# Fast Multiply and Divide for a VLSI Floating-Point Unit

*B. K. Bose, L. Pei, G. S. Taylor, D. A. Patterson*

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

*ABSTRACT*

This paper presents the design of a fast and area-efficient multiply-divide unit used in building a VLSI floating-point processor (FPU), conforming to the IEEE standard 754. Details of the algorithms, implementation techniques and design tradeoffs are presented. The multiplier and divider are implemented in 2 micron CMOS technology with two layers of metal, and occupy 23 square mm (23% of the entire FPU). We expect to perform extended-precision multiplication and division in 1.1 and 2.8 microseconds, respectively.

## 1. Introduction

High speed computation is essential for a large class of problems, like computer modeling and simulation, CAD/CAM, computer graphics, image processing, and robotics, where integer arithmetic lacks the range and precision of most of these real-world needs. Traditionally, floating point arithmetic has been slow in software and expensive in hardware. VLSI technology is now making it possible to have fast, inexpensive floating point arithmetic.

In keeping with increasing CPU performance, floating point computation needs to be fast. A balanced implementation should take into account the relative frequency of different floating point operations and implementation constraints, that in turn affect the choice of algorithms, the micro-architecture, the clocking methodology and the design style.

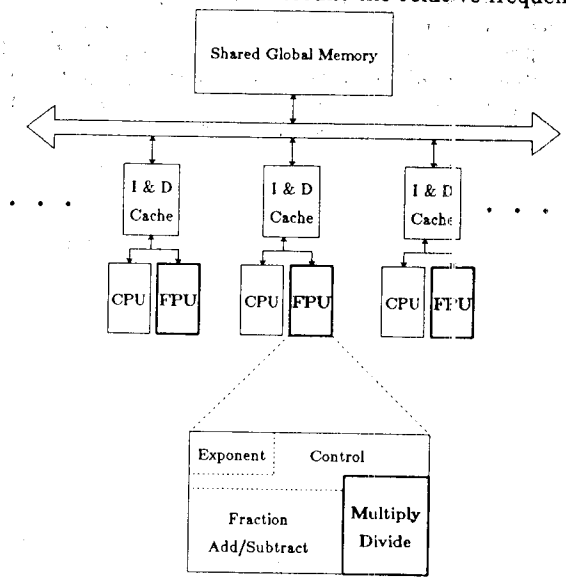


*Figure 1: The SPUR System*

In order to investigate these design tradeoffs, a single-chip floating-point processor has being designed. It is a tightly coupled coprocessor in the SPUR shared-memory multi-processor system [Hill86], providing instruction set enhancement for fast floating point computation (Figure 1).

The VLSI chip conforms to the IEEE Standard 754, providing the basic arithmetic functions for single, double and extended precisions, IEEE-style rounding, and detection of special operands and exceptions. It is designed in 2 micron, 2-layer-metal CMOS technology. This paper will focus on the multiply/divide section of this chip, which has a target speed of 1.1 microseconds for extended-precision multiply and 2.8 microseconds for extended-precision divide.

Several floating point units are commercially available, and some have recently been described in the literature [Nave80], [Shahan84], [Wolrich84], [Gavrielov86], [Troutman86]. We compare execution times between different implementations, to get an idea of the relative performance of our algorithms and implementation. We do not include multiple-chip designs in this comparison, since the design tradeoffs are then quite different. Table 1 summarizes the results for register-to-register operations.

*Table 1: Performance of Multiply and Divide in 1-chip VLSI FPUs*

| Implementation | Significand | Cycle Time (ns) | Multiply (μs) | Divide (μs) |
|---|---|---|---|---|
| Intel 8087 | 64 bits | 100 | 13.80 | 19.80 |
| National 32081 | 53 bits | 100 | 6.20 | 11.30 |
| Motorola 68881 | 64 bits | 60 | 3.12 | 5.04 |
| Western Elec. 32106 | 64 bits | 56 | 2.80 | 16.80 |
| Digital Equipment* | 56 bits | 100 | 2.80 | 4.70 |
| Fairchild Clipper* | 53 bits | 30 | 2.07 | 5.46 |
| SPUR | 64 bits | 140 | 1.12 | 2.80 |

*These numbers are typical, while the others are worst-case.

## 2. Choice of Algorithms

Recent papers [Gamal86], [Uya84] show that with current 2 micron technology, combinational 32 bit multipliers require about 40,000 transistors, and take up about 30 square mm. Full 64-bit array multipliers are still difficult to integrate onto a single chip. Even if we could build a multiplier that computed in a single cycle, it would be difficult to find an adder that was proportionately faster. Again, divide times must improve with multiply; otherwise, algorithm designers will be tempted to avoid divide. Addition and multiplication in VLSI arithmetic accelerators have received a lot of attention

lately, but not much work has been reported to build fast division schemes in VLSI. Higher radix non-restoring division is becoming feasible in VLSI, and here again, operation frequencies can serve as a guideline for choosing the most appropriate algorithm and for budgeting hardware. This is especially critical in VLSI, where propagation delay between modules within a chip and inter-chip delay are so different, directly affecting what goes onto a chip. Thus, chip area and I/O pins need to be allocated very carefully. Again, since there is a linear relationship between delay and fanout in MOS technology, putting more functionality on chip can adversely affect cycle time, since control signals have to now drive larger capacitance.

There have been several studies of various programs and benchmarks that are intensive in floating-point computation. We summarize results from Berkeley [Leung86] with those of Knuth[1971] and Gibson[1970] in Table 2. We see that add/subtract operations occur from 1.5 to 2.5 times more frequently than multiply operations, which in turn are 2 to 3 times as frequent as divide operations. The Lattice Filter seems to be an exception in that divisions occur much less often than in the others, and additions occur less frequently than multiplications. This table indicates chip resource allocation for a balanced design, where the proportion of hardware for add : multiply : divide should be close to the ratio of operation frequency. For example, a large chip area invested in an array multiplier may not be cost-effective without a proportionately fast divider.

*Table 2: Relative Frequencies of Floating Point Operations*

| Source | Add,Subtract | Multiply | Divide |
|---|---|---|---|
| Knuth | 2.30 | 1.00 | 0.38 |
| Gibson Mix | 1.80 | 1.00 | 0.39 |
| Whetstone | 1.80 | 1.00 | 0.50 |
| Lattice Filter | 0.75 | 1.00 | 0.08 |
| SPICE | 1.45 | 1.00 | 0.35 |

**Operation frequencies are normalized to Multiply.**

**Algorithms can have quite different area and time costs depending on their implementation technology, whether it is Schottky TTL, ECL MSI, ECL gate arrays or MOS VLSI. Estimates of area and delay depending on gate count ignore such realities as fan-in, fan-out, interconnect and chip crossings. In VLSI, datapath pitch is usually determined by interconnect requirements, like the number of data busses that need to traverse it. Thus, the size of a variety of circuits is the same in one direction, while varying in the other. Naturally, some circuits will be much more densely packed than others, and so merely counting the number of gates in a circuit block can give a misleading idea of the area it requires. In Table 3, we present the relative areas of some basic circuit blocks, together with their relative delays, for driving identical loads.**

To illustrate this technology dependence, let us see whether it is advisable to build Booth recoding into an iterative multiplier in the two technologies. Without recoding, we require eight rows of adders to reduce one multiplier byte into its partial 'sum' and 'carry' vectors; and with recoding, we require four 4:1 multiplexors and four adders. Note that some CSA rows can evaluate in parallel, resulting in 5 and 3 effective adder delays in the two cases. From Table 3, we see that in Bipolar LSI, the areas of the two schemes are the same, and the scheme with Booth recoding is 7% faster. In CMOS, the area with Booth recoding is 37% less than without recoding, and is also 33% faster. Clearly, Booth recoding is preferable in CMOS, whereas it is a toss-up in Bipolar LSI.

*Table 3: Circuit Area-Time Relationships in Bipolar LSI and CMOS VLSI*

| Circuit | Technology | | | |
|---|---|---|---|---|
| | Macrocell ECL LSI | | Custom CMOS VLSI | |
| | Area | Time | Area | Time |
| 2:1 Multiplexor | 1 | 1.0 | 1 | 1 |
| 4:1 Multiplexor | 4 | 2.0 | 2 | 1 |
| 2-in OR,NAND | 1 | 0.8 | 1 | 2 |
| 2 2-in Exclusive-OR | 1 | 1.2 | 2 | 2 |
| D Flip-Flop | 2 | 1.2 | 2 | 3 |
| Full Adder | 4 | 1.2 | 8 | 3 |
| 8-bit barrel-shift | 7 | 3.0 | 7 | 1 |

Area and time of circuit blocks are normalized to a 2:1 multiplexor in each technology. In ECL, for example, a full adder is the same size as a 4:1 multiplexor, whereas in CMOS, a full adder is four times the size of a 4:1 multiplexor.

Using process yield curves, we estimated that the maximum size of our chip can be 10mm × 10mm. Accounting for periphery, this leaves about 9mm on a side for circuits. Allowing 25% area for control and routing, this leaves 60 square mm. for the exponent and fraction datapaths. Using the relative frequencies in Table 2, multiply and divide together accounts for 40% of the operations, and so are allocated close to that percentage of the datapath, i.e., 24 square mm. Given this area constraint, we moved away from purely combinational algorithms to iterative ones.

## 3. Implementation Considerations

**The custom chips in the SPUR system are being designed in CMOS technology. Complimentary MOS provides a variety of desirable characteristics for implementation of a system of this complexity. CMOS technology, with its characteristic scalability, provides a good speed-power product, making it attractive at high levels of integration. CMOS has high noise immunity, high circuit density, provides a wide choice of design styles, and is compatible with the CAD tools at our disposal.**

The maximum width of the fraction datapath is 73 bits (for example, the partial product vectors for multiply). Delay in control signals that run the entire length of the datapath has a large impact in the delay in and between modules in the datapath. If there is any appreciable resistance in these control lines, the RC delay can become a significant fraction of module delay, leading to slower computation rates and large clock non-overlap times to protect against clock skew. We chose a process with two available layers of metal, with control and data signals running orthogonally for the most part in the two metal layers, thus virtually eliminating any resistive delay. To minimize clock skew, we scale the drivers of the control lines to match the capacitances they have to drive, so that control delay is held between very tight tolerances for the entire width of the datapath.

To allow for a mix of static and dynamic design styles, we chose a four-phase clocking scheme, used in the

other chips in the SPUR system as well. With an on-chip register file with an access time of one phase, this allows two accesses, one for read and one for write per cycle, with the two intermediate phases going to precharge the dynamic busses. The cycle time is limited by the register file read and write time. The current technology allows transistors with minimum channel length of 2 microns, with a minimum size inverter discharging 1pf capacitor in one phase. The present clocking scheme is shown in Figure 2.

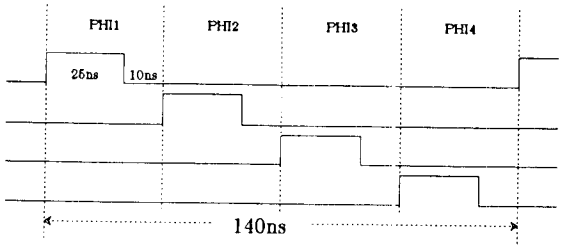


*Figure 2: SPUR Clocking Scheme*

### 3.1. Pipelining the Inner Loop

A key factor in making our design area-efficient is that the multiply and divide schemes have much in common, enabling significant sharing of hardware. Figure 3 shows the entire datapath for multiply and divide. A key factor in making our design achieve the speeds it did is the fact that the inner loop, the heart of our iterative algorithms, is completely pipelined, with different stages of the pipeline carefully adjusted for balanced delay. Modules that are density-critical are designed using dynamic circuits, and modules that have rigid timing constraints, and where precharge times cannot be overlapped with evaluation times, are designed using static circuits. The hardware blocks that are not shared are the multipler Booth recoder and the divider quotient selector and accumulator, and account for 8% of the entire multiply/divide unit.

The following sections describe the algorithmic and implementation considerations for this multiply/divide datapath.

## 4. The Multiplier

Since it is not feasible to build a 64 × 64 array multiplier as part of a single-chip FPU with currently available technology, we considered several iterative schemes. A 64 × 32 array requires 2 iterations to compute the full product, but takes up about twice as much area as a 32 × 32 array, which requires 4 iterations. Even the area of a 32 × 32 array just for the multiplier exceeds our area budget for both multiply and divide.

### 4.1. The Algorithm

Our multiplier is implemented in nine iterative steps. Each iteration implements a 64 bit by 8 bit multiplication. In each iteration, four overlapped triplets of multiplier bits (9 bits) are decoded by a modified Booth recoder. Four multiplicand multiples of magnitude +2MCD, +1MCD, -1MCD and -2MCD are needed per iteration, along with 0. The relative cost of a multiplexor

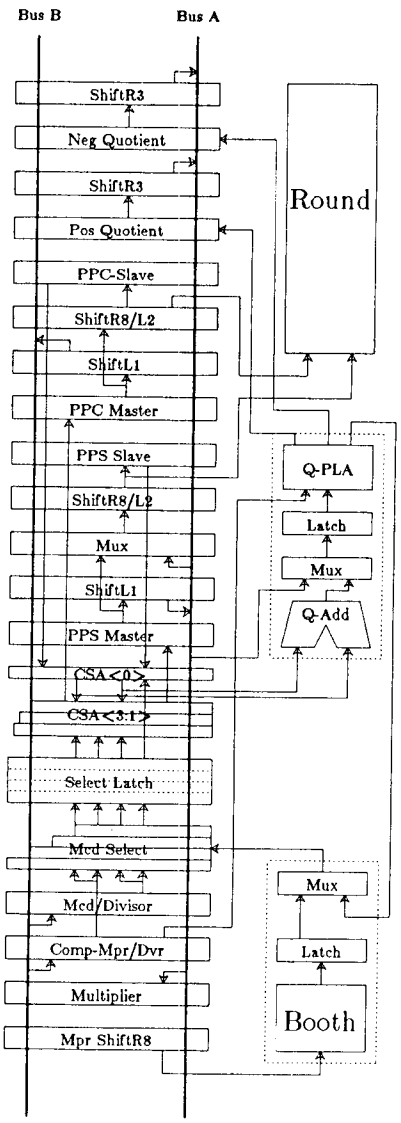


*Figure 3: The Multiply-Divide Datapath*

The datapath has the following sections, from bottom to top: input latches and multiplier shifter; multiplicand/divisor select; carry-save-adder tree; partial sum and partial carry formation; quotient accumulation.

compared to that of an adder makes Booth recoding feasible in CMOS, exchanging six rows of carry-save-adders with four rows of multiplexors and four carry-save-adders in the datapath. Also, separating the recoding from the carry-save-addition allows us to evaluate them in separate time-slots in our pipelined implementation, thus avoiding six CSA rows in the critical path.

The four overlapped triplets of multiplier pairs generate the four multiples of the multiplicand. They are added to the partial 'sum' and 'carry' terms of the previous iteration, using an array of four carry-save-adders (CSA). Note that the four multiples of the MCD are shifted left 2 bits with respect to each other, depending on

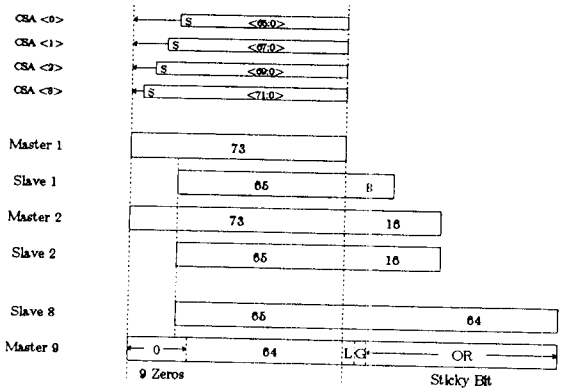


*Figure 4: Forming the Product*

The width of the CSA, multiplexors and PPS/PPC are 73 bits. After right-shifting 8 bits, the sum of the two partial products is 65 bits (64 bit magnitude and 1 sign bit).

the significance of each multiplier triplet. The partial 'sum' and 'carry' are shifted left 8 bits and 7 bits respectively, when looping them back to be the new inputs of the CSA for the next iteration. Since there are negative as well as positive operands, both in two's complement form, the multiplexers and CSA must be fully sign-extended to the left (MSB) side.

A carry-look-ahead adder is necessary at the start of the multiply operation, to produce the complement of the multiplicand. It is also necessary at the end to form the final result by adding the partial product vectors. Since the fraction unit has such an adder already, we share this module instead of duplicating it in the multiply/divide unit. This increases the setup and completion times by one cycle, but reduces the area of the unit by 14%.

### 4.2. The Multiply Inner Loop

The CSA tree, containing four rows of adders, is the critical path for the multiplier inner loop. To reduce this delay, two rows of the CSA tree, CSA<1,0>, evaluate in parallel, reducing the net delay to 3 CSA stages. This makes the interconnect less regular, but provides a 25% speed improvement in the CSA stage. For the divider, only one row of the CSA tree is necessary, and so we can use the isolated CSA row to advantage. Figure 5 shows the organization of the CSA tree.

In the multiplier inner loop, multiplicand selection is overlapped with shifting and rounding the partial product vectors. During each cycle, two phases are for signals controlled by the master, and the other two phases are for the evaluation of signals controlled by the slave. Blocks that are controlled by the master are the Booth recoder and the CSA tree, and blocks controlled by the slave are the multiplexor set and the shifter between master and slave partial product latches. Figure 6 shows the multiplier pipeline.

We have shown how the different parts of the multiply inner loop are pipelined. Several of the modules were designed using dynamic circuits to meet area, interconnect and timing constraints. Table 4 shows the design

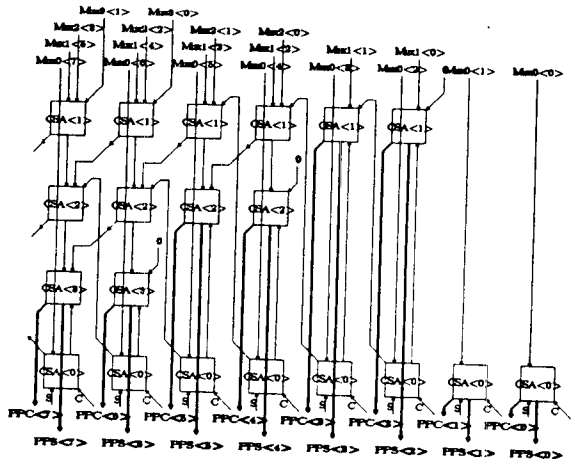

*Figure 5: Least Significant Bits of the CSA Tree*

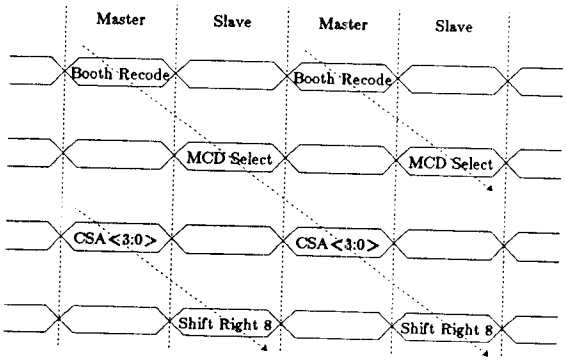


*Figure 6: The Multiplier Pipeline*

style used for building the modules in the multiply/divide inner loop, together with their area and time relationships, normalized to the Booth recode block.

The four rows of carry-save-adders take the longest time among the pipeline components. The dynamic CSA design was 16% smaller than its static counterpart, and was still able to meet the timing requirements. If we went to a more aggressive clocking scheme, it would be feasible to build the four multiplexors and the CSA tree static, and modify the pipeline to have only two stages. One stage could perform Booth recoding and MCD selection, while the other could do CSA evaluation and the right shift. Not only could the disparity between the stages become smaller, but we could also utilize two out of the four clock non-overlap times.

*Table 4: Area-Time Relationships of Multiply Function Modules*

| Function | Design Style | Area | Time |
|---|---|---|---|
| Booth Recode | Dynamic | 1.0 | 1.0 |
| MCD Select | Dynamic | 2.6 | 0.9 |
| CSA Tree | Dynamic | 9.9 | 4.3 |
| Partial Sum & Carry | Static | 8.7 | 0.5 |

Area and time are normalized to the Booth recode block. The CSA, for example, is almost ten times larger and more than four times slower than the Booth Recode block.

### 4.3. Rounding

The ability to perform unbiased rounding with error less than half a unit in the last place requires three extra bits, called the Guard, Round and Sticky bits. The Guard and Round bits are used if the intermediate result of a division is between .5 and 1 and hence requires a one-bit normalizing left shift. The Sticky bit is equal to zero only if all subsequent bits in a result of infinite precision are zero and is used to identify the half-way case for unbiased rounding.

During a multiply operation, the vectors containing the partial products are shifted right eight bits before being returned for the next iteration. These two eight-bit quantities are added together, and ORed to form the 'partial sticky' bit. This is fed back to the rounding adder for the next iteration. At the end of the final rounding addition, bit<0> of the result is the most significant bit of the rounding adder result, followed by the Guard bit. The Round bit is zero, since the result must be in a range between 1 and 4. The OR of the remaining bits of the rounding adder provides the Sticky bit, as shown in Figure 7.

Since each iteration takes half a cycle in this pipelined scheme, partial product evaluation takes four and a half cycles. It takes almost that long, again, for evaluating the MCD complement, the final carry-propagate addition and the rounding, making the total multiply latency 8 cycles. Some of this can be saved by duplicating the fraction ALU in the multiply/divide unit, and also duplicating the rounding PLA that generates the least significant bit. Currently, both of these are shared with the fraction unit; if these were duplicated, the potential time saving can be 20%.

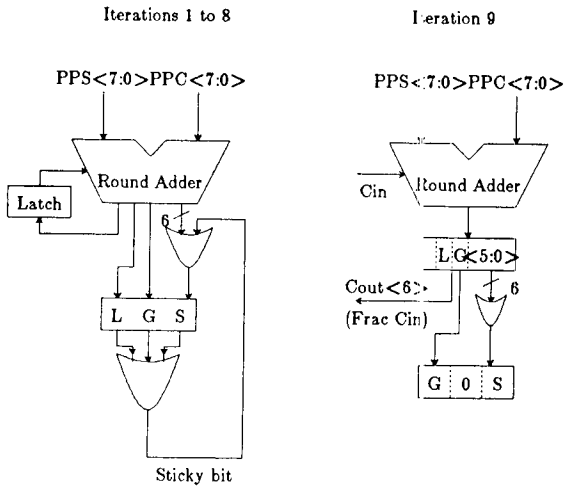


*Figure 7: Multiply Rounding*

## 5. The Divider

Restoring divide is the least expensive approach for radix-two division, but the cost increases exponentially with the number of bits generated per iteration. The same is true of parallel-serial schemes. Multiplicative inverse schemes produce incorrectly rounded quotients and inexact remainders, and later fix-ups can be expensive and time-consuming. SRT division [Robertson65], [Atkins67] focuses attention on quotient digit selection, and the remainder iteration does not require back-tracking. Higher radix SRT division schemes are likely to provide significant gains in area and speed, as better ways are found to provide compact quotient-selection logic, and concurrency between different portions of the algorithm (like partial remainder formation and quotient selection) is exploited.

### 5.1. The Algorithm

Our algorithm is based on radix-four, non-restoring division, using estimates of the divisor and partial remainder. The radix-four quotient digits are expressed using redundant representations of -2, -1, 0 +1 and +2, and the partial remainder is irredundant. This redundancy in the quotient digits permits less precision in comparing the divisor and partial remainder to select a quotient digit. The required precision required in inspecting the partial remainder and the divisor can be determined using P/D plots. It can be shown that six bits of partial remainder and four bits of divisor are needed to determine the next quotient digit [Freiman61],[Atkins68].

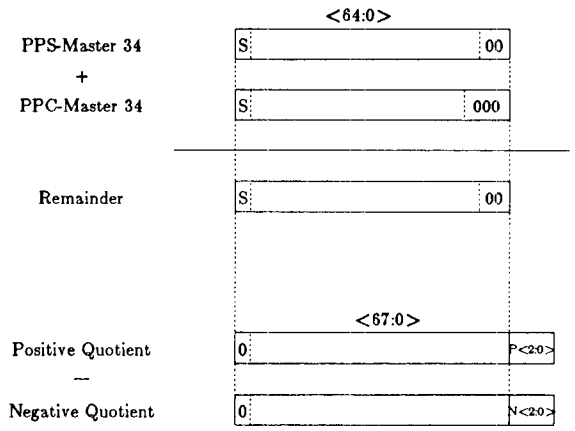


*Figure 8: Forming the Remainder and Quotient*

The division is done iteratively, with two quotient bits computed per iteration, with the equation expressed as follows:

$$p_{(j+1)} = r \times p_j - q_{(j+1)} \times d$$

where j = index of the recursive loop <33:0>
$p_j$ = partial remainder after the jth loop
$p_0$ = dividend
$q_{(j+1)}$ = quotient digit after the jth loop
d = divisor
r = radix <4>

The hardware loop for generating the next remainder and the next quotient estimate contains an eight-bit carry-lookahead-adder, which generates six bits of truncated partial remainder. Together with these six bits, four bits of the truncated divisor are sent to the quotient selection logic, which in turn generates three bits, representing one of the five possible values of the quotient digit. Depending on the sign of the quotient digit, it is channeled into one of two registers, one holding positive and the other hold-

ing negative quotient estimates. These registers are shifted left two bits per iteration. The quotient selection logic is decoded to control a multiplexor, that decides what multiple of the divisor to use for the next iteration.

### 5.2. The Divide Inner Loop

For the divider, the partial product latches are used to hold the partial remainders, with one modification. Multiplexors in front of these latches let us load the master with the dividend at the very beginning of a divide. The PPC latch gets loaded with zero. The partial remainders are shifted left by two bits after every iteration.

Since the divider uses only one row of the CSA tree, one of its rows is separated out from the rest of them, to enable this fast and direct path for the divider. Partial remainder evaluation and quotient estimation are done in parallel. In the remainder evaluation, the CSA and left shift operations are split into master and slave phases. For the quotient evaluation, the 8-bit estimation adder is evaluated at the master time, while the quotient selection PLA and the divisor selection is done at the slave time. The divider pipeline is shown in Figure 9.

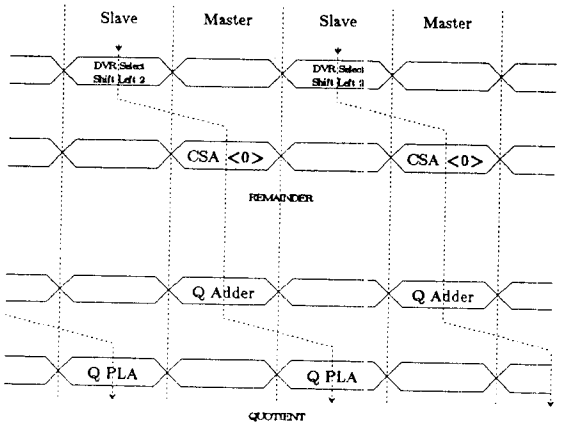


*Figure 9: The Divider Pipeline*

There are several reasons for our efficient divider. First, we use radix four, allowing the evaluation of two quotient bits per iteration. Second, non-restoring divide allows us to look ahead for quotient selection, keeping exact quotient determination until later, without backtracking at every iteration. Third, a small degree of redundancy in the quotient digit representation keeps us from having to generate more costly multiples of the divisor, albeit requiring an increase in quotient selection logic. Lastly, the concurrency between partial remainder formation and quotient selection [Atkins74] significantly increases algorithm efficiency. Table 5 shows the relative area-time cost for different sections of the divide pipeline, normalized to the divisor select block.

### 5.3. Quotient Selection

An integral part of the divide scheme is the logic for quotient selection. To keep the quotient selection logic from becoming the critical path, the quotient adder and PLA were split into two phases. The adder can now be

*Table 5: Area-Time Relationships of Divide Function Modules*

| Function | Design Style | Area | Time |
|---|---|---|---|
| Divisor Select | Dynamic | 1.0 | 1.0 |
| CSA Row | Dynamic | 4.1 | 3.2 |
| Quotient Adder | Static | 0.4 | 3.5 |
| Quotient PLA | Static | 0.5 | 4.8 |
| Quotient Accumulate | Static | 6.4 | 0.8 |

Area and time are normalized to the divisor select block. The quotient PLA, for example, is half as large and almost 5 times slower than the divisor selector.

dynamic, since it can be precharged when the PLA evaluates, thereby saving area. The PLA that generates the two bits of quotient per iteration, gets its inputs at the beginning of every slave phase and has to evaluate the quotient bits by the end of that phase. To achieve this strict timing requirement, we went with a pseudo-static PLA design. At the expense of a little DC power, output evaluation can be done in a single phase time. There are ten inputs to the PLA (6 bits from the quotient adder, and 4 from the divisor), and three outputs (2 quotient bits and the sign bit). Two optimizations were done to reduce the size of the PLA. Firstly, the quotient sign bit is the same as the sign of the partial dividend, and hence does not have to be a PLA output. Secondly, with optimal encoding of the PLA outputs, the number of product terms were reduced significantly. Given the small number of outputs, it was possible to exercise the PLA minimization tools [Rudell86] to find what encoding of outputs resulted in the smallest number of minterms (*). Table 6 summarizes the results.

Quotient selection takes the longest time for radix 4 division; remainder evaluation takes only 40% of the time of quotient selection. For radix 16 division, there is no extra cost in remainder evaluation, but quotient selection area increases by about 6 times. In the critical path, we now require several quotient adders that work in parallel, with the final result muxed out to the quotient PLA. The extra time cost is about 30%.

*Table 6: Effect of Output Encoding on the Quotient PLA*

| Output | Twelve Unique Output Encodings | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q=0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 |
| Q=1 | 1 | 1 | 3 | 0 | 0 | 2 | 2 | 3 | 3 | 0 | 1 | 1 |
| Q=2 | 2 | 3 | 1 | 2 | 3 | 0 | 3 | 0 | 2 | 1 | 2 | 0 |
| P-Terms | 43 | 26 | 26 | 26 | 26 | 29 | 30 | 27 | 27 | 26 | 44 | 25* |

### 5.4. Rounding

The divider must provide three rounding bits along with a 65-bit result. Since two quotient bits are generated at every iteration, 34 iterations are necessary to generate the partial remainder and quotient vectors. After adding the two partial remainder vectors, the sign of the remainder is returned to the multiply-divide unit. The OR of the rest of the bits provides the 'partial sticky' bit. Using a 3-bit rounding subtractor, which uses the complement of the sign of the remainder as carry, bits <2:0> of the quotient vectors are subtracted. The least significant bit is ORed with the 'partial sticky' bit to form the final Sticky bit, while the other two bits of result provide the Guard and Round bits. The carry out of the subtractor goes out to the fraction adder, as shown in Figure 10. It would be possible to eliminate most of this logic if we had a 68-bit ALU. But since we share the ALU with the fraction unit, which has only a 65-bit ALU, we have to retain this logic.

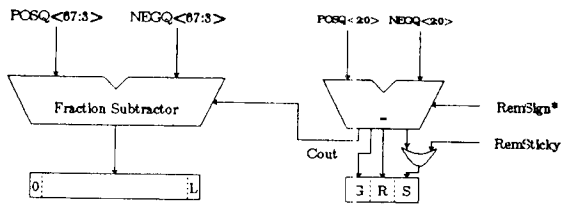


*Figure 10: Divide Rounding*

## 6. Scaling Implications

Further enhancement in performance will come from two inter-related causes. As technology continues to scale to smaller and smaller geometries, individual transistors and logic gates will improve in performance. Scaling will also enable more logic to be put on a single chip, allowing for further algorithmic enhancements. Reducing the feature size by $\alpha$ causes area to decrease by the square of $\alpha$ and the speed to increase by almost the same factor. Table 7 indicates the effects of halving the smallest geometry on chip, from 2 microns to 1 micron, on algorithms and their area and time.

*Table 7: Technology Impact on Algorithm Area-Time Cost*

| Algorithm | Technology | | | |
|---|---|---|---|---|
| | 2 micron | | 1 micron | |
| | Area | Time | Area | Time |
| Multiply (64 × 8, iterative) | 1.0 | 1.00 | 0.25 | 0.25 |
| Multiply (64 × 32, array) | 1.9 | 0.65 | 0.48 | 0.16 |
| Multiply (64 × 64, array) | 3.8 | 0.35 | 0.95 | 0.09 |
| Divide (SRT,r=4) | 1.0 | 1.00 | 0.25 | 0.25 |
| Divide (SRT,r=16) | 1.5 | 0.70 | 0.38 | 0.18 |
| Divide (Prescale, r=256) | 3.5 | 0.40 | 0.88 | 0.10 |

**Area and time are normalized to those for our implementation in 2 micron CMOS. For example, our multiply scheme will be 25% its present size when built in 1 micron CMOS and will be 4 times as fast. Again, a radix 16 divider will be 50% larger than radix 4 in 2 micron CMOS, but 38% its present size in 1 micron CMOS.**

**We can expect significant speed enhancements by matching technology to algorithm, while retaining a balanced design. Large combinational multipliers will soon be feasible as components in an FPU. Rounding hardware will need to be integrated into the multiply unit, to keep it from becoming the bottleneck.**

Dividers, being inherently sequential in nature, are harder to speed up. The size of escalating cost (in area and time) in quotient selection logic will probably limit the use of non-restoring divide to radix 16 [Taylor85]. Partial remainder evaluation will be virtually unaffected, but the quotient selection logic will increase from radix 4 by about 6 times. Prescaling schemes [Ercegovac85] are worth exploring, for generating more quotient digits per iteration. Eight and even sixteen bits per iteration seem feasible, provided initial setup, final remainder adjustment and data flow can be handled efficiently. The estimate in Table 7 takes into account the fact that we need two 64 × 8 multipliers for the prescaling, and the datapath width increases by eight bits; on the other hand, quotient selection logic is greatly simplified.

## 7. Conclusion

VLSI provides certain unique opportunities and constraints in the design and implementation of high-performance arithmetic. Chip area and delay closely interact to affect all levels of design, from available functionality and choice of algorithms to clocking strategies and circuit design styles. In particular, we study the design tradeoffs offered by CMOS technology, the dominant VLSI technology in the 1980's. In an attempt to explore tradeoffs in this complex and inter-dependent design space, we present the design of a fast and compact multiply/divide unit implemented for a single-chip FPU.

## 8. Acknowledgements

Prof. Kahan has helped us understand the intricacies of the IEEE Standard and encouraged us all along; Prof. Hodges has always been available to advise us on circuit issues. Glenn Adams wrote portions of the FPU simulator, which Corinna Lee has completed and tested. Albert Wang and Timothy Hu have helped in layout and circuit simulation. Principal funding for the project is by DARPA, under contract N00039-85-C-0269.

## 9. References

[Atkins67]- D. E. Atkins, 'The Theory and Implementation of SRT Division,' Report No. 230, Dept. of Computer Science, Unviersity of Illinois, June, 1967.

[Atkins68]- D. E. Atkins, 'Higher-Radix Division Using Estimates of the Divisor and Partial Remainders,' IEEE Trans. Computers, Vol. C-17, No. 10, October 1968, pp. 925-934.

[Atkins74]- D. E. Atkins & U. Kalaycioglu, 'Concurrency in Generalized Radix Non-Restoring Division,' Proc. Twelfth Allerton Conference on Circuit and Switching Theory, October 1974, pp. 628-640.

[Ercegovac85] - M. Ercegovac & T. Lang, 'A Division Algorithm with Prediction of Quotient Digits,' Proc. Seventh IEEE Symposium on Computer Arithmetic, June 1985, pp. 51-56.

[Freiman61] - C. V. Freiman, 'Statistical Analysis of Certain Binary Division Algorithms,' Proc.IRE, Vol. 49, January 1961, pp. 91-103.

[Gamal86]- A. E. Gamal et al, 'A CMOS 32b Wallace Tree Multiplier-Accumulator,' Proc. ISSCC, February 1986, pp. 194-195.

[Gavrielov86] - M. Gavrielov & L. Epstein, 'The NS32081 Floating-Point Unit,' IEEE Micro, April 1986, pp. 6-12.

[Gibson70] - J. C. Gibson, 'The Gibson Mix,' IBM Systems Development Div., Tech. Report, June 1970.

[Hill86] - M. D. Hill et al, 'SPUR - A Multiprocessor Workstation,' to appear in IEEE Computer, November 1986.

[Knuth71]- D. E. Knuth, 'An Empirical Study of FORTRAN Programs,' Software: Practice and Experience, Vol. 1, No.2, April 1971, pp. 105-133.

[Leung86] - B. Leung & Y. M. Lin, ' Statistics on Floating Point Arithmetic,' CS 252 Report, May 1986.

[Nave80] - R. Nave & J. Palmer, 'A Numeric Data Processor,' Proc. Intl. Solid-State Circuits Conference, Febru-

ary 1980, pp.108-109.

[Robertson65] - J. E. Robertson, 'Methods of Selection of Quotient Digits during Digital Division,' File No. 663, University of Illinois, Urbana, June 1965.

[Rudell86]- R. Rudell, 'ESPRESSO,' 1986 VLSI Tools, Report No. UCB/CSD 86/272.

[Shahan84] - V. Shahan, 'The MC68881: The IEEE Floating Point Standard Reduced to One VLSI Chip,' Proc. IEEE Computer Conference, March 1984, pp. 172-176.

[Taylor85] - G. S. Taylor, 'Radix 16 SRT Division Methods With Overlapped Quotient Selection Stages,' Proc. Seventh IEEE Symposium on Computer Arithmetic, June 1985, pp. 64-71.

[Troutman86] - W. W. Troutman et al, 'Design of a Standard Floating-Point Chip,' IEEE J. of Solid-State Circuits, Vol. SC-21, No.3, June 1986, pp. 396-399.

[Uya84] - M. Uya et al, ' A CMOS Floating Point Multiplier,' IEEE J. Solid-State Circuits, Vol. SC-19, No.5, October 1984, pp. 697-702.

[Wolrich84] - G. Wolrich et al, 'A High Performance Floating Point Coprocessor,' IEEE J. of Solid-State Circuits, Vol. SC-19, No.5, October 1984, pp. 690 - 696.