posedge of clock due to synchronization.

The control logic that is used in the test bench to drive the hardware according to the prescribed data path is tabulated below for division, square root and complex division in that order.

Cycle	M.cand	Multiplier	Sel L1	Sel L2	Sel R1	Sel R2	SUB	R_D	R_A	R_B	R_c	R_D	R_A	R_B	R_c (K_i)	R_X	R_Y	R_Z	R_X	R_Y	R_Z
0	K_0 (IA)	N	0	0	0	0	x	x	1	0	0	x	$N.K_0$	x	x	x	x	x	x	x	x
1	K_0 (IA)	D	0	0	1	0	x	x	0	1	1	x	$N.K_0$	$D.K_0$	$2-D.K_0$	x	x	x	x	x	x
2	K_1	$N.K_0$	1	0	2	0	x	x	1	0	0	x	$N.K_0.K_1$	$D.K_0$	$2-D.K_0$	x	x	x	x	x	x
3	K_1	$D.K_0$	1	0	3	0	x	x	0	1	1	x	$N.K_0.K_1$	$D.K_0.K_1$	$2-D.K_0.K_1$	x	x	x	x	x	x
4	K_2	$N.K_0.K_1$	1	0	2	0	x	x	1	0	0		$N.K_0.K_1.K_2$	$D.K_0.K_1$	$2-D.K_0.K_1$	x	x	x	x	x	x
5	K_2	$D.K_0.K_1$	1	0	3	0	x	x	0	1	1	x	$N.K_0.K_1.K_2$	$D.K_0.K_1.K_2$	$2-D.K_0.K_1.K_2$	x	x	x	x	x	x

Control logic for the implementation of Goldschmidt's division using Newton Raphson's method

Cycle	M.cand	Multiplier	Sel L1	Sel L2	Sel R1	Sel R2	SUB	R_D	R_A	R_B	R_c	R_D	R_A	R_B	R_c (K_i)	R_X	R_Y	R_Z	R_X	R_Y	R_Z
0	K_0 (IA)	K_0	0	0	0	1	x	1	x	x	x	K_0^2	x	x	x	x	x	x	x	x	x
1	K_0 (IA)	N	0	0	0	0	x	0	1	x	x	K_0^2	$N.K_0$	x	x	x	x	x	x	x	x
2	K_0^2	D	2	0	1	0	x	0	0	1	0	K_0^2	$N.K_0$	$D.K_0^2$	x	x	x	x	x	x	x
3	0.5	$D.K_0^2$	3	0	3	0	x	0	0	0	1	K_0^2	$N.K_0$	$D.K_0^2$	$1.5-0.5 * D.K_0^2$	x	x	x	x	x	x
4	K_1	K_1	1	0	1	1	x	1	0	0	0	K_1^2	$N.K_0$	$D.K_0^2$	K_1	x	x	x	x	x	x
5	K_1	$N.K_0$	1	0	2	0	x	0	1	0	0	K_1^2	$N.K_0.K_1$	$D.K_0^2$	K_1	x	x	x	x	x	x
6	K_1^2	$D.K_0^2$	2	0	3	0	x	0	0	1	0	K_1^2	$N.K_0.K_1$	$D.K_0^2.K_1^2$	K_1	x	x	x	x	x	x
7	0.5	$D.K_0^2$	3	0	3	0	x	0	0	0	1	K_1^2	$N.K_0.K_1$	$D.K_0^2.K_1^2$	$1.5-0.5 * D.K_1^2$	x	x	x	x	x	x
8	K_2	K_2	1	0	1	1	x	1	0	0	0	K_2^2	$N.K_0.K_1$	$D.K_0^2.K_1^2$	K_2	x	x	x	x	x	x
9	K_2	$N.K_0.K_1$	1	0	2	0	x	0	1	0	0	K_2^2	$N.K_0.K_1.K_2$	$D.K_0^2.K_1^2$	K_2	x	x	x	x	x	x
10	K_2^2	$D.K_0^2.K_1^2$	2	0	3	0	x	0	0	1	0	K_2^2	$N.K_0.K_1.K_2$	$D.K_0^2.K_1^2.K_2^2$	K_2	x	x	x	x	x	x

Control logic for the implementation of Goldschmidt's square root using Newton Raphson's method

Cycle	M.cand	Multiplier	SelL1	SelL2	SelR1	SelR2	SUB	R_D	R_A	R_B	R_c	R_D	R_A	R_B	R_c (K_i)	R_X	R_Y	R_Z	R_X	R_Y	R_Z
0	K_0 (IA)	N	0	0	0	0	x	x	1	0	0	x	$N.K_0$	x	x	x	x	x	x	x	x
1	K_0	D	0	0	1	0	x	x	0	1	1	x	$N.K_0$	$D.K_0$	$2-D.K_0$	x	x	x	x	x	x
2	K_1	$N.K_0$	1	0	2	0	x	x	1	0	0	x	$N.K_0.K_1$	$D.K_0$	$2-D.K_0$	x	x	x	x	x	x
3	K_1	$D.K_0$	1	0	3	0	x	x	0	1	1	x	$N.K_0.K_1$	$D.K_0.K_1$	$2-D.K_0.K_1$	x	x	x	x	x	x
0	b	NK	1	1	2	0	x	x	0	0	0	x	x	x	x	1	0	0	$b.NK$	x	x
1	a	NK	0	1	2	0	x	x	0	0	0	x	x	x	x	0	1	0	$b.NK$	$a.NK$	x
2	d	NK	3	1	2	0	x	x	0	0	0	x	x	x	x	0	0	1	$b.NK$	$a.NK$	$d.NK$
	Add 1	Add 2																			
0	a	$b.NK$	0	1	2	1	0	x	x	x	x	x	x	x	x	1	0	0	$a+b.NK$	x	x
1	b	$a.NK$	1	1	3	1	1	x	x	x	x	x	x	x	x	0	1	0	$a+b.NK$	$b-a.NK$	x
2	c	$d.NK$	2	1	4	1	0	x	x	x	x	x	x	x	x	0	0	1	$a+b.NK$	$b-a.NK$	$c+d.NK$
0	K_0 (IA)	$a+b.NK$	0	0	2	1	x	x	1	0	0		$(a+b.NK).K_0$	x	x	x	x	x	x	x	x
1	K_0	$c+d.NK$	0	0	4	1	x	x	0	1	1		$(a+b.NK).K_0$	$(c+d.NK).K_0$	$2-(c+d.NK).K_0$	x	x	x	x	x	x
2	K_1	$(a+b.NK).K_0$	1	0	2	0	x	x	1	0	0		$(a+b.NK).K_0.K_1$	$(c+d.NK).K_0$	$2-(c+d.NK).K_0$	x	x	x	x	x	x
3	K_1	$(c+d.NK).K_0$	1	0	3	0	x	x	0	1	1		$(a+b.NK).K_0.K_1$	$(c+d.NK).K_0.K_1$	$2-(c+d.NK).K_0.K_1$	x	x	x	x	x	x
0	K_0 (IA)	$b-a.NK$	0	0	3	1	x	x	1	0	0		$(b-a.NK).K_0$	x	x	x	x	x	x	x	x
1	K_0	$c+d.NK$	0	0	4	1	x	x	0	1	1		$(b-a.NK).K_0$	$(c+d.NK).K_0$	$2-(c+d.NK).K_0$	x	x	x	x	x	x
2	K_1	$(b-a.NK).K_0$	1	0	2	0	x	x	1	0	0		$(b-a.NK).K_0.K_1$	$(c+d.NK).K_0$	$2-(c+d.NK).K_0$	x	x	x	x	x	x
3	K_1	$(c+d.NK).K_0$	1	0	3	0	x	x	0	1	1		$(b-a.NK).K_0.K_1$	$(c+d.NK).K_0.K_1$	$2-(c+d.NK).K_0.K_1$	x	x	x	x	x	x

Control logic for the implementation of Complex Divide using Goldschmidt's division with Newton- Raphson's method

Area and Delay Analysis

Hardware component	Area of the unit(gates)	Delay (delta)
mux41_24	288	6
mux21_24	96	3
mux51_24	384	9
CSAM	5433	276
Rounding (CPA)		
- Bitwise_or (fanin 4)	7	3
- Or,or,and	3	3
- Add Round bit (CPA)	120	96
complement		
- negation	6	1
- Add bit (CPA)	120	96
- Add 2 24 bit numbers (has one Half adder and remaining are full adders)	212	142
Adder		
- XOR	96	3
- Add/subtract 2 24 bit (has all Full adders)	216	144

Area and delay of individual components used in the data path

The area can be determined after the combinational cycle and it remains constant after that. The delay changes based on the number of sequential cycles.

Hardware component	Area (gates)	Delay (delta)
Mux – SL0 (mux41_24)	288	6
Mux – SL0 (mux41_24)	288	6
Mux – SL1 (mux21_24)	96	3
Mux – SR0 (mux41_24)	288	6
Mux – SR0 (mux51_24)	384	9
Mux – SR1 (mux21_24)	96	3
CSAM	5433	276
Rounding (CPA)		
- Bitwise_or (fanin 4)	7	3
- Or,or,and	3	3
- Add Round bit (CPA)	120	96
complement		
- negation	6	1
- Add bit (CPA)	120	96
- Add 2 24 bit numbers (has one Half adder)	212	142
Final mux21_24 (store division or square root)	96	3
Total Division area and delay one multiplication cycle (only NK0) ---(a)	7437	653
Add all below delays to get total delay		
Total Division area and delay one complete multiplication cycle (NK0, DK0 and Ki) = 2* (a)---(b)	7437	***1306***
Total area and delay for 6 complete cycles to achieve 22 bits of precision = 6* (b)	7437	***7836***
Total area and delay for division	7437	***9142***

Area and Delay analysis for Division

One division cycle is equivalent to calculating NK0 and then DK0 along with K1. The number of cycles to get the results as per desired accuracy also contributes to the delay.

Nk0 calculation does not require the complement (2-DXi), but in this current design it is calculated anyway. After computing the complement, the mux decides if it it's a division or square root value and then stores Ki which is a redundant operation for the NK0 cycle. Because the Rki register is '0', the Ki is not stored, even though its calculated. This is an area of improvement.

Adder is not included in the analysis for division and square root as the test bench is different and there is no call to the adder. But in hardware, a mux should be used to completely avoid the adder during division and square root. This will also save power.

Hardware component	Area (gates)	Delay (delta)
Mux – SL0 (mux41_24)	288	6
Mux – SL0 (mux41_24)	288	6
Mux – SL1 (mux21_24)	96	3
Mux – SR0 (mux41_24)	288	6
Mux – SR0 (mux51_24)	384	9
Mux – SR1 (mux21_24)	96	3
CSAM	5433	276
Rounding (CPA)		
- Bitwise_or (fanin 4)	7	3
- Or,or,and	3	3
- Add Round bit (CPA)	120	96
complement		
- negation	6	1
- Add bit (CPA)	120	96
- Add 2 24 bit numbers (has one Half adder)	212	142
Final mux21_24 (store division or square root)	96	3
Total square root area and delay one multiplication cycle (only NK0) ---(a)	7437	653
Add all below delays to get total delay		
Total square root area and delay one complete multiplication cycle (K0^2, NK0, DK0^2, 0.5 DK0^2 for Ki) = 4* (a)---(b)	7437	***2612***
Total area and delay for 4 complete cycles to achieve 22 bits of precision = 6 * (b)	7437	***15672***
Total area and delay for square root	7437	***18284***

Area and Delay analysis for square root

The total delay includes the time taken for all individual multiplications and also the number of cycles required to achieve the desired accuracy of result with the hardware.

Hardware component	**Area (gates)**	**Delay (delta)**
Mux – SL0 (mux41_24)	288	6
Mux – SL0 (mux41_24)	288	6
Mux – SL1 (mux21_24)	96	3
Mux – SR0 (mux41_24)	288	6
Mux – SR0 (mux51_24)	384	9
Mux – SR1 (mux21_24)	96	3
CSAM	5433	276
Rounding (CPA)		
- Bitwise_or (fanin 4)	7	3
- Or,or,and	3	3
- Add Round bit (CPA)	120	96
Adder		
- XOR	96	3
- Add/subtract 2 24 bit (has all Full adder)	216	144
complement		
- negation	6	1
- Add bit (CPA)	120	96
- Add 2 24 bit numbers (has one Half adder)	212	142
mux21_24 (store division or square root)	96	3
Final mux21_24 (store multiplication or addition)	96	3
Total complex division area and delay one complex division/multiplication cycle -–(a)	7845	803
Add all below delays to get total delay		
Total complex division area and delay one complete cycle involves $\left({}^{d}/_{c}\right), b\left({}^{d}/_{c}\right), a\left({}^{d}/_{c}\right), d\left({}^{d}/_{c}\right), a + b\left({}^{d}/_{c}\right), b - a\left({}^{d}/_{c}\right), c + d\left({}^{d}/_{c}\right), \{a + b\left({}^{d}/_{c}\right)/c + d\left({}^{d}/_{c}\right)\}, \{b - a\left({}^{d}/_{c}\right)/c + d\left({}^{d}/_{c}\right)\}$		
Main division => $\left({}^{d}/_{c}\right)$ (2 *(a) * 5 cycles) – **(b)**	7845	***8030***
3 multiplications $b\left({}^{d}/_{c}\right), a\left({}^{d}/_{c}\right), d\left({}^{d}/_{c}\right)$ **(a)*3**	7845	***2409***
3 add/sub => $a + b\left({}^{d}/_{c}\right), b - a\left({}^{d}/_{c}\right), c + d\left({}^{d}/_{c}\right)$ **(a)*3**	7845	***2409***
2 divisions => $\{a + b\left({}^{d}/_{c}\right)/c + d\left({}^{d}/_{c}\right)\}, \{b - a\left({}^{d}/_{c}\right)/c + d\left({}^{d}/_{c}\right)\}$ **(2* (b))**	7845	***16060***
Total area and delay for complex division	**7845**	***28908***

Area and Delay analysis for complex division

The total delay includes time taken for a division, three multiplications, three additions and two final divisions.

Unit	Area in gates	delay
Division	7437	***9142***
Square root	7437	***18284***
Complex division	**7845**	***28908***

Total of Area and delay for to compute one division, square root or complex divide with enough iterative cycles desired accuracy

Error analysis for division: comparison of actual (N/D from java program with NK value from Verilog)

S. No		Division - N/D	Binary result	Cycles
1	Actual value from java	1.7612245082855224609375/1.90303391218185424804 6875 = 0.92548250256	00.111011001110110001 1011	6
	Value from verilog	IA=24'b0011_0000_0000_0000_0000_0000=0.75 Ans=0.9254825115203857421875	00.111011001110110001 1011	6
	Error		0.000000008960	
	#bits of error		-18.1639	
2	Actual value from java	1.5223456621170043945312 5/1.761224567890167236328125 = 0.8643677234649658	00.110111010100011100 1101	6
	Value from verilog	IA=24'b0011_0000_0000_0000_0000_0000=0.75 Ans=0.86436748504638671875	00.110111010100011100 1100	6
	Error		0.000000238419	
	#bits of error		-14.8827	
3	Actual value from java	1.2024554014205932617187 5/1.4025342464447021484375 = 0.8573448657989502	00.110110110111101011 1101	6
	Value from verilog	IA = 24'b0010_1100_0000_0000_0000_0000 Ans=0.856250047683715820312 5	00.110110110011001100 1101	6
	Error		0.001094818115	
	#bits of error		-6.45065	
4	Actual value from java	- 1.2524554729461669921875/1.4025342464447021484375 = - 0.8929946422576904	11.000110110110010010 1101	6
	Value from verilog	IA=24'b1000_0000_0000_0000_0000_0001	10.100000001011010000 1100	6
	Error		Results are wrong	
	#bits of error		Results are wrong	

Error analysis for Square root

S.No		Sqrt(N)	Binary result	Cycles
1	Actual value from java	sqrt(1.76122450828552246093 75) = 1.3271112442016602	01.0101001110111101100100	5
	Value from verilog	1.356464147567749023437 5	01.0101101101000001001111	5
	Error		0.029352903	
	#bits of error		-3.16185	
2	Actual value from java	Sqrt(1.5223455429077148437 5) = 1.2338337898254395	01.0011101111011100100010	5
	Value from verilog	1.233846664428710937 5	01.0011101111011101011000	5
	Error		1.28746E-05	
	#bits of error		-10.8937	
3	Actual value from java	Sqrt(1.2524554729461669921875) = 1.1191315650939941	01.0001111001111111011010	6
	Value from verilog	1.1242725849151611328125	01.0001111111010000010101	6
	Error		0.00514102	
	#bits of error		-4.90399	
4	Actual value from java	Sqrt(1.1525342464447021484375)= 1.073561429977417	01.0001001011010100111011	6
	Value from verilog	1.087577342987060546875	01.0001011001101011011110	6
	Error		0.014015913	
	#bits of error		-3.90105	

Error analysis for complex division

S.No		**Complex division**	**cycles**
1	Problem to solve	(1.001002+i*1.015625)/1.140625+i*0.94999)	
	From excel – no rounding	0.956019 + i* 0.094165	
	From verilog	0.963588476181030273437 5 + i* 0.082525491714477539062 5	0011110110101011101101111 + i* 0000010101001000000011001
	Problem to solve		
2	From excel – no rounding	(1+i)/(1.34125 + i* 0.8)	
	From verilog	0.877939 + i * 0.221919	
		0.87076091766357421875 + i* 0.15032482147216796875	0011011110111010100011 00 + i* 0000100110011110111011 00

Conclusion

Overall the programs work well with signed positive numbers but the results are not correct for negative numbers. The error analysis shows that the number of bits of precisions is less than 22 for most cases. Converting certain fractions from binary to decimal and then computing the error results in some loss of precision as some numbers cannot be represented exactly in binary from their decimal equivalents. But observing the binary results, some cases are very close to the desired accuracy. The errors may be occurring in certain areas where rounding is done or subtraction to get the next value of Ki is done. Mostly the error occurs in representing certain fraction numbers in binary form. The areas of improvement are listed below.

Observations

1) Pipelining the multiplier along with the CPA for the rounding logic was not implemented in this project. One reason was that even if a register was used to store the multiplication results before the rounding, it was not possible to adjust the control logic to achieve any reduction in time. This is an area of improvement for this project
2) Improving the precision and reducing the number of bits of error.
3) Because of the way the data path is designed, certain additional computations and delays come into the picture when they are not required. For example in division when NK0 is computed, there is no need to compute a (2-Nx) value, and introduce a delay. A mux may be used here to avoid this.
4) For negative numbers, the multiplication results are wrong. This is because of the way the calculation of $(2 - D.X_i)$ is implemented. The $D.X_i$ bits are inverted and a '1' is added. Then its added to 1000_0000_0000_0000_0000_0000 (2). This gives correct results for positive numbers but wrong results for negative numbers because for positive numbers the results is less than '2' whereas for negative numbers, it could be greater than '2' resulting in overflow. Since there is only integer bit used both in the 24 bit multiplicand and multiplier, '2' could be represented. One way to resolve this issue is to assign two integer bits in the multiplicand and multiplier.
5) When the multiplication is performed, adder is also adding the same numbers for complex divide. Initial experimentation to introduce a mux so that no control signal reaches the adder when its not needed were not fruitful due to increased complexities. If successful, this can also result in power savings in the hardware.
6) Automatically calculating the initial approximation for the numbers has not been implemented. In case of complex division, good approximations were hard coded as it was difficult to calculate automatically as its done in a software.

References

1. M. D. Ercegovac and J. M. Muller, Complex Square Root with Operand Prescaling: Los Angeles, CA, 2006.
2. Notes from ECEN 4233
3. http://fpgasimulation.com/systemverilog_primer/
4. http://www.ecs.umass.edu/ece/koren/arith/simulator/
5. http://www.cl.cam.ac.uk/teaching/1112/ECAD+Arch/files/SystemVerilogCheatSheet.pdf

Appendices

Cycle	SelL1	SelL2	SelR1	SelR2
0	000	0	000	0
1	000	0	001	0
2	001	0	010	0
3	001	0	011	0
4	001	0	010	0
5	001	0	011	0

Multiplexer select signal values for Division

Cycle	SelL1	SelL2	SelR1	SelR2
0	000	0	000	1
1	000	0	000	0
2	010	0	001	0
3	011	0	011	0
1	001	0	001	1
5	001	0	010	0
6	010	0	011	0
7	011	0	011	0
8	001	0	001	1
9	001	0	010	0
10	010	0	011	0

Multiplexer select signal values for Square Root

Cycle	SelL1	SelL2	SelR1	SelR2
0	000	0	000	0
1	000	0	001	0
2	001	0	010	0
3	001	0	011	0
0	001	1	010	0
1	000	1	010	0
2	011	1	010	0
0	000	1	010	1
1	001	1	011	1
2	010	1	100	1
0	000	0	010	1
1	000	0	100	1
2	001	0	010	0
3	001	0	011	0
0	000	0	011	1
1	000	0	100	1
2	001	0	010	0
3	001	0	011	0

Multiplexer select signal values for Complex Division

A	B	C	D	E	F	G	H	I	J
0.95	N	IA	0.75	d=1.875				pos b-aNk	neg b-aNk
1.140625	D			c=1.625			a	1.001002	1.5
							b	1.015625	1.25
0.832876712	N/D						c	1.140625	
							d	0.95	
q*K	r*K	2-D*Xi	TRUE	Error	#bits				
0.7125	0.855469	1.14453125	0.832877	0.120377	-3.05437				
0.815478516	0.979111	1.02088928	0.832877	0.017398	-5.84492				
0.832513276	0.999564	1.00043636	0.832877	0.000363	-11.426				
0.832876554	1	1.00000019	0.832877	1.59E-07	-22.5882				
0.832876712	1	1	0.832877	3.02E-14	-44.9125				
0.832876712	1	1	0.832877	1.11E-16	-53				
	d/c=Nk=	0.83287671							
bNK	0.84589								
aNk	0.833711		find	(1.001002+i*1.015625)/1.140625+i*0.95)					
dNk	0.791233								
a+bNk	1.846892								
b-aNk	0.181914								
c+dNk	1.931858								
a+bNk/c+dNk	0.956019								
b-aNk/c+dNk	0.094165								

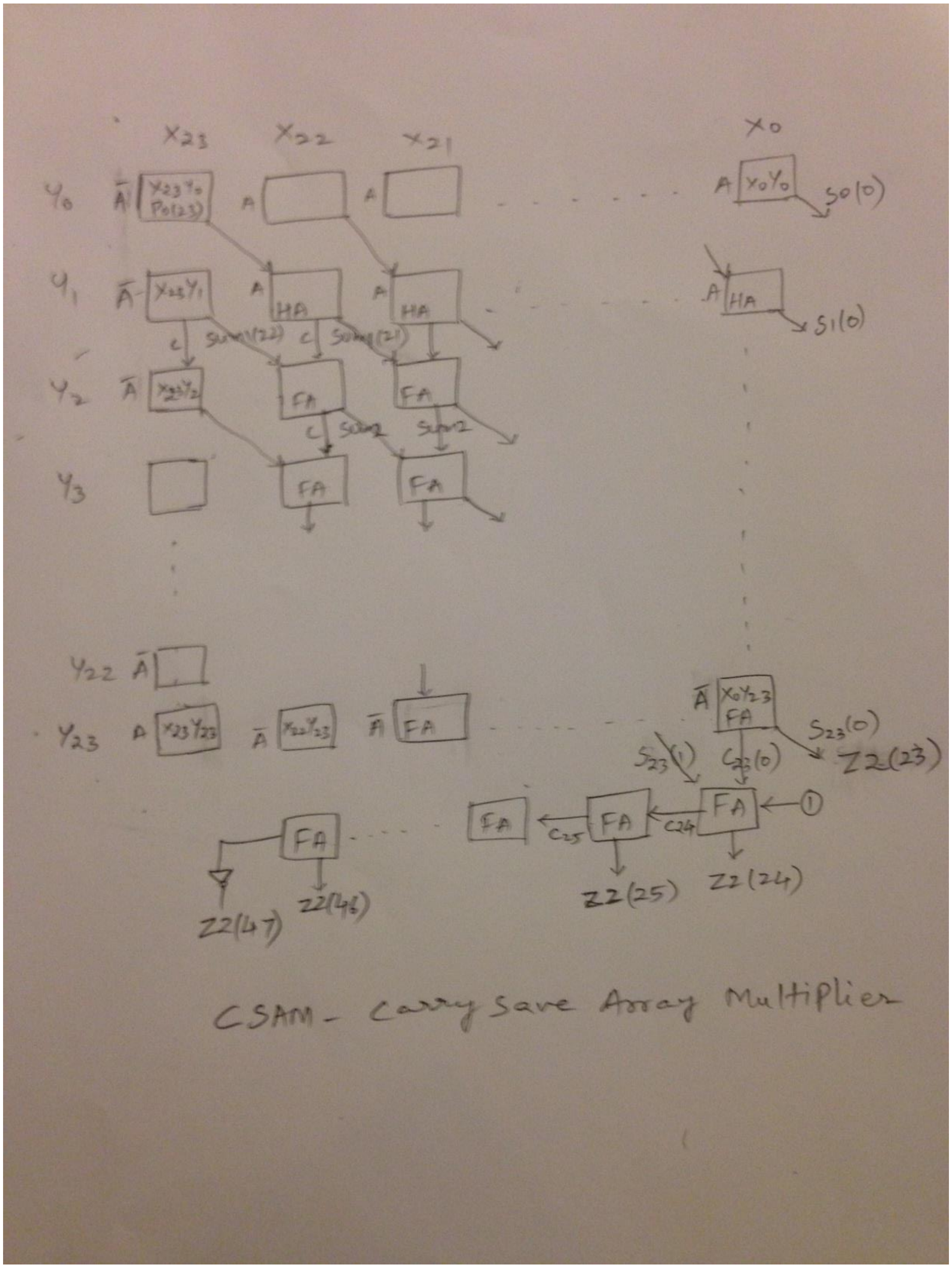
Complex division calculations in excel: Example - 1

0.8	N	IA	0.75	d=1.875				pos b-aNk	neg b-aNk
1.34125	D			c=1.625			a	1	1.5
							b	1	1.25
0.596458527	N/D						c	1.34125	
							d	0.8	
q*K	r*K	2-D*Xi	TRUE	Error	#bits				
0.6	1.005938	0.9940625	0.832877	0.232877	-2.10236				
0.5964375	0.999965	1.00003525	0.832877	0.236439	-2.08046				
0.596458527	1	1	0.832877	0.236418	-2.08059				
0.596458527	1	1	0.832877	0.236418	-2.08059				
0.596458527	1	1	0.832877	0.236418	-2.08059				
0.596458527	1	1	0.832877	0.236418	-2.08059				
	d/c=Nk=	0.59645853		find	(1+i)/1.34125+i*0.8)				
aNk	0.596459								
bNK	0.596459								
dNk	0.477167								
a+bNk	1.596459								
b-aNk	0.403541								
c+dNk	1.818417								
a+bNk/c+dNk	0.877939								
b-aNk/c+dNk	0.221919								

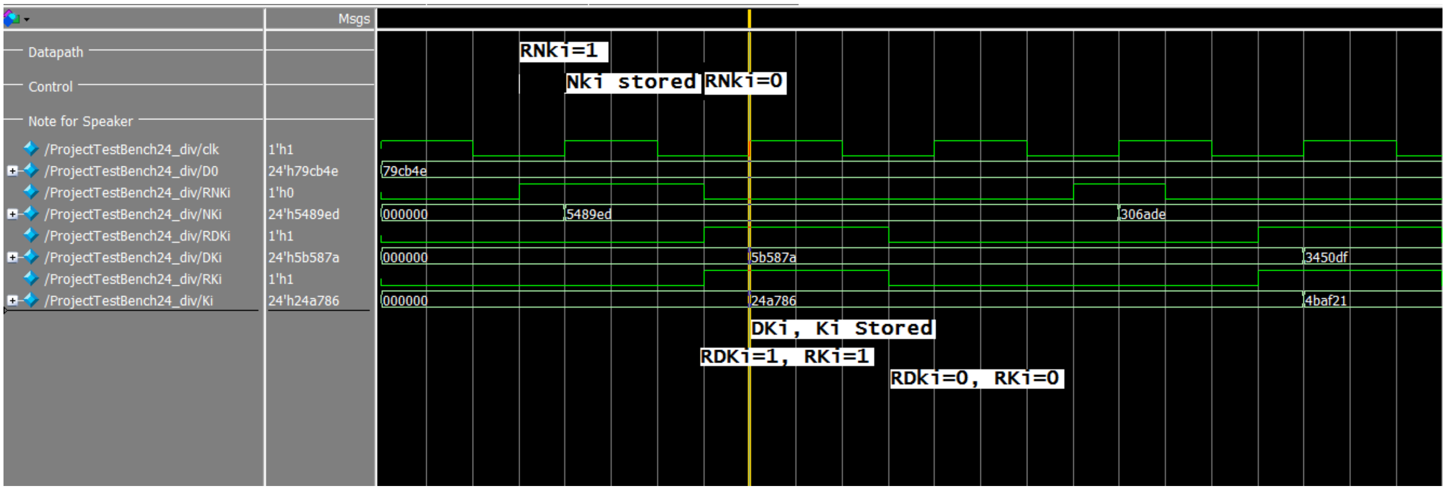
Complex division calculations in excel: Example - 2

Error Analysis				
java results	verilog results	error	ln(2)	# bits of error
Division				
0.925482503	0.925482512	8.96038E-09	0.693147181	-18.1639397
0.864367723	0.864367485	2.38419E-07	0.693147181	-14.8827251
0.857344866	0.856250048	0.001094818	0.693147181	-6.45065411
Square root				
1.327111244	1.356464148	0.029352903	0.693147181	-3.16185089
1.23383379	1.233846664	1.28746E-05	0.693147181	-10.893741
1.119131565	1.124272585	0.00514102	0.693147181	-4.90399089
1.07356143	1.087577343	0.014015913	0.693147181	-3.90104903

Error Analysis between actual results from java vs verilog



Carry save Array multiplier used in the project



Sample timing diagram for division illustrating how the registers and used for reading and writing