COMPATIBLE HARDWARE FOR DIVISION AND SQUARE ROOT [1]

George S. Taylor

Computer Science Division
University of California
Berkeley, California 94720

ABSTRACT

Hardware for radix four division and radix two square root is shared in a processor designed to implement the proposed IEEE floating-point standard. The division hardware looks ahead to find the next quotient digit in parallel with the next partial remainder. An 8-bit ALU estimates the next remainder's leading bits. The quotient digit look-up table is addressed with a truncation of the estimate rather than a truncation of the full partial remainder. The estimation ALU and the look-up table are asymmetric for positive and negative remainders. This asymmetry reduces the width of the ALU and the number of minterms in the logic equations for the look-up table. The square root algorithm obtains the correctly rounded result in about two division times using small extensions to the division hardware.

Introduction

An IEEE Computer Society working group has recommended a standard for binary floating-point arithmetic based on the proposal by Kahan, Coonen and Stone [1][2]. To investigate the feasibility of the *KCS* architecture, we are building a substitute floating-point accelerator for the DEC VAX 11/780 minicomputer [3]. The proposed standard requires that an implementation provide correctly rounded quotients and square roots. We found that radix four division hardware provides high speed at reasonable cost and, as a by-product, accommodates square root with minor extensions. This paper describes the algorithms and hardware for both operations.

Antecedents

We use nonrestoring division with redundant quotient digits and an irredundant partial remainder. Selection of another digit is overlapped with calculation of a partial remainder using the current digit. The theory and general implementation of higher radix nonrestoring division are explained by Atkins [4][5][6], based on the work of Robertson [7]. The Illiac III was an early machine which selected quotient digits using truncations of the divisor and partial remainder [8]. Tan reports [9] that certain IBM processors use a short precision ALU to estimate the next remainder. Quotient selection which is overlapped with the full width remainder iteration in this way is classified as QS2 by Kalaycioglu [10]. Baron's study [11] of several division schemes includes a radix four method similar to ours, but she recommends more redundancy in the quotient digit representation than we found to be optimal.

[1] This work was supported by the U. S. Department of Energy under contract DE-AT03-76SF00034, project agreement DE-AS03-79ER10358, and by the National Science Foundation under grant MCS78-07291. The author was supported by an NSF Graduate Fellowship.

Design Overview

Our division and square root board contains 150 ICs. Division is limited to a single board because of constraints on the size of the entire accelerator and the difficulty of passing wide operands between boards. 65 ICs on the addition/subtraction board also support the division operation. All parts are Schottky TTL except three programmable array logic (PAL[2]) packages which implement the quotient digit look-up table.

The accelerator supports three floating-point formats: single, double and (double) extended, with significand widths of 24, 53 and 64 bits, respectively. We use the term *significand* rather than fraction as a reminder that the significant digit field of normalized numbers in all formats has one bit to the left of the binary point. Single and double precision significands are left justified with zero fill before reaching the division board, so its data paths are designed for 64-bit operands. The operands are positive numbers because KCS uses sign-magnitude representation.

Internal data paths and functional units are slightly wider than the operands. Quotients in all formats are developed with three more bits than the operands have in order to allow unbiased rounding with an error $\leq \frac{1}{2}$ ULP (unit in the last place), as KCS requires. The three bits are called Guard, Round and Sticky. The Guard bit is used if the quotient is normalized by one bit before rounding, the maximum normalization that KCS permits. The Sticky bit is equal to zero only if the result is exact, i.e., all subsequent bits in an infinite precision result would be zero. Square roots have two more bits, Round and Sticky, than the operand because there is no need for normalization. The results are rounded after they leave the division board.

Register to register operation times are given in Table 1. Our division iteration produces twice as many bits per cycle, but has the same cycle time as the original VAX accelerator. The inner loop accounts for about two-thirds of the time in each instruction.

Table 1. -- Accelerator Instruction Times (μsec)

Instruction	Berkeley	VAX
Divide -- single	2.6	4.2
Divide -- double	4.8	8.8
Divide -- extended	5.8	---
Square root -- single	4.2	---
Square root -- double	7.6	---
Square root -- extended	9.2	---

[2] PAL is a trademark of Monolithic Memories.

CH1630-3/81/0000/0127$00.75 © 1981 IEEE

The VAX 11/780 microinstruction cycle time is 200 ns, with minor cycles at 50 ns intervals. For this reason, our inner loop times had to be 50, 100 or 200 ns. The 67-bit ALU in the main partial remainder data path takes 64 ns because we use small 74s381 parts. This lead us to design simple data paths in order to achieve a 100 ns step time. The microprogrammer can request either one or two division or square root steps per microinstruction cycle.

The hardware necessary for division alone is shown in Figure 1, while that for square root alone is shown in Figure 2. Shared functional units and data paths are readily identified. Some small data paths which ease the tasks of loading the operands and generating the Sticky bit are not included.

Division

The division procedure was chosen through a series of decisions:

(1) the radix – four
(2) divisor multiples -- one and two, but not three
(3) parallelism -- overlapped quotient and remainder
(4) width of remainder estimate -- eight bits
(5) estimation ALU operation -- asymmetric

The choices made imply that the quotient selection logic must observe seven remainder bits and four divisor bits (one known implicitly). Table 2 is a P/D plot of the logic, where g_{i+1} (to be defined later) plays the role of the partial remainder. The choices are explained next.

Radix Four

We had more board space available than the simplest radix two restoring division scheme requires. The original VAX accelerator produced one quotient bit per 100 ns by this method. We wished to make division faster in order to keep it in balance with addition and multiplication. Radix two with a redundant partial remainder and carry save addition could run at less than 100 ns per cycle, but the hardware cost would be substantial and square root could not be accommodated easily. Radix four at 100 ns per cycle provides equal or better performance at lower cost. Higher radix methods were unattractive because they required divisor multiples which could not be generated merely by shifting. Radix four can be implemented with a two-input ALU and and a two-input multiplexer in front of the divisor register, instead of functional units with three or more inputs as the higher radix methods would require.

We use a nonrestoring method rather than a restoring one so that the remainder iteration requires only a single data path and no backtracking. Backtracking would waste the gain made through lookahead quotient digit selection. The cost is that negative quotient digits must be combined in an ALU with the ones previously accumulated. Due to the low degree of redundancy in our quotient digit representation, this ALU must be the full width of the quotient rather than the width of one quotient digit. Since the ALU on the accelerator's addition board can be shared for this purpose, the division board contains only one full precision ALU.

ESTIMATED NEXT REMAINDER $\overline{g_{i+1}}$ (two's complement)

$g_1 g_2 g_3 g_4 \bullet g_3 g_2 g_1$

DIVISOR $\overline{d}$ (positive)	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
1.000	-2	-2	-2	A	-1	-1	Ø	Ø	1	1	2	2	2			
1.001	-2	-2	-2	B	-1	-1	Ø	Ø	1	1	C	2	2	2		
1.010	-2	-2	-2	-2	-1	-1	D	Ø	Ø	1	1	1	2	2	2	2
1.011	-2	-2	-2	-2	B	-1	-1	D	Ø	Ø	1	1	1	2	2	2	2
1.100	-2	-2	-2	-2	-1	-1	-1	Ø	Ø	Ø	E	1	1	C	2	2	2	2
1.101	-2	-2	-2	-2	-1	-1	-1	Ø	Ø	Ø	Ø	1	1	1	2	2	2	2	2
1.110	-2	-2	-2	-2	B	-1	-1	-1	Ø	Ø	Ø	Ø	1	1	1	2	2	2	2	2
1.111	-2	-2	-2	-2	-1	-1	-1	-1	Ø	Ø	Ø	Ø	1	1	1	1	2	2	2	2	2

g_{i+2}

A $-(2 - g2 \bullet g1)$
B $-(2 - g2)$
C $1 + g2$
D $-(1 - g2)$
E $g2$

TABLE 2 — P/D PLOT FOR QUOTIENT SELECTION LOGIC

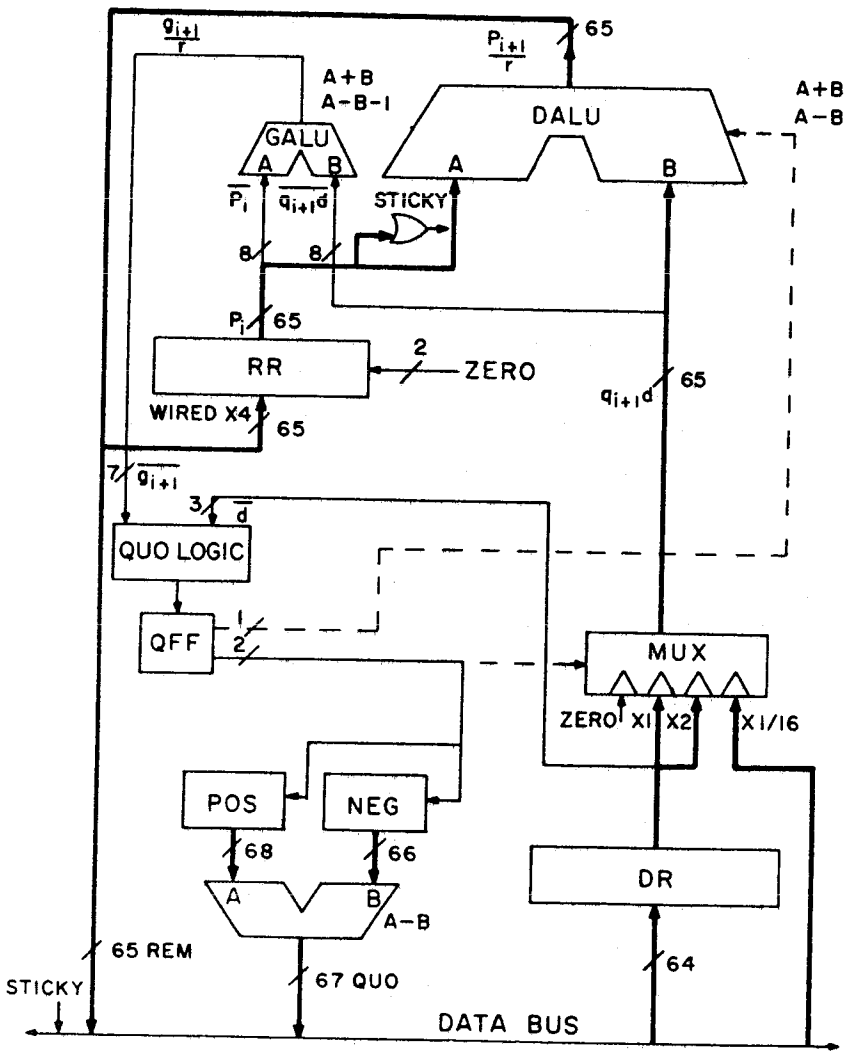

FIGURE 1 – DIVISION DATA PATHS

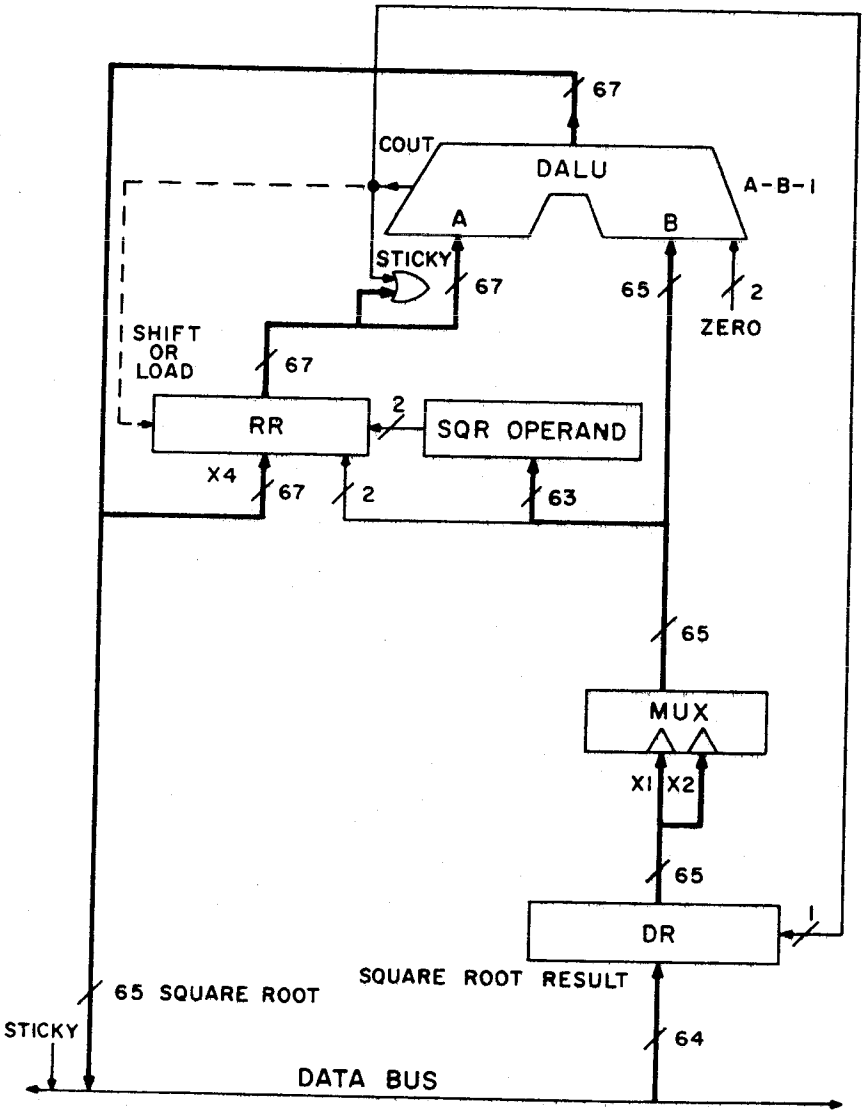

FIGURE 2 – SQUARE ROOT DATA PATHS

A square root algorithm shifts its remainder by twice the number of result bits found per iteration. A division algorithm, by contrast, shifts its remainder by the same number of bits as are generated during the cycle. Division and square root hardware can be merged if twice as many bits are produced per division step as per square root step. Thus radix four division hardware has the advantage of convenient reuse for square root.

Redundancy and Simple Divisor Multiples

A redundant quotient digit representation permits lookahead logic to select the next digit before the full precision next remainder is determined. For maximum redundancy in radix four, quotient digits could be selected from a set containing up to seven values: {-3, -2, -1, 0, 1, 2, 3}. Because the multiple of three times the divisor would be costly to generate, quotient digits are selected instead from the set {-2, -1, 0, 1, 2}. The cost is more complicated quotient selection hardware, but programmable logic limits the increase to a few ICs.

Parallelism

In the algorithm's inner loop, a quotient digit is selected and that multiple of the divisor is subtracted from the shifted previous remainder. Using notation suggested by Atkins and Kalaycioglu [12],

$$\frac{1}{r}p_{i+1} = p_i - q_{i+1}d \quad \text{for } i = 0,\ldots,m-1 \qquad (1a)$$

where

p_i = partial remainder after ith iteration

p_0 = dividend

r = radix = 4

q_i = ith quotient digit

d = divisor

m is the number of radix r digits in Q_m, the last quotient before rounding. Q_m has the form $q_1 . q_2 \cdots q_m$, with the binary point between q_1 and q_2.

In a logical sense, the quotient is accumulated during the iteration by resolving the negative quotient digits. Using Q_i for the partial quotient after the ith iteration,

$$\frac{1}{r}Q_{i+1} = Q_i + q_{i+1} \qquad (1b)$$

where $Q_0 = 0$. An equivalent procedure saves hardware in our design. The positive and negative q_i's are held in separate shift registers. At the end of the iteration, the negative register is subtracted from the positive one.

It is not possible first to select q_{i+1} and then carry out equation (1a) in 100 ns. If q_{i+1} were known immediately, the worst case delay to form the next full-width remainder would be:

Read q_{j+1} from flip-flop QFF	9 ns
Select $q_{j+1}d$ through MUX	18 ns
Add or subtract in DALU	64 ns
Setup time for p_{i+1} in RR	5 ns
Total	96 ns

In order to know q_{i+1} at the beginning of a cycle, it is calculated in parallel during the previous one. The GALU in Figure 1 uses truncations of p_i and q_{i+1} to guess quickly the leading bits of p_{i+1}. Then the quotient digit logic equations are evaluated using the guess. For later reference, define $\overline{p_i}$ and $\overline{q_{i+1}d}$ as the truncated inputs to GALU, and $\frac{1}{r}\overline{g_{i+1}}$ as its output. The quotient selection table is addressed with the leading bits of $\overline{g_{i+1}}$ and d, which we denote by $\overline{g_{i+1}}$ and $\overline{d}$. $\overline{g_{i+1}}$ is not a truncation of p_{i+1}, since it may differ by one unit in its last place from the corresponding bits of p_{i+1}.

The worst case delay around the selection loop is:

Read q_{j+1} from flip-flop QFF	9 ns
Select $q_{j+1}d$ through MUX	18 ns
Add or subtract in GALU	35 ns
Prediction logic	30 ns
Setup time for q_{i+2} flip-flop QFF	3 ns
Total	95 ns

The Algorithm

Before examining the guess ALU in more detail, we need to explain the division algorithm. The significands of the initial division operands lie in the range

$$0 \le \text{dividend } p_0 < 2 \qquad (2)$$

$$1 \le \text{divisor } d < 2 \qquad (3)$$

because the divisor must be normalized. The partial remainders p_i are two's complement, while the divisor d is always positive.

After step i,

$$-\frac{2}{3}d \le \frac{1}{r}p_{i+1} \le \frac{2}{3}d \qquad (4)$$

Consequently, at the beginning of the next step, after $\frac{1}{r}p_{i+1}$ has been shifted left to multiply by r,

$$-\frac{8}{3}d \le p_{i+1} \le \frac{8}{3}d \qquad (5)$$

p_{i+1} can be driven back into the interval of equation (4) by the appropriate subtraction or addition of zero, d or $2d$. The process is illustrated in Figure 3.

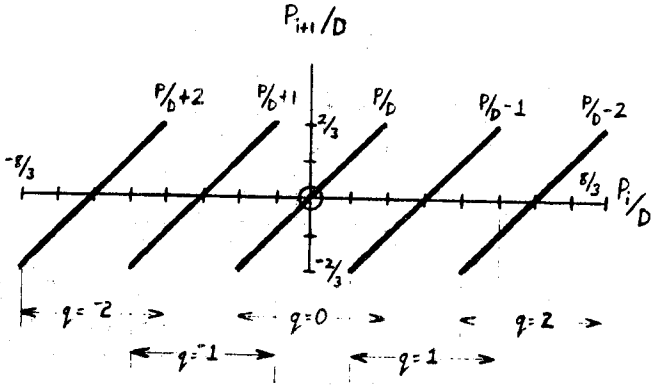

FIGURE 3 – DIVISION ALGORITHM

The setup step for the algorithm selects q_1, which is used in the first iteration to move the dividend from the range of equation (2) into the range of equation (4). Since the dividend is strictly less than $2d$, ultimately q_1 contributes one bit to quotient Q_m rather than two. If $q_1 = 2$, then p_1 will be negative and the adjustment to Q_1 during the second iteration of equation (1b) will be subtraction. Consequently, Q_m has an odd number of significant bits.

8-bit Next Remainder Prediction ALU

The guess ALU's width is chosen to satisfy the conflicting demands of high speed and simple quotient selection logic. Meeting an 8-bit boundary is desirable for design with 4-bit ALU slices. To determine the minimum reasonable width, we contruct a table for the quotient selection logic. Inspection shows that five remainder bits and three divisor bits (plus the first bit which is always one) are enough to determine q_{i+2} except in a few cases. Table 3 shows our quotient selection logic organized by these eight bits. In the exceptional cases, either one or two more bits of $\overline{g_{i+1}}$ must be observed.

The fifth and sixth columns of Table 3 contain the bounds on $\frac{p_{i+1}}{d}$ which can be set by observing the bits of $\overline{g_{i+1}}$ and $\bar{d}$ shown in columns two and four. q_{i+2} can be selected only if the minimum and the maximum ratios in a given row are within $\frac{2}{3}$ units of the same integer. The bounds depend on the relationship between g_{i+1} and p_{i+1}. The GALU's inputs are truncations of the main ALU's inputs.

$$|\overline{q_{i+1}d}| = |q_{i+1}d \text{ chopped}| = |q_{i+1}d| + (-1,0] \quad (6)$$

$$\overline{p_i} = p_i \text{ chopped} = p_i + (-1,0] \quad (7)$$

where the intervals are in units of the least significant bit of GALU. Depending on the sign of q_{i+1}, the GALU performs

$$\frac{g_{i+1}}{r} = \overline{p_i} \pm |\overline{q_{i+1}d}| \quad (8)$$

Case +:

$$\frac{g_{i+1}}{r} = \frac{p_{i+1}}{r} + (-2,0] \quad (9a)$$

Case −:

$$\frac{g_{i+1}}{r} = \frac{p_{i+1}}{r} + (-1,1) \text{ if GALU performs A-B, but} \quad (9b)$$

$$\frac{g_{i+1}}{r} = \frac{p_{i+1}}{r} + (-2,0) \text{ if GALU performs A-B-1.} \quad (9c)$$

Since a particular g_{i+1} may result from either addition or subtraction,

$$p_{i+1} = g_{i+1} + [0,2) \text{ ULPs of } g_{i+1} \quad (10)$$

for the asymmetric GALU which performs A+B or A-B-1. The quotient selection logic addressed by $\overline{g_{i+1}}$ and $\bar{d}$ can bound $\frac{p_{i+1}}{d}$ by:

for $g_{i+1} \geq 0$,

$$\frac{\overline{g_{i+1}}}{\bar{d} + 1 \text{ ULP of } \bar{d} - \varepsilon} \leq \frac{p_{i+1}}{d} \leq \frac{\overline{g_{i+1}} + 1 \text{ ULP of } \overline{g_{i+1}} + 1 \text{ ULP of } g_{i+1} - 2\varepsilon}{\bar{d}} \quad (11a)$$

for $g_{i+1} < 0$,

$$\frac{\overline{g_{i+1}} + 1 \text{ ULP of } \overline{g_{i+1}} + 1 \text{ ULP of } g_{i+1} - 2\varepsilon}{\bar{d} + 1 \text{ ULP of } \bar{d} - \varepsilon} \leq \frac{p_{i+1}}{d} \leq \frac{\overline{g_{i+1}}}{\bar{d}} \quad (11b)$$

where ε = 1 ULP of p_{i+1} and d.

The bounds can be evaluated once the widths of g_{i+1} and $\bar{d}$ are chosen. In our design, g_{i+1} has eight bits and $\bar{d}$ has four, so one ULP of $g_{i+1} = \frac{1}{16}$, one ULP of $\overline{g_{i+1}} = \frac{1}{8}$ and one ULP of $\bar{d} = \frac{1}{8}$. Figure 4 illustrates the calculation of $\overline{g_{i+1}}$.

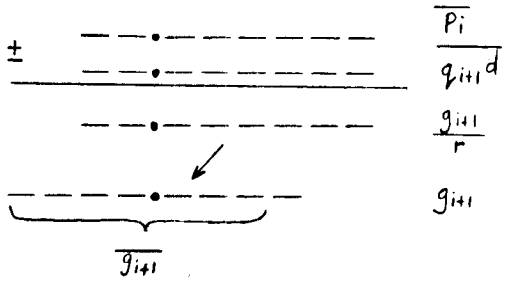

FIGURE 4 – $\overline{g_{i+1}}$

As an example, assume that $\overline{g_{i+1}}$ = binary 0001.1 and $\bar{d}$ = binary 1.000.

$$1.3333 < \frac{1.5}{1.0 + 0.125 - \varepsilon} \leq \frac{p_{i+1}}{d} \leq \frac{1.5 + 0.5 + 0.625 - 2\varepsilon}{1.0} < 2.0625$$

Asymmetric GALU

The advantage of equation (10) over its counterpart for a symmetric GALU is that g_{i+1} wiggles in only one direction. $|g_{i+1}|$ is never larger than $|p_{i+1}|$. Many $\frac{g_{i+1}}{\bar{d}}$ and $\frac{\overline{g_{i+1}}}{\bar{d} + 1 \text{ ULP of } \bar{d}}$ ratios are multiples of $\frac{1}{3}$. If a predicted minimum magnitude for $\frac{p_{i+1}}{d}$ equals $\pm \frac{1}{3}$ or $\pm \frac{4}{3}$, or a predicted maximum magnitude equals $\pm \frac{2}{3}$ or $\pm \frac{5}{3}$, then more than five bits of g_{i+1} must be observed. To avoid looking at more bits when the maximum ratio is $\frac{5}{3}$, for example, the predicted minimum ratio would have to be $\geq \frac{4}{3}$. But Table 3 shows that the difference

between the predicted bounds is never as small as $\frac{1}{3}$ unit. Since g_{i+1} is uncertain in only one direction rather than two, there is sufficient information without observing another bit in approximately half of the boundary cases.

Our quotient selection logic implements Table 3 using 39 minterms. An earlier design based on a 9-bit symmetric GALU would have required 56 minterms. An 8-bit symmetric ALU would have required even more minterms and at least one more bit in g_{i+1} or $\bar{d}$.

Although asymmetry decreases the size of both the ALU and the programmable logic, it might not simplify a RAM implementation. The width of g_{i+1} remains seven bits rather than six because of one bad case: see $(g_{i+1}, \bar{d}) = (1.10.0xx, 1.000)$ in Table 3. A single level RAM would require ten address bits. However, a two level RAM implementation, such as the one suggested by Tan [9], could trade one more bit of $\bar{d}$ for one less bit g_{i+1}.

Verification

The quotient selection table was tested by simulation with all pairs of 12-bit dividends and divisors. No error was found and no part of the unimplemented region in Table 3 was accessed. Random modifications to the table caused errors to be detected.

Division Step by Step

The division operation proceeds in four steps. Refer to Figure 1.

Step 1

Load the divisor into DR. Load the dividend into RR through MUX and DALU. The MUX shifts the dividend right by four bits and there is a wired left shift by two bits at RR. The net effect is to shift the dividend right by two bits. The dividend is loaded by a roundabout path in order to save the space and delay which a multiplexer in front of RR would cost.

Step 2

Put q_1 into QFF by adding zero to RR in the ALUs and reloading RR. This leaves the dividend in RR with its binary point in the same relative position as the divisor's binary point occupies in DR. GALU's output $\frac{g_0}{r}$ equals $\frac{p_0}{r}$. Consequently, the quotient selection logic chooses q_1 by comparing the dividend and divisor with their binary points correctly aligned.

Step 3

Repeat equations (1ab) 34 times. The sign bit of QFF controls the ALU operation. The other two bits control the MUX. At the end of each cycle, clock QFF into the POS and NEG registers, p_{i+1} into RR, and q_{i+2} into QFF.

Step 4

Subtract NEG from POS to form Q_m. If p_m (in RR) is negative, then subtract one more ULP from Q_m. The Sticky bit is zero if $p_m = 0$ and one otherwise.

For the purpose of division, DR is a register of the same length as the operands. RR is a register three bits longer than the operands. DALU is one bit wider than the operands. POS and NEG are shift registers four bits longer than the operands.

Remainder

KCS defines a remainder operation whose result has magnitude no greater than half the divisor's magnitude. To produce this result, a fixup step is required after division. It is convenient to change RR from a register to a shift register so that the last partial remainder can be shifted back to the right by two bits in order to align it with the divisor.

Square Root

The restoring square root algorithm produces one result bit per step. The accumulated partial result after any step is the truncation of the infinitely precise answer, so the bits may be collected in a shift register.

The algorithm consists of "completing the square." Two bits of the operand are brought into the calculation during each cycle. Imagine that before each cycle the remainder and partial result are aligned so that

$$(ar)^2 \le \text{operand} = (ar + b)^2 + c < (ar + r)^2 \quad (12)$$

where

ar = the truncated result already found

r = radix = 2

a, b are integers

c is a real number $\le r$

and we seek b in $0 \le b \le r-1$ to minimize $c \ge 0$. The current remainder $= (ar + b)^2 + c - (ar)^2 = 2arb + b^2 + c$. b is either 0 or 1. To find the next result bit, assume $b = 1$ and subtract $4a + 1$ from the current remainder. The next result bit is one if this difference is ≥ 0, and zero otherwise.

The position of the binary point within the operand imposes only one restriction. Pairs of operand bits brought into the calculation must lie on the same side of the binary point. Thus if the exponent's value is even and the significand's value is between 1 and 2, only one bit will be used during the first iteration. The significand is shifted left by one bit if the exponent is odd. This may raise the significand's value in the first iteration to between 2 and 4, so that two bits are used.

The hardware previously described for division and remainder is extended in three ways for square root. Shift register SQR holds the operand until it is introduced into the computation. RR becomes a two bit at a time left shift register. (The remainder fixup step already requires it to be a two bit at a time right shift register.) DR is changed from a register to a one bit at a time left shift register in order to hold the developing square root result.

As used in square root, RR and DALU are three bits wider than the operand. DR is one bit wider than the operand because the point at which result bits are inserted into DR is two bits left of the least significant end of RR and DALU. SQR is one bit narrower than the operand because the first two bits of it load directly into RR during the intitialization step.

A Note on Software Square Root

W. Kahan has shown that software square root algorithms can find the correctly rounded result using intermediate quantities no wider than the precision of the operand [13]. The calculation is simpler if the machine can chop quotients and round sums. Software methods can be expected to take between six and fifteen divide times, depending on the size of the processor. The

larger the processor, the greater the ratio. If hardware square root takes between one and two divide times, it will be about ten times faster than software. The choice of implementation depends on the importance of the square root operation and its incremental cost in the total hardware design.

Square Root Step by Step

The square root operation proceeds in five steps. Refer to Figure 2.

Step 1

Load the operand into DR. The operand should be normalized in order to avoid wasted cycles at the beginning of the iteration.

Step 2

Set QFF to one if the operand's unbiased exponent is even. Set QFF to two if the exponent is odd. In the latter case, the operand will be shifted left by one bit during the next step.

Step 3

Move the operand from DR through the MUX into RR and the square root register SQR. 65 bits (not including the sign which is known to be positive) come out of the MUX. The two high order bits, which are conceptually to the left of the binary point, go into the least significant bits of RR. Clear the remaining bits of RR. The 63 bits which are conceptually to the right of the binary point go into SQR. Clear DR to prepare for shifting in the result bits.

Step 4

Repeat 65 times: Subtract DR plus one from RR. If the difference is non-negative, then shift it left by two bits and store it in RR. Shift DR left by one bit and carry in a logic one.[3] If the difference is negative, then shift the old contents of RR left by two bits and shift DR left by one bit with a carry-in of zero. In either case, shift SQR left by two bits and fill in the rightmost two bits of RR with the bits shifted out of SQR.

Step 5

Move the 65-bit result from SQR to the normalization and rounding logic by clearing RR and adding in the DALU. The Sticky bit is the 66th bit of the result. It is formed by the logical OR of the DALU carry-out and the bits of RR during the last iteration of Step 4.[4] The sticky bit is latched at the end of Step 4 so that information is not lost when RR is cleared.

Conclusions

Radix four division offers us the most cost-effective improvement (in the same technology) to radix two restoring division. Radix four uses the same hardware structure in the partial remainder loop except for a multiplexer to produce a second multiple of the divisor. Since the ALU delay dominates the loop, radix four has the same step time as radix two. Quotient digit selection is the limiting task, so we reduce the width of the guess ALU to eight bits in order to speed that path. Our tradeoffs benefit a programmable logic implementation of the look-up table. Different choices could be better for a RAM implementation, especially a two level one. The cost of resolving the redundant quotient representation is low because registers and an ALU elsewhere in the accelerator can be shared for this purpose. Hardware square root is an inexpensive extension to our division scheme. The extra hardware is a shift register to hold the operand and a shift register to hold the result.

[3] The carry-out from DALU is tied directly to DR's left shift input.

[4] If the last iteration produces a one, then the square root has an infinite number of nonzero bits and the Sticky bit should be a one. The ALU's carry-out is a one in this case. If the last iteration produces a zero, then the Sticky bit is a one if the previous remainder was nonzero.

Acknowledgement

W. Kahan has offered encouragement and valuable suggestions throughout the course of this project.

References

[1] IEEE Computer Society Microprocessor Standards Committee Task P754, "A Proposed Standard for Binary Floating Point Arithmetic, Draft 8.0," *Computer* 14, No. 3, March, 1981, pp 52-63.

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

[3] G. Taylor and D. Patterson, "VAX Hardware for the Proposed IEEE Floating Point Standard," Fifth IEEE Symposium on Computer Arithmetic, May, 1981.

[4] D. Atkins, "The Theory and Implementation of SRT Division," Report 230, Dept. of Computer Science, University of Illinois, Urbana, June, 1967.

[5] D. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Transactions on Computers* 17, No. 10, October, 1968, pp. 925-934.

[6] D. Atkins, "A Study of Methods for Selection of Quotient Digits during Digital Division," Ph.D. dissertation, Report 397, Dept. of Computer Science, University of Illinois, Urbana, June, 1970.

[7] J. Robertson, "Methods of Selection of Quotient Digits during Digital Division," File 663, Dept. of Computer Science, University of Illinois, Urbana, June, 1965.

[8] D. Atkins, "Design of the Arithmetic Units of Illiac III: Use of Redundancy and Higher Radix Methods," Report 333, Dept. of Computer Science, University of Illinois, Urbana, May, 1969.

[9] K. Tan, "The Theory and Implementations of High-Radix Division," Fourth IEEE Symposium on Computer Arithmetic, October, 1978, pp. 154-163.

[10] U. Kalaycioglu, "Analysis and Synthesis of Generalized Radix Additive Normalization Division Techniques," Ph.D. dissertation, SEL Report 88, Dept. of Electrical and Computer Engineering, University of Michigan, Ann Arbor, May, 1975.

[11] J. Baron, "Implementation Study of Generalized Radix, Non-Restoring Division Techniques," SEL Report 102, Dept. of Electrical and Computer Engineering, University of Michigan, Ann Arbor, September, 1977.

[12] D. Atkins and U. Kalaycioglu, "Concurrency in Generalized Radix Non-Restoring Division", Twelfth Allerton Conference on Circuit and Switching Theory, University of Illinois, Urbana, October, 1974, pp. 628-640.

[13] W. Kahan, "Software Square Root for the Proposed IEEE Floating Point Standard," Computer Science Division, University of California, Berkeley, August, 1980, submitted to *IEEE Transactions on Mathematical Software*.

Table 3. -- Quotient Selection Logic:
Asymmetric 8-bit Next Remainder Prediction ALU

$\overline{g_{i+1}}$ first 5-7 bits of estimated next remainder (2's comp)		$\overline{d}$ first 4 bits of divisor (positive)		ratio of shifted full remainder p_{i+1} to full divisor d		q_{i+2} next quotient digit (sign mag)
decimal	binary	decimal	binary	minimum	maximum	
-3.5	1100.1	1.000	1.000	-2.6111	-3.5000	-2
-3.0	1101.0	1.000	1.000	-2.1667	-3.0000	-2
-2.5	1101.1	1.000	1.000	-1.7222	-2.5000	-2
-2.000	1110.000	1.000	1.000	-1.6111	-2.0000	-2
-1.875	1110.001	1.000	1.000	-1.5000	-1.8750	-2
-1.750	1110.010	1.000	1.000	-1.3889	-1.7500	-2
-1.625	1110.011	1.000	1.000	-1.2778	-1.6250	-1
-1.5	1110.1	1.000	1.000	-0.8333	-1.5000	-1
-1.0	1111.0	1.000	1.000	-0.3889	-1.0000	-1
-0.5	1111.1	1.000	1.000	0.0625	-0.5000	-0
0.0	0000.0	1.000	1.000	0.0000	0.5625	+0
0.5	0000.1	1.000	1.000	0.4444	1.0625	+1
1.0	0001.0	1.000	1.000	0.8889	1.5625	+1
1.5	0001.1	1.000	1.000	1.3333	2.0625	+2
2.0	0010.0	1.000	1.000	1.7778	2.5625	+2
2.5	0010.1	1.000	1.000	2.2222	3.0625	+2
-3.5	1100.1	1.125	1.001	-2.3500	-3.1111	-2
-3.0	1101.0	1.125	1.001	-1.9500	-2.6667	-2
-2.5	1101.1	1.125	1.001	-1.5500	-2.2222	-2
-2.00	1110.00	1.125	1.001	-1.3500	-1.7778	-2
-1.75	1110.01	1.125	1.001	-1.1500	-1.5556	-1
-1.5	1110.1	1.125	1.001	-0.7500	-1.3333	-1
-1.0	1111.0	1.125	1.001	-0.3500	-0.8889	-1
-0.5	1111.1	1.125	1.001	0.0556	-0.4444	-0
0.0	0000.0	1.125	1.001	0.0000	0.5000	+0
0.5	0000.1	1.125	1.001	0.4000	0.9444	+1
1.0	0001.0	1.125	1.001	0.8000	1.3889	+1
1.50	0001.10	1.125	1.001	1.2000	1.8111	+1
1.75	0001.11	1.125	1.001	1.4000	1.8333	+2
2.0	0010.0	1.125	1.001	1.8000	2.2778	+2
2.5	0010.1	1.125	1.001	2.0000	2.7222	+2
3.0	0011.0	1.125	1.001	2.4000	3.1667	+2
-4.0	1100.0	1.250	1.010	-2.5000	-3.2000	-2
-3.5	1100.1	1.250	1.010	-2.1364	-2.8000	-2
-3.0	1101.0	1.250	1.010	-1.7727	-2.4000	-2
-2.5	1101.1	1.250	1.010	-1.4091	-2.0000	-2
-2.0	1110.0	1.250	1.010	-1.0455	-1.6000	-1
-1.5	1110.1	1.250	1.010	-0.6818	-1.2000	-1
-1.00	1111.00	1.250	1.010	-0.5000	-0.8000	-1
-0.75	1111.01	1.250	1.010	-0.3182	-0.6000	-0
-0.5	1111.1	1.250	1.010	0.0500	-0.4000	-0
0.0	0000.0	1.250	1.010	0.0000	0.4500	+0
0.5	0000.1	1.250	1.010	0.3636	0.8500	+1
1.0	0001.0	1.250	1.010	0.7273	1.2500	+1
1.5	0001.1	1.250	1.010	1.0909	1.6500	+1
2.0	0010.0	1.250	1.010	1.4545	2.0500	+2
2.5	0010.1	1.250	1.010	1.8182	2.4500	+2
3.0	0011.0	1.250	1.010	2.1818	2.8500	+2
3.5	0011.1	1.250	1.010	2.5455	3.2500	+2
-4.5	1011.1	1.375	1.011	-2.6250	-3.2727	-2
-4.0	1100.0	1.375	1.011	-2.2917	-2.9091	-2
-3.5	1100.1	1.375	1.011	-1.9583	-2.5455	-2
-3.0	1101.0	1.375	1.011	-1.6250	-2.1818	-2
-2.50	1101.10	1.375	1.011	-1.4583	-1.8182	-2
-2.25	1101.11	1.375	1.011	-1.2917	-1.6364	-1
-2.0	1110.0	1.375	1.011	-0.9583	-1.4545	-1
-1.5	1110.1	1.375	1.011	-0.6250	-1.0909	-1
-1.00	1111.00	1.375	1.011	-0.4583	-0.7273	-1
-0.75	1111.01	1.375	1.011	-0.2917	-0.5455	-0
-0.5	1111.1	1.375	1.011	0.0455	-0.3636	-0
0.0	0000.0	1.375	1.011	0.0000	0.4091	+0
0.5	0000.1	1.375	1.011	0.3333	0.7727	+1
1.0	0001.0	1.375	1.011	0.6667	1.1364	+1
1.5	0001.1	1.375	1.011	1.0000	1.5000	+1
2.0	0010.0	1.375	1.011	1.3333	1.8636	+2
2.5	0010.1	1.375	1.011	1.6667	2.2273	+2
3.0	0011.0	1.375	1.011	2.0000	2.5909	+2
3.5	0011.1	1.375	1.011	2.3333	2.9545	+2
-4.5	1011.1	1.500	1.100	-2.4231	-3.0000	-2
-4.0	1100.0	1.500	1.100	-2.1154	-2.6667	-2
-3.5	1100.1	1.500	1.100	-1.8077	-2.3333	-2
-3.0	1101.0	1.500	1.100	-1.5000	-2.0000	-2
-2.5	1101:1	1.500	1.100	-1.1923	-1.6667	-1
-2.0	1110.0	1.500	1.100	-0.8846	-1.3333	-1

$\overline{g_{i+1}}$ first 5-7 bits of estimated next remainder (2's comp)		$\overline{d}$ first 4 bits of divisor (positive)		ratio of shifted full remainder p_{i+1} to full divisor d		q_{i+2} next quotient digit (sign mag)
decimal	binary	decimal	binary	minimum	maximum	
-1.5	1110.1	1.500	1.100	-0.5769	-1.0000	-1
-1.0	1111.0	1.500	1.100	-0.2692	-0.6667	-0
-0.5	1111.1	1.500	1.100	0.0417	-0.3333	-0
0.0	0000.0	1.500	1.100	0.0000	0.3750	+0
0.50	0000.10	1.500	1.100	0.3077	0.5417	+0
0.75	0000.11	1.500	1.100	0.4615	0.7083	+1
1.0	0001.0	1.500	1.100	0.6154	1.0417	+1
1.5	0001.1	1.500	1.100	0.9231	1.3750	+1
2.00	0010.00	1.500	1.100	1.2308	1.5417	+1
2.25	0010.01	1.500	1.100	1.3846	1.7083	+2
2.5	0010.1	1.500	1.100	1.5385	2.0417	+2
3.0	0011.0	1.500	1.100	1.8462	2.3750	+2
3.5	0011.1	1.500	1.100	2.1538	2.7083	+2
4.0	0100.0	1.500	1.100	2.4615	3.0417	+2
-5.0	1011.0	1.625	1.101	-2.5357	-3.0769	-2
-4.5	1011.1	1.625	1.101	-2.2500	-2.7692	-2
-4.0	1100.0	1.625	1.101	-1.9643	-2.4615	-2
-3.5	1100.1	1.625	1.101	-1.6786	-2.1538	-2
-3.0	1101.0	1.625	1.101	-1.3929	-1.8462	-2
-2.5	1101.1	1.625	1.101	-1.1071	-1.5385	-1
-2.0	1110.0	1.625	1.101	-0.8214	-1.2308	-1
-1.5	1110.1	1.625	1.101	-0.5357	-0.9231	-1
-1.0	1111.0	1.625	1.101	-0.2500	-0.6154	-0
-0.5	1111.1	1.625	1.101	0.0385	-0.3077	-0
0.0	0000.0	1.625	1.101	0.0000	0.3462	+0
0.5	0000.1	1.625	1.101	0.2857	0.6538	+0
1.0	0001.0	1.625	1.101	0.5714	0.9615	+1
1.5	0001.1	1.625	1.101	0.8571	1.2692	+1
2.0	0010.0	1.625	1.101	1.1429	1.5769	+1
2.5	0010.1	1.625	1.101	1.4286	1.8846	+2
3.0	0011.0	1.625	1.101	1.7143	2.1923	+2
3.5	0011.1	1.625	1.101	2.0000	2.5000	+2
4.0	0100.0	1.625	1.101	2.2857	2.8077	+2
4.5	0100.1	1.625	1.101	2.5714	3.1154	+2
-5.5	1010.1	1.750	1.110	-2.6333	-3.1429	-2
-5.0	1011.0	1.750	1.110	-2.3667	-2.8571	-2
-4.5	1011.1	1.750	1.110	-2.1000	-2.5714	-2
-4.0	1100.0	1.750	1.110	-1.8333	-2.2857	-2
-3.5	1100.1	1.750	1.110	-1.5667	-2.0000	-2
-3.00	1101.00	1.750	1.110	-1.4333	-1.7143	-2
-2.75	1101.01	1.750	1.110	-1.3000	-1.5714	-1
-2.5	1101.1	1.750	1.110	-1.0333	-1.4286	-1
-2.0	1110.0	1.750	1.110	-0.7667	-1.1429	-1
-1.5	1110.1	1.750	1.110	-0.5000	-0.8571	-1
-1.0	1111.0	1.750	1.110	-0.2333	-0.5714	-0
-0.5	1111.1	1.750	1.110	0.0357	-0.2857	-0
0.0	0000.0	1.750	1.110	0.0000	0.3214	+0
0.5	0000.1	1.750	1.110	0.2667	0.6071	+0
1.0	0001.0	1.750	1.110	0.5333	0.8929	+1
1.5	0001.1	1.750	1.110	0.8000	1.1786	+1
2.0	0010.0	1.750	1.110	1.0667	1.4643	+1
2.5	0010.1	1.750	1.110	1.3333	1.7500	+2
3.0	0011.0	1.750	1.110	1.6000	2.0357	+2
3.5	0011.1	1.750	1.110	1.8667	2.3214	+2
4.0	0100.0	1.750	1.110	2.1333	2.6071	+2
4.5	0100.1	1.750	1.110	2.4000	2.8929	+2
-5.5	1010.1	1.875	1.111	-2.4688	-2.9333	-2
-5.0	1011.0	1.875	1.111	-2.2188	-2.6667	-2
-4.5	1011.1	1.875	1.111	-1.9688	-2.4000	-2
-4.0	1100.0	1.875	1.111	-1.7188	-2.1333	-2
-3.5	1100.1	1.875	1.111	-1.4688	-1.8667	-2
-3.0	1101.0	1.875	1.111	-1.2188	-1.6000	-1
-2.5	1101.1	1.875	1.111	-0.9688	-1.3333	-1
-2.0	1110.0	1.875	1.111	-0.7188	-1.0667	-1
-1.5	1110.1	1.875	1.111	-0.4688	-0.8000	-1
-1.0	1111.0	1.875	1.111	-0.2188	-0.5333	-0
-0.5	1111.1	1.875	1.111	0.0333	-0.2667	-0
0.0	0000.0	1.875	1.111	0.0000	0.3000	+0
0.5	0000.1	1.875	1.111	0.2500	0.5667	+0
1.0	0001.0	1.875	1.111	0.5000	0.8333	+1
1.5	0001.1	1.875	1.111	0.7500	1.1000	+1
2.0	0010.0	1.875	1.111	1.0000	1.3667	+1
2.5	0010.1	1.875	1.111	1.2500	1.6333	+1
3.0	0011.0	1.875	1.111	1.5000	1.9000	+2
3.5	0011.1	1.875	1.111	1.7500	2.1667	+2
4.0	0100.0	1.875	1.111	2.0000	2.4333	+2
4.5	0100.1	1.875	1.111	2.2500	2.7000	+2
5.0	0101.0	1.875	1.111	2.5000	2.9667	+2