THE DESIGN OF A 64-BIT INTEGER MULTIPLIER/DIVIDER UNIT

David Eisig, Josh Rotstain
Intel Israel Ltd.
Haifa, Israel

Israel Koren
Dept. of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

Abstract

The highlights of the design of an integer multiplier/divider unit within a 64-bit processor are presented. The final design is the result of a compromise between performance, complexity, and transistor count. It is optimized for two specific operations with the same hardware being shared by the remaining operations. Thus, for example, the multiplier can be configured for the execution of several different multiply operations and its hardware is also heavily utilized in division. The divider design is optimized for repetitive division by small numbers, since this is a characteristic of several important applications planned for the processor. For such small divisors, the reciprocal is calculated and stored in a content-addressable memory. The stored reciprocals can then be used to generate quotients through fast multiplication. Simulations of the planned applications show a 20 to 30 percent performance increase over alternative designs.

1 Introduction

We present in this manuscript some of the decisions that were made during the design of an integer multiply/divide unit. This is a separate unit within a 64-bit integer arithmetic unit in a high-performance 64-bit processor. The multiply/divide unit executes several types of multiply, divide and remainder instructions and provides high performance within a given transistor count budget.

The instructions that are executed include:

MULL – a 64-bit multiply with the result being the low order 64 bits of the product. An overflow exception must be generated when appropriate.

PACKED_MULL – four 16-bit multiply operations performed in parallel. The 64-bit result consists of four 16-bit products; each constitutes either the low order or the high order half of the 32-bit result.

DIV – a 64-bit integer divide.

REM – a 64-bit integer remainder.

Unit design goals and constraints

In view of the planned applications for the 64-bit processor, higher priority is assigned to two particular operations. One is the packed data multiply operation (i.e., the PACKED_MULL instruction). This instruction must provide a throughput of one instruction per clock with a latency of 2 clocks. The regular 64 × 64-bit multiply can tolerate up to 8 clocks throughput and latency.

As for the divide and remainder instructions, special emphasis is put on divide operations with a divisor which is 13 bits long or shorter. In the planned applications for the designed processor, integer divide operations with such a small divisor are expected to be very common. Moreover, a very small number of divisors (each of length 13 bits or less) are likely to be used repeatedly. The selected design supports a low execution time for such operations, while regular divide or remainder operations would take a substantially higher execution time. The details of this design are presented in Section 3.

The most significant constraint in the design was the limited transistor budget of 150K. Being a 64-bit unit implied that in the implementation of all operations besides the packed data multiply and the repeated division with a small divisor, the major concern would be the required number of transistors. Hence, these implementations needed to reuse as many of the already available circuits (used for implementing the high priority operations) as possible.

Other severe constraints on the unit design were the aggressive clock period and input/output timing requirements. These determined the amount and partitioning of functionality among pipeline stages.

2 Multiplier Design

The multiplier must be capable of executing the packed data multiply as well as the ordinary 64 × 64 multiply. The first multiplies four sets of 16-bit

1063-6889/93 $03.00 © 1993 IEEE

multiplier-multiplicand pairs. All four are executed in parallel, with each providing either the 16 least significant bits or the 16 most significant bits of the product. The 64 × 64 multiply calculates the low 64 bits of the result and generates an overflow indication if the high 64 bits of the product are not just a sign extension.

The underlying multiplication algorithm is the commonly used radix-4 modified Booth's algorithm for both operations [2]. Since the packed data multiply has the highest priority, the multiplier was designed as four small independent 16 × 16 multipliers. For each 16-bit multiplier nine partial products are generated to allow either a signed or an unsigned multiplication. These partial products are accumulated using three levels of carry-save adders. The first and second levels are ordinary $3 \rightarrow 2$ units reducing the number of operands to 6 and 4, respectively. The last level is a $4 \rightarrow 2$ unit, reducing the final number of operands from 9 to 2. These steps, which include the generation of the partial products (i.e., 0, ±1, ±2 times the 16-bit multiplicand) through suitable multiplexors (MUXs) and the carry-save addition, are completed within one clock cycle. In the second cycle the two operands are added using a carry-propagate adder. The partial product reduction and the final addition are each performed in separate stages of a 2-stage pipeline, thus providing four 16-bit results per cycle.

Due to the overall limited transistor budget the 64×64 multiply operation uses as much of the circuitry for the packed data multiply as possible. Therefore, the carry-save adder which consumes the majority of transistors was designed as a reconfigurable carry-save tree with MUXs that can connect the four separate trees to form a single 80-bit wide tree. Also, full sign extension was performed in the separate trees, although a simple technique to avoid this is very well-known [2]. The reason for this design is that the additional positions in the carry-save tree were needed anyway when used as part of the 80-bit tree.

The 64 × 64 multiply is executed as a sequence of four 64 × 16 multiplications, the results of which are added and shifted in the second stage of the multiplier to produce the final result (for a total of 5 clock cycles).

In each 64 × 16-bit multiply, the four independent 16 × 16-bit multiply units are provided with the same common 16-bit multiplier. Each of these multiply units also receives the most significant bit of the previous 16-bit multiplicand (of lower significance) to allow forming the multiples ±2 (this bit is zero in the case of the first multiplicand). The CSA tree is configured by a small number of multiplexors to concatenate the sign-extension portions of the four independent CSA trees, while at the same time bypassing the parts of the CSA tree not required in the 80-bit partial-product summation.

In a similar manner, a 64×80 multiply can easily be performed as a sequence of five 64 × 16-bit multiplications. This is done in order to speed up the divide and remainder instructions in the case of small divisors, as described in the next section.

3 Divider Design

In order to speed up division by small numbers, the divider employs two division procedures rather than one. A standard radix-4 SRT division algorithm is used in the general case. A multiply by the reciprocal procedure, $Q = A \cdot 1/B$ is used in cases where the reciprocal of the divisor has been precalculated. A reciprocal of the divisor is calculated and stored in a content-addressable memory (CAM) whenever a divisor of 13 bits or less is encountered for the first time. This is done for small divisors only, since small divisors are frequently used in several inner loops within the major application programs that will run on this processor. Also, the reciprocal of a small divisor requires less storage space and can be calculated faster, making this approach even more attractive.

The reciprocals are calculated using the standard radix-4 division with a dividend of 1. The result is then incremented at the least significant bit to ensure a positive error. Denote the calculated estimation of the reciprocal of B by B_r. Then,

$$B_r = 1/B + \epsilon$$

where ϵ is the error, satisfying $\epsilon > 0$. When multiplying the reciprocal retrieved from the CAM by a dividend A we obtain

$$A \cdot B_r = A \cdot (1/B + \epsilon) = A/B + A \cdot \epsilon$$

Integer division should satisfy

$$A/B = Q + R/B$$

where Q and R are the integer quotient and remainder, respectively, with $R \leq B - 1$. Thus,

$$A \cdot B_r = Q + R/B + A \cdot \epsilon$$

and the sum of the last two terms must be smaller than 1 in order to have a correct quotient. The maximum value that R/B can assume is

$$R/B \leq (B-1)/B = 1 - 1/B \leq 1 - 2^{-13}$$

and $A < 2^{64}$, therefore

$$\max\{R/B + A \cdot \epsilon\} \leq 1 - 2^{-13} + 2^{64} \cdot \epsilon < 1$$

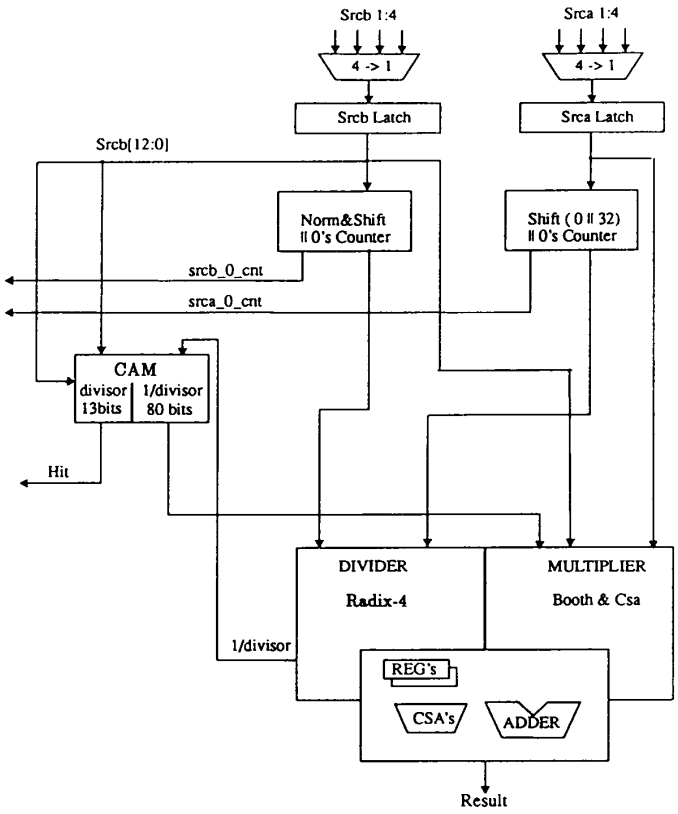

Figure 1: The general organization of the multiplier/divider unit.

Consequently,

$$\epsilon < 2^{-77}$$

In other words, the precision of the stored reciprocal must be 78 bits or higher. The length of 80 bits has been selected since the multiplier circuitry operates on multiples of 16.

The reciprocals are stored in a CAM (a.k.a associative memory) containing eight elements. A pseudo LRU (least recently used) algorithm is used for entry replacements. The decision regarding the number of entries in the associative memory unit was influenced by the requirements of the application software as well as by cost considerations. An important feature of the planned application is that about 95% of the divisions are by a divisor having 13 or less bits. A typical inner loop that includes division may be repeated hundreds to thousands of times, and may simultaneously involve around four separate small divisors.

The speed of the multiply-by-reciprocal division procedure depends on the speed of the multiplier circuitry. The multiply hardware described in Section 2 can perform a 64 x 80-bit multiplication in 6 clocks only, thus meeting the requirements of the divide algorithm. The fast divide operation can therefore take a total of 7 to 8 clocks. The only drawback is that the first time a division by a certain divisor occurs, it takes longer than a normal division. This happens since the reciprocal is calculated to an extended precision of 80 bits, then incremented, and finally used in a multiply operation. All these steps take about 47 clocks. In contrast, a standard division takes 3 to 35 clocks depending on the divisor. There is also the possibility that a spurious one-time division will replace a desired reciprocal from the associative memory.

The stored reciprocals are also used to speed up the remainder instruction. If the reciprocal is available, the quotient is first obtained by calculating $Q = A \cdot B_r$ (where B_r is the stored reciprocal). Then $Rem = A - Q \cdot B$. Thus a fast remainder can be calculated in 9 to 10 clocks, rather than up to 35 clocks.

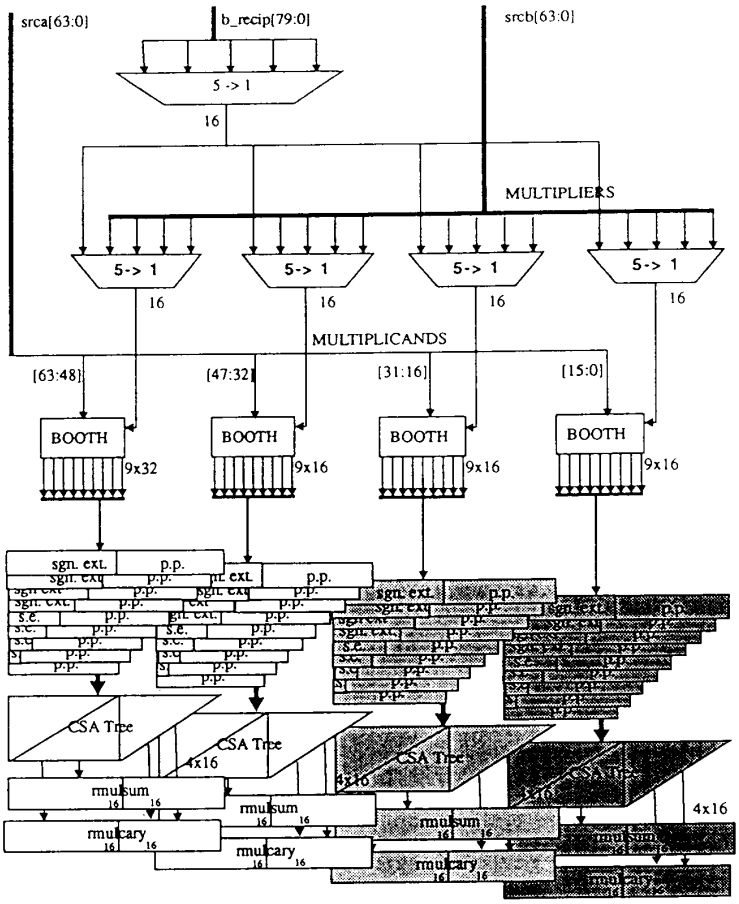

Figure 2: Four 16 × 16-bit multipliers.

4 Implementation Notes

The integer multiplier/divider is a separate unit in a high-performance processor implemented in a BiCMOS process. The clock period is determined by an aggressive processor design target.

Figure 1 depicts the major blocks of the multiplier/divider unit, consisting of registers, carry-save adders and a carry-propagating adder. The latter can be configured using multiplexors to allow for its use in different operations. Operands are selected from four sources and are examined by a special circuitry that decodes leading zeros and special instances and characteristics of the operands like zero, ± 1, < 8196, etc. The operands may be negated and/or normalized, as deemed appropriate to the instruction being executed. Normalization of operands is required in SRT division and allows us to reduce the number of divide iterations. Leading zeros detection is used to minimize the number of multiply iterations. Indications of operand characteristics is provided to the control logic (not shown in the figure). The low 13 bits of the divisor (the B operand) are used to address the CAM to access a previously stored reciprocal, which is then routed to the multiplier.

The first stage of the multiplier

The multiplier comprises two stages. The first stage performs Booth's recoding, partial product generation, and reduction of the resulting nine partial products to two partial products. This stage can be configured to generate and accumulate four sets of nine 32-bit partial products in the case of a PACKED_MULL instruction (see Figure 2), or a single set of nine 80-bit partial products in the case of a 64 × 16-bit multipli-

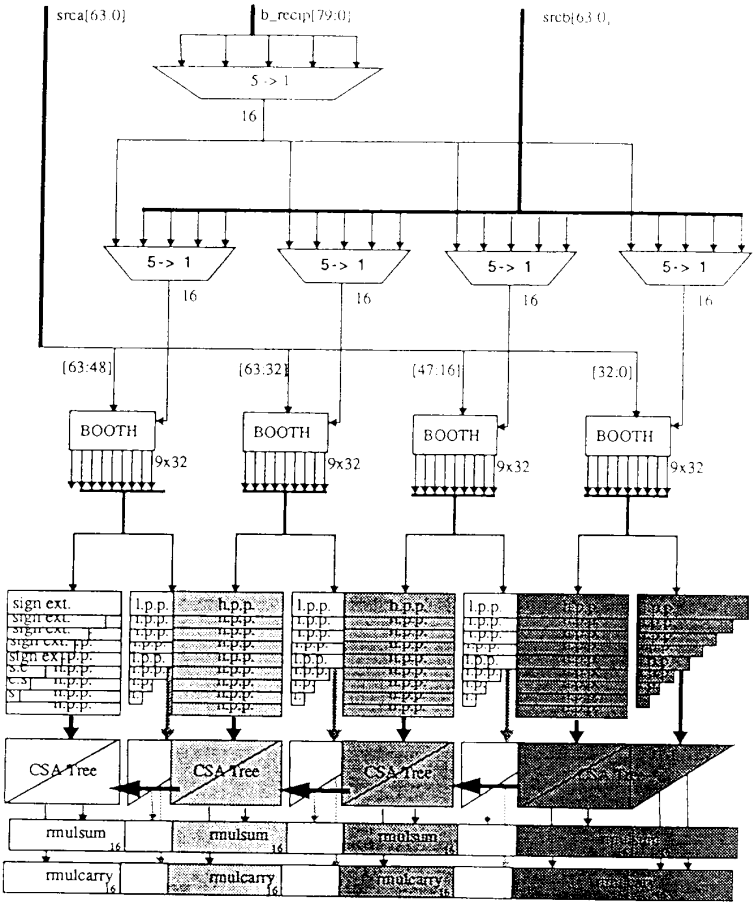

Figure 3: The partial product generation for the 64 × 16-bit multiplication.

cation (see Figure 3). The configuration is performed at the partial product generator and at the CSA reduction tree.

The configuration at the partial product generator is done as follows. The partial product generators are implemented with a multiplexor which selects one out of five multiples of the multiplicand, namely, -2, -1, 0, $+1$ or $+2$. The multiples are implemented by appropriate complements and/or shifts of the multiplicand. To generate a 32-bit partial product, the multiplicand provided to the partial product generator is a 16-bit value sign extended to 32-bits. To generate an 80-bit partial product, the multiplicand provided to the partial product generator is a 32-bit value comprised of two adjacent 16-bit segments of the 64-bit multiplicand operand. This causes some partial product bits to be generated redundantly by more than one partial product generator, and the redundant bits are ignored.

The configuration of the CSA is performed by a set of six 2-to-1 multiplexors at each of three locations. Figure 4 shows the details of the interconnections. In effect, the switches bypass the unnecessary sparse portions of the CSA array.

The second stage of the multiplier

The second stage of the multiplier (see Figure 5) performs the result accumulation. It contains an 80-bit adder with selectable inputs. The inputs correspond to various partial results of multiplication and division. One of the input pairs is the output of the CSA. Since the output of the first stage of the multiplier is in redundant sum and carry form, a $3 \rightarrow 2$ CSA is required to accumulate these outputs with the previous result of the adder.

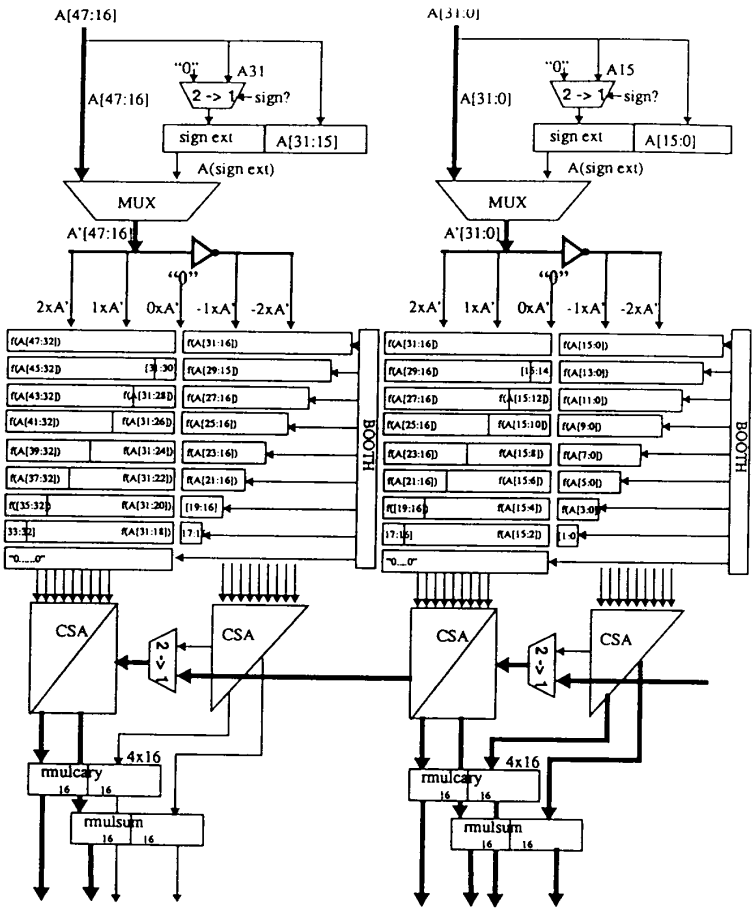

Figure 4: The CSA configuration for the 64 x 16-bit multiplication.

The radix-4 divider

The radix-4 divider (Figure 6) performs SRT division where multiples $(\pm 2, \pm 1, 0)$ of the divisor are iteratively subtracted from the partial remainder. The partial remainder is generated by a $3 \rightarrow 2$ CSA and maintained in redundant sum and carry form. The implementation details of this well-known algorithm (see for example, [1], [2], [3]) are not described here. We only mention that, in keeping with the approach to share circuitry whenever possible, the second stage of the multiplier is used to accumulate the division quotient, perform the final sum of the remainder, and negate the divisor and final quotient if necessary. In addition, the multiplier's first and second stages are used to perform the steps of multiplication, subtraction, and increment when executing the fast division and remainder operations, as well as the reciprocal generation itself.

5 Cost and Performance

The performance of the selected design was analyzed and compared to the expected performance of alternative designs. The execution clock counts were calculated for three different designs using a typical application code. These designs include: (I). A straight-forward radix-4 divider and standard iterative multiplier summing 32-bit partial products. (II). The selected design. (III). Same as II, but with additional "caching" of dividends and remainders to support a fast look-up Remainder operation. The results are shown in Table 1.

Table 1 also includes the cost associated with the three designs in terms of area and number of transistors. The additional cost of the selected implementa-

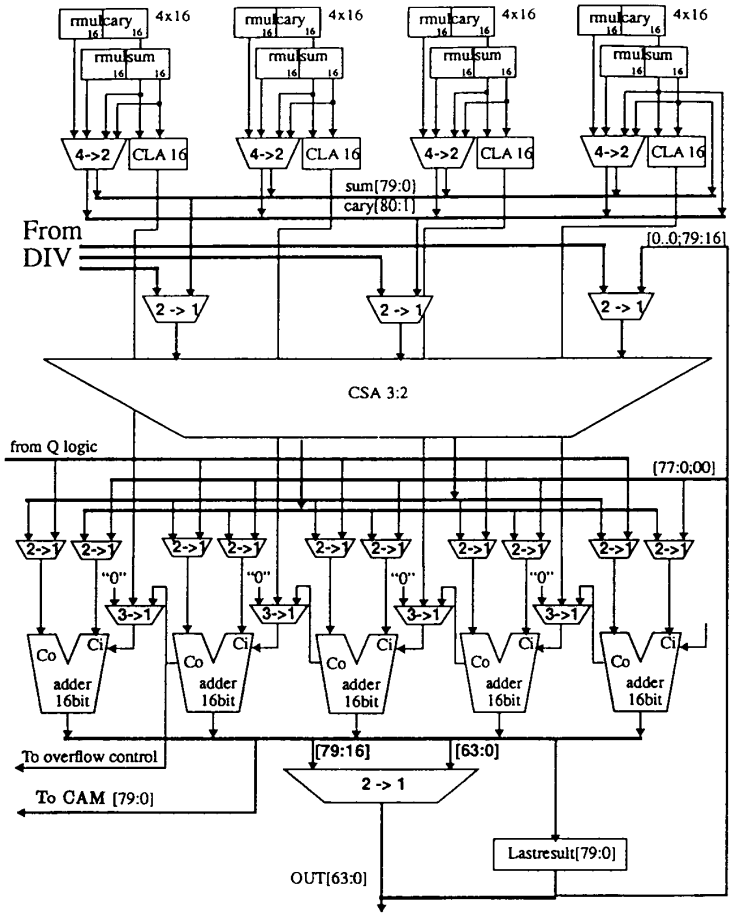

Figure 5: The second stage of the multiplier.

tion (II) stems mostly from the configurability feature of the CSA array. The addition of the CAM has a relatively small impact due to the high density of the memory cells.

6 Conclusions

The design of an integer multiplier/divider unit has been presented in this paper. The divide (and remainder) operation is accelerated through the use of a small associative memory to store the reciprocals of a few divisors. This is an economical solution since almost only the most likely to be used reciprocals are stored by using a cache-type approach. This can become a viable approach in designs of arithmetic units with limited transistor budgets.

Performance optimization can come about from increased cooperation between the designer and the user/customer. Such cooperation helps the designer in understanding the problem, refining the goals of the design and achieving a satisfactory tradeoff between the application requirements and the implementations constraints.

References

[1] J. Fandrianto, "Algorithm for high speed shared radix 4 division and radix 4 square root," *Proc. 8th Symp. on Comp. Arith.*, pp. 73-79, 1987.

[2] I. Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[3] G. S. Taylor, "Compatible hardware for division and square root," *Proc. 5th Symp. on Comp. Arith.*, pp. 127-134, 1981.

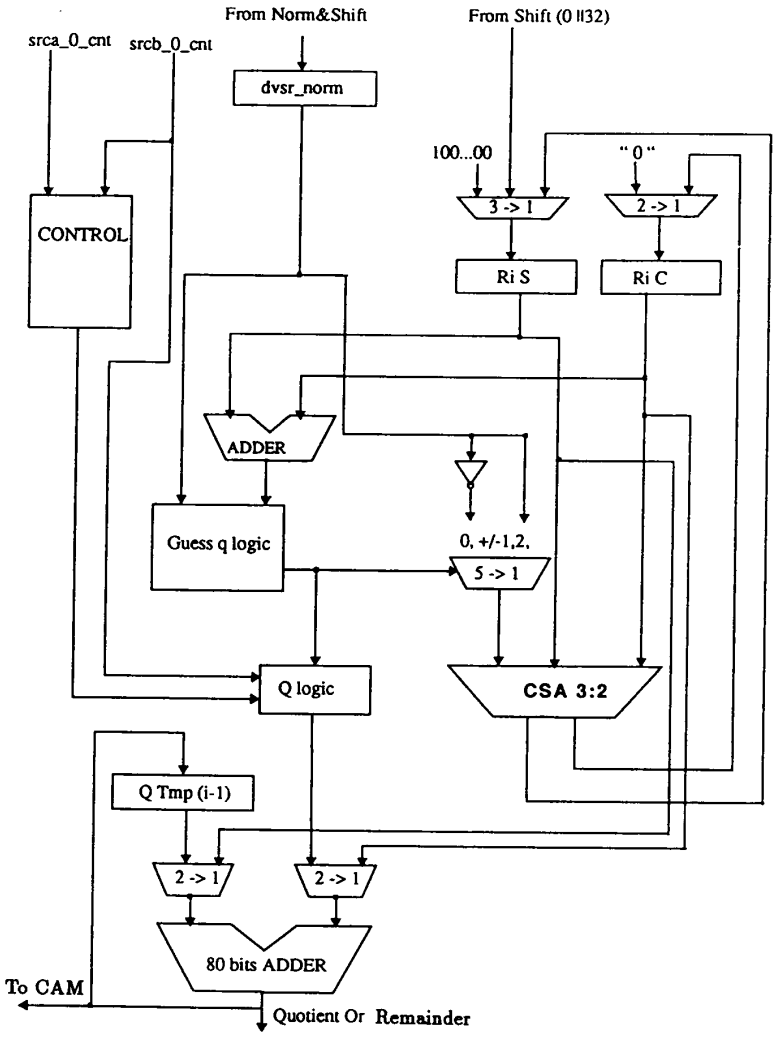

Figure 6: The radix-4 divider.

Design Alternative	MUL	DIV (CLKS)	REM	Normalized Performance	Normalized Area	Number of Transistors
I	10	24	26	1.00	1.00	115,000
II	6	10	12	0.72	1.05	130,000
III	6	10	8	0.71	1.07	140,000

Table 1: Performance and cost comparison.