A Radix-8 CMOS S/390 Multiplier

Eric M. Schwarz
IBM S/390
Mailstop: P310
522 South Rd.
Poughkeepsie, NY 12601
schwarz@vnet.ibm.com

Robert M. Averill III
IBM S/390
Mailstop: P310
522 South Rd.
Poughkeepsie, NY 12601

Leon J. Sigal
IBM Research
T.J. Watson Research Center
Route 134
Yorktown Heights, NY 10598

Abstract

The multiplier of a S/390 CMOS microprocessor is described. It is implemented in an aggressive static CMOS technology with 0.20 μm effective channel length. The multiplier has been demonstrated in a single-image shared-memory multiprocessor at frequencies up to 400 MHz. The multiplier requires three machine cycles for a total latency of 7.5 ns. Though, the design can support a latency of 4.0 ns if the latches are removed. The design goal was to implement a versatile S/390 multiplier with reasonable performance at a very aggressive cycle time. The multiplier implements a radix-8 Booth algorithm and is capable of supporting S/390 floating-point and fixed-point multiplications and also division and square root. Logic design and physical design issues are discussed relating to the Booth decode and counter tree implementations.

1. Introduction

This paper describes the uncommon usage of a high radix algorithm for multiplication in a high-performance microprocessor. The microprocessor is optimized for commercial workloads. The S/390 microprocessor has dimensions of 17.35x17.30mm^2 and has two Floating Point Units (FPU) utilizing 16 percent of the chip area. The dual FPUs operate on the same instruction stream for fault tolerant purposes. The counter tree requires about 15 percent of the FPU area and is 3470 μm by 1475 μm.

In commercial workloads the key issues in FPU implementations are different from scientific workloads. FPU operations are less frequent and more spread out in commercial workloads, and the combination of multiply and addition is less frequent. Performance is important, but not at the cost degradating the performance of other central processor functions. Thus, cycle time and area are critical followed by latency and throughput. Multiplication is the second most frequent arithmetic operation following addition. So, a large investment of area is usually spent on the multiplier which is best utilized for many operations to save overall area.

S/390 architecture has its own unique requirements not commonly found in other architectures. S/390 architecture defines floating point numbers to be in a hexadecimal format with a 1 bit sign, a 7 bit characteristic, and a fraction which is 24 bits for short format, 56 bits for long format and 112 bits for extended format. The following equation describes an operand, X, in this format:

$$X = (-1)^{S_x} \quad * \quad 16^{(C_x - 64)} \quad * \quad F_x$$

where S_x is the sign bit, C_x is the characteristic, and F_x is the fraction which is less than 1.0. The sign of the product and the characteristic are easy to implement and are not described in this paper. The fraction dataflow of the multiplier is described and it is optimized for long format.

S/390 fixed point format consists of two's complement integer notation and has 16 bit and 32 bit formats and in rare cases 64 bit format. Multiplication instructions are defined for operations on 16 bit by 32 bit and 32 bit by 32 bit. Division is defined for a 64 bit dividend and a 32 bit divisor, quotient, and remainder.

To take advantage of the high-speed multiplier, division and square root instructions are executed in the FPU using a Goldschmidt algorithm [6, 1, 12, 13] which has two multiplications in its iteration step. Intermediate calculations need to be computed in a higher precision to avoid truncation errors. Thus, a multiplier is desired which can support more than 56 bit operands. However, this support is not as critical as meeting a cycle time objective for long multiplication. The compromise is to only extend the multiplicand operand with additional precision, so as not to affect the multiplier operand, which has a greater effect on the cycle time. Thus, a radix-8 Booth algorithm which supports operands of 56 bits and 64 bits and supports both fixed point and floating point formats was chosen.

1.1. Design Issues

Three bit (radix-4) overlapped scanning Booth algorithms [3, 14] are commonly implemented in industry due to their simplicity and ease of creating multiples of the multiplicand. Radix-4 implementations using 4:2 counters are becoming very popular[18, 11, 7]. Though, for a S/390 architecture this may not be the optimal choice. IEEE 754 standard only has 53 bit operands for double format which partitions nicely with a radix 4 algorithm to require a 27 to 2 counter tree. This can be implemented in 4 levels of 4:2 counters or 7 levels of 3:2 counters. But a 56 bit radix-4 multiplier has 29 partial products which require 4 levels of 4:2 counters or 8 levels of 3:2 counters. The worst timing path in a 3:2 counter is a 3 way exclusive-OR to produce the sum output. The propagation delay of 8 levels of 3:2 counters and Booth multiplexing did not meet our cycle time objective. However, there have been S/390 implementations with a more relaxed cycle time which have been fabricated with this type of implementation[5].

The counter tree is a difficult function to spread over multiple cycles. At the bottom of the counter tree there are 2 operands of 120 bits for a total of 240 bits which are latched. This increases to approximately 360 bits and 480 bits if the latch point is moved one or two levels up into the tree. Therefore, partitioning the counter tree into separate cycles with latch boundaries is not a desirable implementation. Another option is to remove the latch boundaries and create a two cycle path in the counter tree. But this creates problems for AC test pattern analysis and timing analysis for latches with multiple cycle times. Other common methods are to use an iterative method[9], but our performance objective was to pipeline a multiplication every cycle, so this idea was also rejected.

A 4 bit (radix-8) overlapped scanning algorithm [14, 15, 10] is an attractive option for S/390 format since the counter tree is smaller in area and has less stages. It only requires a 19 to 2 counter tree which can be implemented in 6 levels of 3:2 counters or 4 levels of 4:2 counters. The disadvantage of 4 bit scanning is that it requires a 3X multiple. There have been some unusual designs for reducing the 3X delay but at the cost of adding delay to the counter tree[2]. In our implementation, the key concern was reducing cycle time. This translated into reducing the largest component of overall delay which is the counter tree. Thus, having the the counter tree in a separate cycle from the 3X calculation was allowable and desirable.

A high level comparison of 4 levels of 4:2 counters to 6 levels of 3:2 counters would yield that they are equivalent. Early designs of 4:2 counters required 2 levels of 3:2 counters, but recently 4:2 counters with pass transistors have been shown to require 1.5 levels of 3:2 counters. Thus, they are equivalent from this rough comparison. But it will be shown that 3:2 counters actually have an advantage in that their inputs vary in delay. A design will be shown which optimizes the delay for one of the three inputs. The inputs to the 3:2 counters can be ordered to have late arriving signals use the fast inputs. The delay is optimized to require less than 6 levels of the worst case propagation delay through the counter tree. Thus, using 3:2 counters results in a faster implementation than an equivalent 4:2 counter tree due to the advantages of varying input delays.

Our implementation requires 3 execution cycles for most multiplications as shown in Figure 1. The first cycle involves creating the 3X multiple and performing the Booth decoding. The second cycle involves multiplexing the multiples to create partial products and then reducing 19 partial products to 2 via the counter tree. The third cycle involves an 120 bit addition of the two partial results to produce the product. Also, there is a selection of two possible normalization results in the third cycle. The selection signal is built into the custom designed dataflow for speed. This selection is sufficient for over 90 percent of the cases since unnormalized input fractions are somewhat rare for multiplication. For unnormalized input fractions and for exponents that could potentially overflow or underflow, a fourth execution cycle is required. This cycle contains a full post normalizer with overflow and underflow detection.

This paper describes the Booth decoding, fixed point multiplication adaptation, counter tree design, and the custom circuit implementation. The multiplier has been fabricated and has been clocked at over 400 MHz. The multiplier has been partitioned into 3 execution cycles. The multiplier has a latency without latch delays of 4.0 ns. With further optimizations in the 3X adder and an aggressive process this multiplier could run below 4 ns.

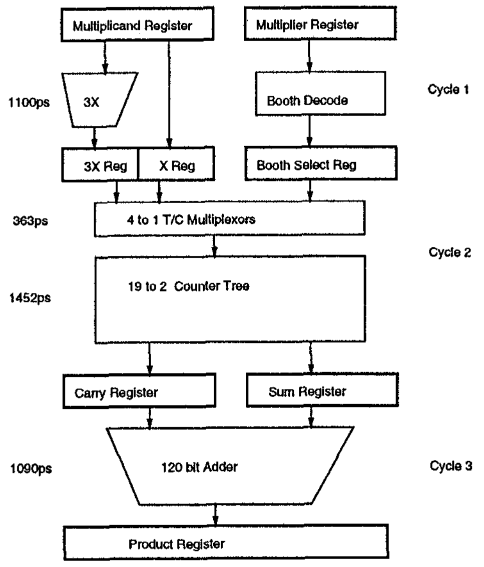

Figure 1. Dataflow of Overall Multiplier

2. Booth Decode

A 4-bit (radix-8) overlapped scanning algorithm [3, 14, 15, 10] is implemented which scans 3 bits plus 1 overlap bit per scan. The 56 bit multiplier operand has 19 scans which recode the multiplier into redundant octal digits. The digits can equal -4, -3, -2, -1, 0, +1, +2, +3, or +4. The scans of the multiplier bits are shown in Figure 2. To perform this recoding an additional bit needs to be concatenated to the left of the most significant bit and this bit is called the sign bit and is denoted by "S". S is equal to zero for a floating point multiplication since only magnitudes are considered and sign calculation is performed elsewhere. For fixed point multiplication, S in previous work [15] was equal to the sign bit. But our implementation eliminates the need for extending sign of the multiplier by altering the Booth decode of a few scans as will be detailed in section 3. In Figure 2 the multiplier is also extended past the least significant bit to include the bit "E". This is to complete the scan and complete the string recoding. E is chosen equal to zero so as not to change the value of the multiplier and just to complete the least significant scan.

For implementation purposes, the scans are numbered from least significant, 1, to most significant, 19. Note the fixed point numbers are placed in the least significant bits of the multiplier register. The upper scans do not have any significant bits and are either all zeros or all ones. A scan of all zeros or all ones translates into the recoding value of zero. The upper scans (12 to 19) do not have to consider fixed point encodings but the lower scans (1 to 11) do have to consider that the multiplicand could be a negative number.

```
bits                     |---- 32 bit ---|----16 bit----|
           11111111112222222222333333333344444444445555555
S01234567890123456789012345678901234567890123456789012345E
+|||  +|||  +|||  +|||  +|||  +|||  +|||  +|||  +|||  +|||
 19    17    15    13    11*    9     7     5     3     1
   +|||  +|||  +|||  +|||  +|||  +|||  +|||  +|||  +|||
    18    16    14    12    10     8    6*     4     2
                                              scans
Notation:
+ - overlap bit  S - sign bit (= 0 )  * - special fixed
| - scan bit     E - extra bit (zero)    point decodes
```

Figure 2. Booth Scanning of the Multiplier

The implementation of the Booth multiplexing consists of 19 - 4:1 true / complement multiplexors that gate the correct multiple for each partial product. There are 4 selection lines for the multiples of 1x, 2x, 3x, and 4X which are signaled by sx, s2x, s3x, and s4x. There is an inversion selection for selecting the one's complement of the multiple and for the hot one encoding, signaled by sinv. And, there is a selection line for the sign encoding bits for each partial product called s_sign. The following equations are used to implement the selection lines for the possible multiples of partial product. These equations are expressed for the general multiplier bits Y_{i-1}, Y_i, Y_{i+1}, and Y_{i+2} from most significant to least significant.

$$\begin{aligned} sx &= (\overline{(Y_{i-1} \oplus Y_i)}(Y_{i+1} \oplus Y_{i+2})) \\ s2x &= (\overline{Y_i} Y_{i+1} Y_{i+2}) + (Y_i \overline{Y_{i+1}}\ \overline{Y_{i+2}}) \\ s3x &= (Y_{i-1} \oplus Y_i)\,(Y_{i+1} \oplus Y_{i+2}) \\ s4x &= (\overline{Y_{i-1}} Y_i Y_{i+1} Y_{i+2}) + (Y_{i-1} \overline{Y_i}\ \overline{Y_{i+1}}\ \overline{Y_{i+2}}) \end{aligned}$$

Note that 1X and 2X multiples need to have sign extension bits filled into their most significant bits. This is accomplished using the sign bit of the multiplicand denoted by X_SIGN.

Additionally, the encoding bits of the partial product array are determined. A right encode of three bits is used to encode a "hot" one to produce a two's complement of the partial product if the multiple is negative. And, a left encode of three bits is used to encode the sign extension if the partial product is negative. There are two different cases for fixed point multiply and only one case for floating point multiply. For fixed point multiply having a multiple which is negative does not imply that the partial product is negative since this is also dependent on the sign of the multiplicand, X.

The inversion and sign extension can be implemented in two different ways: 1) assuming two representations for 0X (+0, -0), or 2) assuming one representation for 0X (+0). Positive zero is represented by a series of zeros whereas negative zero is represented by all ones plus a hot one and forcing the left sign encode bit to be 0 (which catches the carry out). The first option of having two zero representations simplifies the inversion control since it becomes simply equal to the most significant bit of the scan. The following are the equations of the two possible formulations where K is the scan number:

$$
\begin{array}{rcl}
\multicolumn{3}{l}{OPTION\ 1} \\
FOR\ K & = & 19 \\
sinv & = & 0\ \ must\ be\ positive \\
FOR\ 1 \leq & K & \leq 18 \\
sinv & = & Y_{i-1} \\
 & & \\
FOR\ 1 \leq & K & \leq 11 \\
s\_sign & = & Y_{i-1} \oplus (\overline{X\_SIGN} + \\
 & & (Y_{i-1} Y_i Y_{i+1} Y_{i+2}) + (\overline{Y_{i-1}}\ \overline{Y_i}\ \overline{Y_{i+1}}\ \overline{Y_{i+2}}\ )) \\
FOR\ 12 \leq & K & \leq 18 \\
s\_sign & = & \overline{Y_{i-1}}
\end{array}
$$

These equations are detailed in [15]. An alternative is option 2 which has only one representation for zero. The second option is actually implemented in the multiplier.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{OPTION\ 2} \\
FOR\ K & = & 19 \\
sinv & = & 0\ \ must\ be\ positive \\
FOR\ 1 \leq & K & \leq 18 \\
sinv & = & Y_{i-1}\overline{(Y_i Y_{i+1} Y_{i+2})} \\
 & & \\
FOR\ 1 \leq & K & \leq 11 \\
s\_sign & = & (\overline{X\_SIGN} \oplus Y_{i-1}) + (Y_{i-1} Y_i Y_{i+1} Y_{i+2}) \\
 & & +(\overline{Y_{i-1}}\ \overline{Y_i}\ \overline{Y_{i+1}}\ \overline{Y_{i+2}}\ ) \\
 & & \\
FOR\ 12 \leq & K & \leq 18 \\
s\_sign & = & \overline{Y_{i-1}\overline{(Y_i Y_{i+1} Y_{i+2})}}
\end{array}
$$

These selections are summarized in Table 1. Note sinv is used to conditionally invert the K-th partial product, if equal to one it is inverted, if equal to zero it is the true signal. sinv is also used by the K+1 partial product as the hot one encode placed to the right of the partial product. The right encode is equal to $(0\ \ ||\ \ 0\ \ ||\ \ sinv(K))$ for the K+1 partial product. The K + 1 row contains the hot one for the K-th row. s_sign is used to create the left sign encoding for the 18th through 1st partial product. For partial products 18 through 2 the encode is 111 for a positive partial product or 110 for a negative partial product which corresponds to $(1\ \ ||\ \ 1\ \ ||\ \ s\_sign(K))$ for the K-th partial product. For the first partial product the encode is 1000 for a positive partial product and 0111 for a negative partial product which corresponds to $(s\_sign(1)\ \ ||\ \ \overline{s\_sign(1)}\ \ ||\ \ \overline{s\_sign(1)}\ \ ||\ \ \overline{s\_sign(1)})$.

In the implementation of the multiplier the fan-out of the Booth select signals is rather large and the wires are rather long. To solve this, multiple copies of the registers are created to drive only half the width of each partial product. The A copy drives bits 34 to 66 of the multiplexors and the B copy drives bits 1 to 33. In addition, multiple copies of the X and 3X register are created to reduce the fanout to 19 Booth multiplexors for each partial product. The A copy drives partial products 11 to 19 and the B copy drives partial products 1 to 10. These registers are placed close to the counter tree to reduce the wiring length.

3. Fixed Point Multiplication

Fixed point multiplication of 16 by 32 bit and 32 by 32 bit is supported in the multiplier. The multiplier operand can be 16 or 32 bits and the multiplicand is always 32 bits. Fixed point data is right aligned to the least significant bits of the multiplier. Sign-extending the fixed point operands to 56 bits creates large fanouts which can result in a long delay. A simpler solution is used which is equivalent without the large fanouts.

If the multiplier operand is sign extended to 56 bits then certain scans will have all zeros or all ones. For 32 bit format, scans 12 to 19, and for 16 bit format, scans 7 to 19 are affected. These scans are recoded to the value zero for either case. If instead the input operands were not sign extended but instead were zero extended on the most significant side, then these scans would also have the recoded value of zero. So, it is not necessary to sign extend the multiplier operand for these scans, zero extending them is sufficient.

For the scans that have a partial sign extension an equivalent operation can be done. Scan 6 for 16 bit format and scan 11 for 32 bit format can have significant bits of the fixed point numbers and sign bits. Scan 11 has bits 23, 24, 25, and 26 where bit 24 can be the most significant bit / sign bit of a 32 bit fix point operand. In this case bit 24 is sign extended to bit 23 and the resulting signal is used rather than normal fraction bits.

						Option 1: +0/-0						Option 2: +0					
						1 to 11			12 to 19			1 to 11			12 to 19		
X_Sign	Y_{i-1}	Y_i	Y_{i+1}	Y_{i+2}	Select	sinv		s_sign	sinv		s_sign	sinv		s_sign	sinv		s_sign
0	0	0	0	0	0X	0	+0	1	0	+0	1	0	+0	1	0	+0	1
0	0	0	0	1	+1X	0		1	0		1	0		1	0		1
0	0	0	1	0	+1X	0		1	0		1	0		1	0		1
0	0	0	1	1	+2X	0		1	0		1	0		1	0		1
0	0	1	0	0	+2X	0		1	0		1	0		1	0		1
0	0	1	0	1	+3X	0		1	0		1	0		1	0		1
0	0	1	1	0	+3X	0		1	0		1	0		1	0		1
0	0	1	1	1	+4X	0		1	0		1	0		1	0		1
0	1	0	0	0	-4X	1		0	1		0	1		0	1		0
0	1	0	0	1	-3X	1		0	1		0	1		0	1		0
0	1	0	1	0	-3X	1		0	1		0	1		0	1		0
0	1	0	1	1	-2X	1		0	1		0	1		0	1		0
0	1	1	0	0	-2X	1		0	1		0	1		0	1		0
0	1	1	0	1	-1X	1		0	1		0	1		0	1		0
0	1	1	1	0	-1X	1		0	1		0	1		0	1		0
0	1	1	1	1	0X	1	-0	0	1	-0	0	0	+0	1	0	+0	1
1	0	0	0	0	0X	0	+0	1	0	+0	1	0	+0	1	0	+0	1
1	0	0	0	1	+1X	0		0	0		1	0		0	0		1
1	0	0	1	0	+1X	0		0	0		1	0		0	0		1
1	0	0	1	1	+2X	0		0	0		1	0		0	0		1
1	0	1	0	0	+2X	0		0	0		1	0		0	0		1
1	0	1	0	1	+3X	0		0	0		1	0		0	0		1
1	0	1	1	0	+3X	0		0	0		1	0		0	0		1
1	0	1	1	1	+4X	0		0	0		1	0		0	0		1
1	1	0	0	0	-4X	1		1	1		0	1		1	1		0
1	1	0	0	1	-3X	1		1	1		0	1		1	1		0
1	1	0	1	0	-3X	1		1	1		0	1		1	1		0
1	1	0	1	1	-2X	1		1	1		0	1		1	1		0
1	1	1	0	0	-2X	1		1	1		0	1		1	1		0
1	1	1	0	1	-1X	1		1	1		0	1		1	1		0
1	1	1	1	0	-1X	1		1	1		0	1		1	1		0
1	1	1	1	1	0X	1	-0	0	1	-0	0	0	+0	1	0	+0	1

Table 1. Booth Scanning: Determination of Select Signals

For scan 6, bit 40 is the sign bit for a 16 bit fixed point operand and it needs to be sign extended to bits 38 and 39 of the scan. The Booth select signals are created using the resulting signal rather than normal fraction bits.

The sign extension for fixed point only affects 3 bits that drive to two scans. The equations of these signals are as follows:

$$\begin{aligned} Y\_SCAN11(23) &= (\overline{FIXPT\_32} \text{ and } Y(23)) \text{ or} \\ &\quad (FIXPT\_32 \text{ and } Y(24)) \\ Y\_SCAN6(38) &= (\overline{FIXPT\_16} \text{ and } Y(38)) \text{ or} \\ &\quad (FIXPT\_16 \text{ and } Y(40)) \\ Y\_SCAN6(39) &= (\overline{FIXPT\_16} \text{ and } Y(39)) \text{ or} \\ &\quad (FIXPT\_16 \text{ and } Y(40)) \end{aligned}$$

where the signal FIXPT_32 is active high for fixed point 32 bit format, FIXPT_16 is active high for fixed point 16 bit format, Y_SCAN11(23) is the signal driven to the Booth decoder for scan 11 instead of Y(23), and Y_SCAN6(38:39) is driven to the decoder for scan 6 instead of Y(38:39). Note that bit 23 and bit 38 driven to scan 12 and scan 7 respectively are not the new sign extension signals. Instead they must be the original multiplier bits so that these scans will still have all zero or all one bits. This implementation reduces the sign extension of the fixed point multiplier operand to be only 2 bits for 16 bit format and 1 bit for 32 bit format, rather than 41 bits and 25 bits respectively.

4. Counter Tree Design

The Booth decode creates the select lines for the multiplexing of the possible multiples. This is implemented with a 4:1 True/Complement multiplexor. In addition to creating the partial products, the sign encode and hot one encode for each partial product are formed from the sign of the coefficient of the partial product and the sign of the partial product. The partial product array produced is shown in Figure 3. There are 19 partial products. Each partial product has 66 bits of magnitude, all but the top and bottom row have 3 bits of left (sign) encode, and all but the bottom have 3 bits of right (hot one) encode. Thus, the average partial product has 72 bits.

The 19 partial products are summed to 2 partial products. They are summed using carry-save adders (CSAs), sometimes referred to as full-adders or as 3 to 2 counters. The CSAs perform carry-free addition and reduce three inputs to two outputs. The counter tree has 6 levels and outputs a final carry and sum of 120 bits.

Note that any carries out of the most significant bit are discarded. A 56 by 64 bit multiply is guaranteed to only require 120 bits to represent the full precision product. Also, the sign extension encoding can produce a carry out of the partial product array which should be ignored.

A Dadda type[4] implementation is employed where not all the columns participate in a given level of the re-

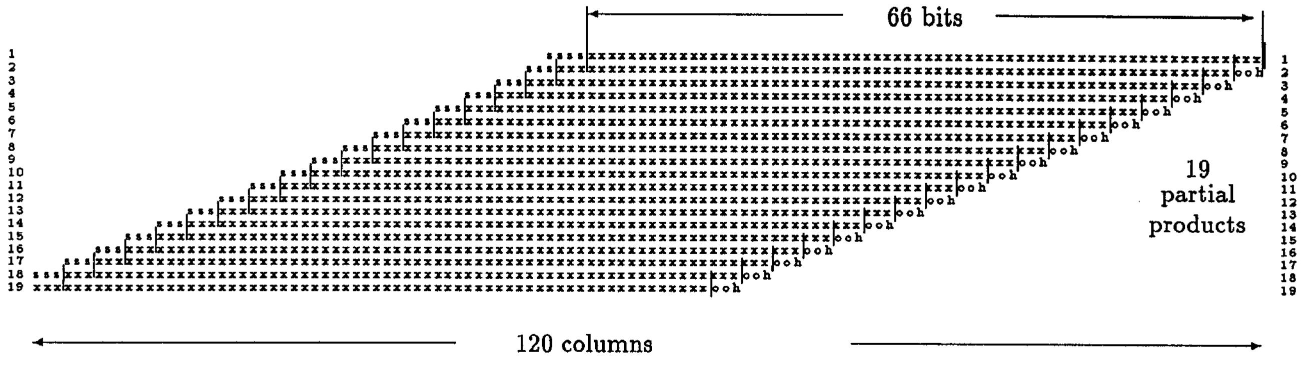

Figure 3. Partial Product Array for 4 Bit Scanning

duction as opposed to a Wallace scheme[16]. Columns 1 to 17 and 105 to 120 (with the most significant column labeled 1) are optimized for counter usage while columns 18 to 104 require all 6 levels of counters.

4.1. Connection and Placement of Counter Trees

Care was taken in connecting the counters to optimize for timing. Approximately 282 ps is needed to compute the sum output from the A and B inputs and 202 ps is needed to compute the carry output (Cout) from the A and B inputs at a 400 MHz clock frequency. There is less delay in the carry input (Cin) signal through the counter. The overall connection of counters is shown in Figure 4.

Counters labeled A through F make up the first level, G through J the second level, K through M the third level, N and O the fourth level, P the fifth level, and Q the sixth level. The counters are arranged in the second level to have counter H and I to have fast inputs and G and J to have slow inputs. Then to save overall delay in the third level the G and J sum outputs are input to carry input of the next level counters. In this way the delay of the overall tree is less than the worse delay per stage times the number of stages (1692 ps) and instead is 1452 ps. This is an average of 242 ps per counter level.

The placement of the counters was also carefully chosen as shown in Figure 5. The sum output bits of the counters and the multiplexor output bits are shown on the left with black arrows. Their outputs stay in the same track since they do not change weights. The carry output bits from the next lesser significant track are shown on the right of the figure with white arrows.

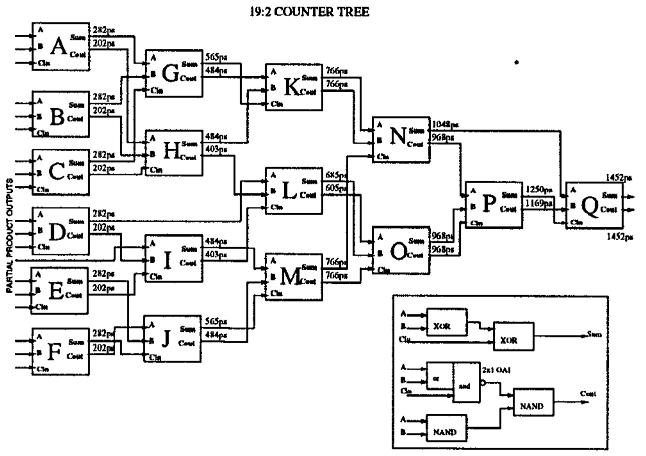

Figure 4. Timing and Connection of Counter Tree

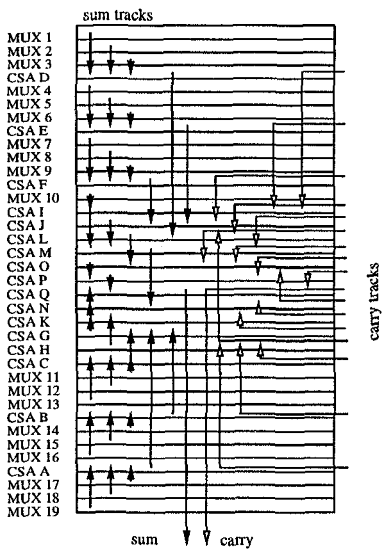

Figure 5. Placement of Counters

The carries propagate one track to the left since they are weighted more than the sum bit from the same counter. The goal in placing the counters and multiplexors was to reduce the number of wiring tracks used and the length of the wire. The multiplexors which feed into a counter are placed next to it. And counters which feed another counter are placed close by. The counters feed inwards. The last counter in the tree, Q, is located in the middle of the tree and its outputs are the overall carry and sum of the counter tree.

The physical design of the counter tree is shown in Figure 6. Note the traditional rhomboid shape of the counter tree layout. Other researchers have shown methods for creating rectangular layouts [2] to reduce area but this creates wiring problems. In our implementation, the fraction dataflow has an 120 bit wide layout, so the width of the rhomboid did not present a problem. The dataflow going into the multiplier is 66 bits wide which takes up approximately half the width of the fraction dataflow and placed along side it is the aligner for floating point addition. The 120 bit adder and the post normalizer require the full fraction dataflow width. So, the counter tree did not present a problem in terms of layout width. Though, it did present a challenge in driving select signals due to wire length and fanout. As mentioned earlier, two copies of the registers which drive the Booth multiplexors were created and placed on each side of the counter tree. The leftover area from placing the rhomboid counter tree in a rectangular layout area is used for these registers. Note that in Figure 6, the rhomboid is 3470 μm by 1475 μm and the total area shown is 3875 μm by 1800 μm. Thus, the select registers are very close to the counter tree and take up otherwise unusable space due to its triangular shape.

5. Overall Delay

There are 3 cycles of execution for the multiplier. At the demonstrated cycle time of 400 MHz (2.5 ns) [17] this is a total of 7.5 ns of latency. Several recent studies have compared multipliers without including latch delays with the best being between 4.1 to 4.4 ns for a 54 by 54 bit multiplier [11, 7, 8]. To fairly compare our multiplier, the delay needs to be determined without latches even though it is not implemented in this fashion. In the first cycle there is a 3X computation which takes 1100 ps. It is not optimized for latency but is a conservative implementation to reduce area with the requirement of meeting cycle time. Also the Booth decode is in the first cycle but it is much faster than the 3X computation. In the second cycle the 4:1 true / complement multiplexor has a delay of 363 ps and the counter tree requires 1452 ps for a total of 1815 ps. In the third cycle there is an 120 bit adder which has a delay of 1090 ns. The 120 bit adder is a conditional sum adder with 8 bit groups using carry lookahead to determine the carry, and ripple add to determine the conditioned sums. So the total of 1100, 1815, and 1090 ps is 4.005 ns for the 56 by 64 bit multiplication. This is faster than these previous multiplier studies. This design could be further optimized for latency with a better 3X adder delay or designing the 3X computation into the partial product array [2].

6. Conclusion

Our design goal was to implement a multiplier supporting a fast cycle time with a latency of approximately 3 execution cycles. And our goal was not to build the fastest multiplier but build a system with a fast multiplier supporting multiple purposes. S/390 fixed point and floating point multiplication and division and square root are implemented using this multiplier. This is a versatile multiplier for S/390 instructions and it is reasonably fast. A S/390 microprocessor employing a 3 cycle execution radix-8 multiplier has been fabricated and operated in a system at speeds up to 400 MHz.

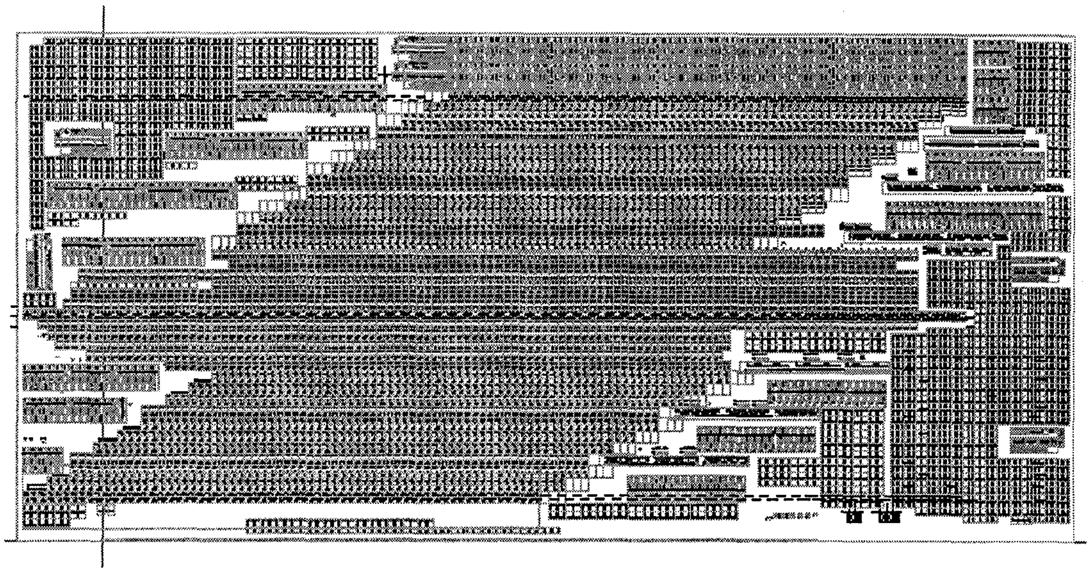

Figure 6. Physical Design of Counter Tree

7. Acknowledgment

The design of a microprocessor's multiplier involves many people and we would especially like to thank the efforts of Barry Winter and Dale Hoffman.

References

[1] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. "The IBM system/360 model 91: floating-point execution unit," *IBM Journal of Research and Development*, 11(1):34–53, Jan. 1967.

[2] G. Bewick. "Fast multiplication: algorithms and implementations," Technical Report CSL-TR-94-617, Stanford Univ., Apr. 1994.

[3] A. D. Booth. "A signed multiplication technique," *Quarterly J. Mech. Appl. Math.*, 4:236–240, 1951.

[4] L. Dadda. "Some schemes for parallel multipliers," *Alta Frequenza*, 34:349–356, May 1965.

[5] S. Dao-Trong and K. Helwig. "A single-chip IBM system/390 floating-point processor in CMOS," *IBM Journal of Research and Development*, 36(4):733–749, July 1992.

[6] R. E. Goldschmidt. "Applications of division by convergence," Master's thesis, M.I.T., June 1964.

[7] M. Hanawa et al.. "A 4.3ns 0.3 μm CMOS 54x54b multiplier using precharged pass-transistor logic," In *ISSCC Digest of Technical Papers*, pages 364–365, Feb. 1996.

[8] A. Inoue et al.. "A 4.1ns compact 54x54b multiplier utilizing sign select Booth encoders," In *ISSCC Digest of Technical Papers*, pages 416–417, Feb. 1997.

[9] R. Jessani and C. Olson. "The floating point unit of the PowerPC 603e microprocessor," *IBM Journal of Research and Development*, 40(5):559–566, Sept. 1996.

[10] J. A. Kowaleski, Jr. et al.. "A dual-execution pipelined floating-point CMOS processor," In *ISSCC Digest of Technical Papers*, pages 358–359, Feb. 1996.

[11] N. Ohkubo et al.. "A 4.4 ns CMOS 54 x 54-b multiplier using pass-transistor multiplexer," *IEEE J. Solid-State Circuits*, 30(3):251–257, Mar. 1995.

[12] C. V. Ramamoorthy, J. R. Goodman, and K. H. Kim. "Some properties of iterative square-rooting methods using high-speed multiplication," *IEEE Trans. Comput.*, C-21(8):837–847, Aug. 1972.

[13] E. M. Schwarz, L. Sigal, and T. McPherson. "A 400 MHz CMOS S/390 Floating Point Unit," to be published in *IBM Journal of Research and Development*, 1997.

[14] S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan. "A general proof for overlapped multiple-bit scanning multiplications," *IEEE Trans. Comput.*, 38(2):172–183, Feb. 1989.

[15] S. Vassiliadis, E. M. Schwarz, and B. M. Sung. "Hard-wired multipliers with encoded partial products," *IEEE Trans. Comput.*, 40(11):1181–1197, Nov. 1991.

[16] C. S. Wallace. "A suggestion for a fast multiplier," *IEEE Trans. Comput.*, EC-13:14–17, Feb. 1964.

[17] C. Webb et al.. "A 400 MHz S/390 microprocessor," In *ISSCC Digest of Technical Papers*, pages 168–169, Feb. 1997.

[18] R. Yu and G. Zyner. "167 MHz Radix-4 Floating Point Multiplier," In *Proc. of Twelfth Symp. on Comput. Arith.*, pages 149–154, Bath, England, July 1995.