CMOS floating-point unit for the S/390 Parallel Enterprise Server G4

by E. M. Schwarz
L. Sigal
T. J. McPherson

The S/390® floating-point unit (FPU) on the fourth-generation (G4) CMOS microprocessor chip has been implemented in a CMOS technology with a 0.20-μm effective channel length and has been demonstrated at more than 400 MHz. The microprocessor chip is 17.35 by 17.30 mm in size, and one copy of the FPU including the dataflow and control flow but not including the FPR register file is 5.3 by 4.7 mm in size. There are two copies on the chip for error-detection purposes only; both copies execute the same instruction stream and are checked against each other. The high-performance implementation has a throughput of one instruction per cycle and an average latency of three execution cycles, yielding approximately 70 MFLOPS at 300 MHz on the Linpack benchmark. Currently, the G4 FPU is the highest-performance S/390 CMOS FPU with fault tolerance. It uses several innovative and high-performance algorithms not commonly found in S/390 FPUs or other FPUs, such as a radix-8 Booth multiplier, a Goldschmidt division and square-root algorithm, techniques for updating the exponent in parallel with normalization, and avoidance of the remainder comparison in quadratically converging division and square-root algorithms. Also demonstrated is a practical design technique for designing control flow into the dataflow and early floorplanning techniques.

Introduction

The IBM S/390* floating-point architecture is an extension of the well-known System/360* architecture from the 1960s [1]. The floating-point format has a hexadecimal exponent with a 7-bit characteristic biased by 64 and a 1-bit sign, as indicated by the following:

$$X = (-1)^{X_s} * 16^{(X_c - 64)} * X_f, \qquad 0.0 \le X_f < 1.0,$$

where X_s is the sign bit, X_c is the characteristic, and X_f is the fraction or mantissa. Extended format was added in the 1970s along with square-root operation, which started out as a mathematical assist until it was recently included in the base architecture. The short format has a fraction of 24 bits, the long format has one of 56 bits, and the extended format has one of 112 bits. The G4 FPU is optimized for long format but also supports the other formats. The short format requires use of a trivial set of masking functions to implement on the long-format dataflow. The extended-format data are partitioned into two long-format numbers per operand, which requires several passes through the FPU to execute. The extended-format operations and high-order arithmetic operations

0018-8646/97/$5.00 © 1997 IBM

such as division and square root operate in a nonpipelined mode. They require many suboperations to complete and internally are highly pipelined but are not pipelined at the instruction level. The most common operations such as load, addition, and multiplication are pipelined and can execute one every cycle. The dataflow has been optimized for addition and multiplication of long-format operands, and very little hardware has been added to support nonpipelined instructions.

The G4 FPU is also responsible for performing fixed-point multiplication and fixed-point division. The FPU has a very fast multiplier which is capable of supporting two's-complement numbers for fixed-point calculations and sign-magnitude numbers for floating-point calculations. The fixed-point arithmetic operations are executed in a nonpipelined mode, which simplifies the data dependency analysis between instructions. The G4 microprocessor issues one instruction per cycle in order and completes at most one instruction per cycle in order. Thus, executing fixed-point multiply in nonpipelined mode does not cause much performance degradation, since most fixed-point instructions require only one execution cycle, and a fixed-point instruction stream would have to wait for the multiply result even if pipelined. This would result in a larger performance degradation in an out-of-order-completion machine. Fixed-point division is also executed in the FPU and uses an algorithm similar to floating-point divide short. However, there is the additional complexity in producing a remainder and conditionally complementing the input operands and output operands.

Also, the fixed-point unit (FXU) performs some functions normally thought to reside within the FPU. The FXU aligns input data from memory for floating-point instructions which are in RX format (register-and-indexed-storage operations). Data in memory are byte-addressable, and floating-point data can be 4 or 8 bytes, which are not necessarily aligned to a cache-line boundary. The cache returns the doublewords of memory containing the data and does not separate and rotate the data for the functional units. The operand buffers in the FXU provide this service for both the FXU and the FPU. The FXU also performs floating-point stores for the FPU, and that activity can require storage alignment, data masking, and multiple data writes that cross doubleword boundaries. The performance penalty of the FXU performing the floating-point stores is zero cycles for non-data-dependent stores, but two cycles for dependent stores.

The G4 FPU executes the most common floating-point instructions in a pipelined fashion with a throughput of one per cycle, and the infrequent operations are executed in a nonpipelined mode. The dataflow is described in detail, along with the execution of each type of instruction. In addition, the overall control flow is presented, followed by circuit implementation, physical design, discussion of designing control flow into the dataflow, and early floorplanning techniques.

Dataflow and execution

The fraction dataflow, shown in **Figure 1**, consists of a long-format multiplication and addition dataflow with a common 120-bit carry-propagate adder. At the top of the figure are the buses into and out of the FPU. The FPU_A_BUS and FPU_B_BUS are 64 bits each and bring operand 1 and operand 2 into the FPU from the FXU. Operand 1 is from the FPRs for floating-point computation and from the GPRs for fixed-point computation. Operand 2 is from the FPRs, the GPRs, or the operand buffers for memory operands. The operands are latched into the FPU A and B registers, which have fraction, exponent, and sign portions. Only the fraction part of the internal dataflow is shown in the figure. The output bus for the floating-point unit is the FPU_C_BUS, which is 64 bits wide and is driven to the register files. Internally there are eight additional bits of precision for intermediate calculations in the division and square-root routines. The FPU_C_BUS drives dependent data back into the A and B registers and into the first cycle of execution through the three late multiplexors. One other bus at the top of the dataflow is the output from the divide and square-root lookup tables. This bus is only 10 bits wide. All of these buses drive data to the A and B registers. The reading of operands from the register files or memory into these latches is defined to be the "E0" cycle. This is the cycle prior to the first execution cycle. Note that the A and B registers have multiplexors on their input which are capable of masking data for short-format instructions.

In the first execution cycle the A and B registers drive data into the late multiplexors. The term "late multiplexor" is actually a misnomer, since these multiplexors are located early in the first execution cycle, but can provide late-arriving interlocked data from the FPU_C_BUS. There is a late multiplexor for the multiply A operand (MAL), the adder A operand (AAL), and the B operand for both multiply and add (BL). The multiply A operand can be 64 bits to support extra precision for division and square-root intermediate calculations. The other late multiplexors are 56 bits to support long format.

The MAL and BL multiplexors drive the multiply first cycle, which consists of a 3$\times$ adder and Booth decode; for addition, the AAL and BL multiplexors drive the compare and swap and aligner and XOR logic. The MAL and BL multiplexors also drive binary shifters which are capable of binary-aligning data or forcing binary shifts of up to 3 bits right or left. These shifters are used for division and square-root operations, and their output is latched back into the A and B registers. The output of the multiply first cycle is latched in the 3$\times$ and $\times$ registers and the Booth

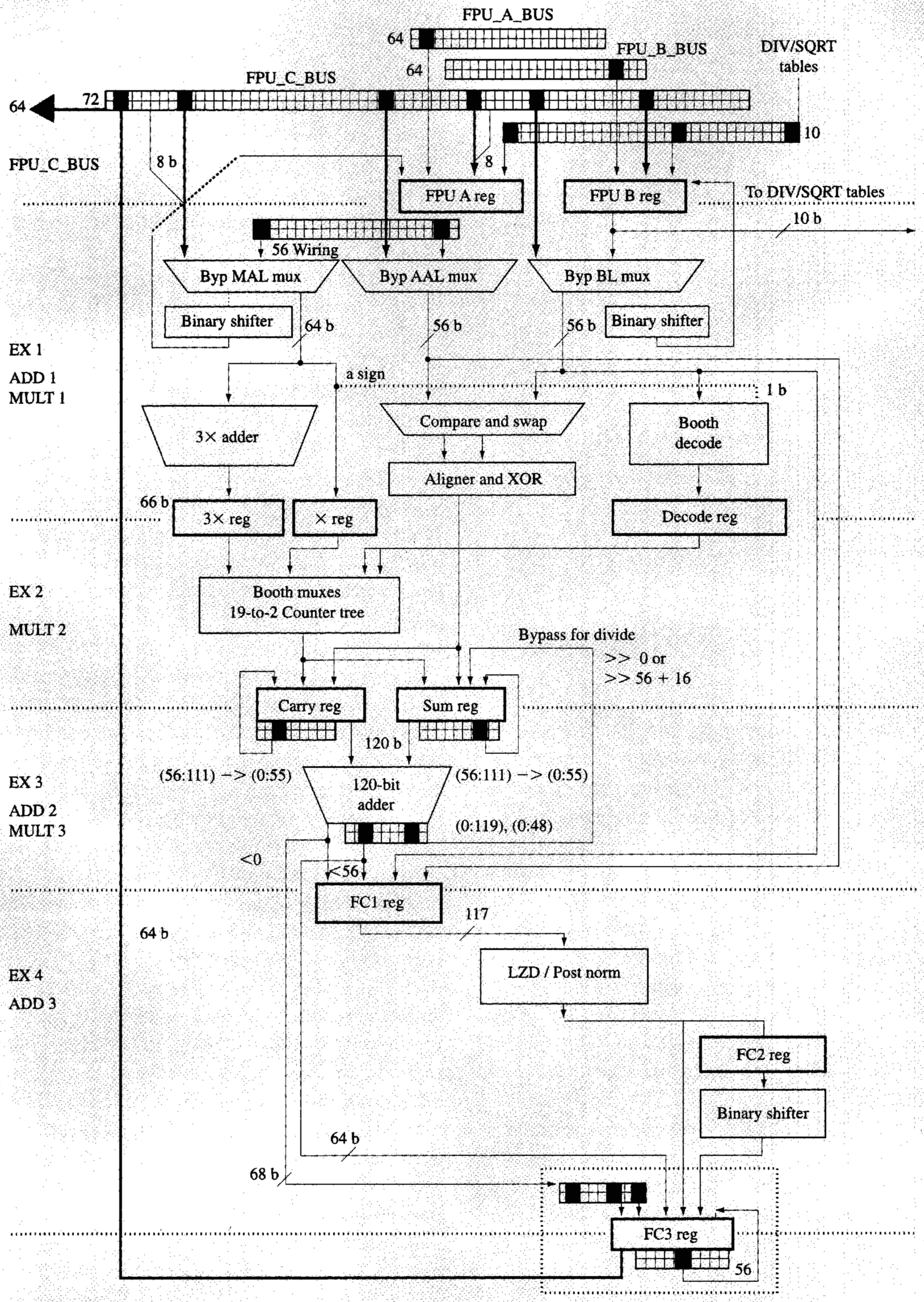

Figure 1

Fraction dataflow.

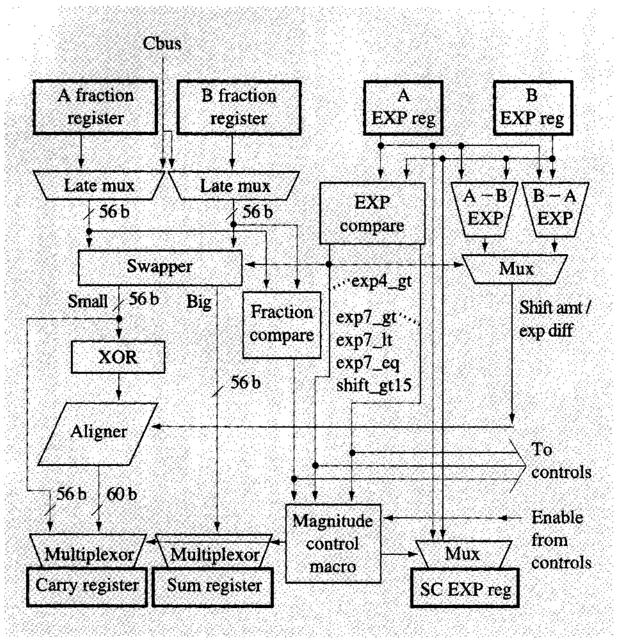

Figure 2

Dataflow of the first addition cycle.

decode registers. The output of the addition first cycle is latched in the carry and sum registers.

The output of the multiply's 3× register, × register, and Booth decode registers feeds a Booth multiplexor and a 19-to-2 counter tree resulting in two partial products. These two partial results are 120 bits each and are latched into the 120-bit carry and sum registers.

The second cycle of an addition and third cycle of a multiplication pass through the 120-bit adder. There are several feedback paths shown which are used for extended-precision operations and for division and square root. The result of the adder can be driven to either the FC1 register or the FC3 register (for the case of a multiply).

The FC1 register drives 117 bits to the post-normalizer for the third addition execution cycle. The post-normalizer determines the shift amount by performing a leading-zero detect (LZD) of the fraction, and then the fraction is shifted and the exponent is updated in parallel. The normalizer output is driven to both the FC2 and FC3 registers.

The FC2 register provides an extra internal working register for division and square root. It also drives to a binary shifter, which is used to transform binary-aligned intermediate results for division and square root into hex-aligned results. The binary shifter can shift the fraction up to 3 bits left or right and is controlled by an LZD of the most significant digit or by a forced shift amount from controls. The binary shifter output is connected to the FC3 register.

The FC3 register receives the result of the arithmetic computation and drives the results to register files or back to the FPU dataflow by way of the FPU_C_BUS. The FC3 register also has the ability to shift itself by 56 bits to the left so that extended data can be stored in the FC3 register and written to the FPU_C_BUS with two back-to-back write cycles without involving the other elements of the FPU fraction dataflow.

The overall FPU fraction dataflow has five stages, though the most common operations require only three cycles or stages of execution. The execution of addition, multiplication, load, division, square-root, and extended-precision instructions is detailed in the following subsections.

• *Addition*

The operations subtract, add, or compare are collectively referred to as an addition. Floating-point addition involves aligning the fractions of the operands, conditional complementation of the smaller operand, a two's-complement addition, normalization, and condition code setting. This usually requires three cycles of execution. The key to subtraction of sign-magnitude numbers is identifying and complementing the smaller of the two operands prior to the carry-propagate addition. The resulting sum is a magnitude and does not require conditional post-complementation. As shown in **Figure 2**, the first cycle consists of comparing the A and B register exponents to determine which operand is the smaller of the two, and conditionally swapping the fractions so that the smaller operand proceeds to be conditionally complemented and aligned by the exponent difference. One additional hex guard digit (4 bits) is maintained during alignment, as specified by ESA/390* floating-point architecture. Then the larger operand is placed in the sum register and the aligned operand is placed in the carry register. The second cycle involves a two's-complement addition of the fractions. The 28 fraction bits of short operands or 60 fraction bits of long operands are easily accommodated by the 120-bit adder. The sum is latched in the FC1 register. The third cycle of execution involves a post-normalization.

Post-normalization of the sum is not always necessary, but it is rather frequent for S/390 architecture. In all architectures there could be cancellation of the most significant bits for an effective subtract operation which changes the location of the most significant bit. There could also be a carry-out for an effective addition operation. But for S/390, there is the additional possibility of unnormalized fractions or the case of a zero fraction with a nonzero characteristic. Other architectures have denormalized numbers, but the occurrence is less frequent and their range of values is limited. So, for simplicity all additions are routed through the post-normalizer.

In the post-normalization cycle, a leading-zero detect is performed on the fraction to determine the shift amount. The most significant bits of the shift amount are available earlier than the least significant bits, and the exponent is updated in parallel with the delayed arrival of the least significant bits [2]. The determination of exponent underflow and overflow is also calculated in parallel with the fraction normalization. The resulting normalized fraction and exponent are latched in the FC3 register at the end of the third cycle of execution. During the following cycle the result is written into the FPR.

Of these three execution cycles, the first addition cycle is the most complex; Figure 2 describes the interconnection of the dataflow. Since this cycle is very timing-critical, both the dataflow and the control design were specified with custom circuits. The cycle begins with the A and B exponent registers being driven to the exponent compare circuit and to the exponent difference logic. A signal called $EXP4\_GT$, which is a compare of A exponent greater than B for the least significant four bits of exponent, is driven to the swapper. It actually uses five bits of exponent for the comparison, as shown by the following equation:

$$EXP4\_GT = (EA_{4lsb} > EB_{4lsb}) \oplus (EA_5 \oplus EB_5).$$

If the fifth-to-least significant bits (EX_5) of the two exponents are not equal, the comparison of the least significant four bits ($EA_{4lsb} > EB_{4lsb}$) is inverted. Thus, $EXP4\_GT$ is an approximation of which operand is greater. The operand guessed to be larger is driven on the Big bus, and the smaller operand is driven on the Small bus. Feeding the swapper in the fraction dataflow are the late multiplexors, which can receive data from the FPU_C_BUS or the fraction registers. The Small bus drives an XOR circuit which conditionally complements the data for an effective subtract operation where the shift amount is less than 16. The XOR output drives an aligner which can shift right 0 to 15 digits. The shift amount is determined by two subtractors. Both exponents A minus B and B minus A are calculated, and then the appropriate result is chosen once $EXP4\_GT$ is known. The output of the aligner drives to the carry multiplexor/register. The Big signal is driven directly to the sum multiplexor/register. The selection of carry and sum multiplexors is determined by a custom control macro called the A1 magnitude control macro. It determines whether the swap was correct and whether to force zeros for shifts greater than the width of the fractions or to mask the operands for short data. In addition to the fraction being latched, the larger of the two exponents is latched into the sum and carry exponent registers (SC EXP reg).

This is the general procedure for most cases of addition execution, although the procedure is actually more complex and can be separated into five cases: effective load, effective add, one's-complement, two's-complement, and simple subtract. The details for each case are described below.

Effective load: ($SHIFT\_GT15$)
If there is an exponent difference greater than 15 (signaled by $SHIFT\_GT15$), the operation is effectively a load. There are 14 hex digits in a long operand, and S/390 dictates that one additional guard digit be maintained, for a total of 15 hex digits. Note that short operands were treated the same as longs, but with additional masking to allow only seven hex digits of precision. Since the rounding mode of S/390 hex floating-point is truncation with the exception of square root, the smaller operand does not contribute to the addition if the exponent difference is greater than 15.

The execution of this case of addition could be accomplished in two execution cycles. However, it was designed to be completed in three cycles to avoid creating any critical control paths. So, for simplicity this case consumes an adder cycle. To accomplish this, the operand with the larger exponent and zeros are gated into the sum and carry registers. There is a complication in doing this, since the swapper does not use an exact exponent greater than the compare signal but instead uses a signal based on the least significant four bits. Thus, there is the potential for the swap to be incorrect (i.e., the Big signal is actually the smaller of the two operands, and Small is the bigger operand). This must be taken into account in the A1 magnitude control macro when selecting the larger operand and zeros to be multiplexed into the two registers.

This case executes in three cycles, and in the first cycle, the operand with the greater exponent and zero is gated into the sum and carry registers.

Effective add: ($\overline{SHIFT\_GT15} \cdot \overline{EFF\_SUB}$)
Another simple case is an effective add operation (signaled by $\overline{EFF\_SUB}$) which exists when the operation is add and the operands have the same sign or the operation is subtract or compare and the operands have different signs. For this case, no complementation is necessary. Since the shift amount is less than 16, the least significant bits of the exponent are guaranteed to give the proper shift amount, and this 4-bit exponent difference is driven to the aligner. The aligner output and the Big bus are driven to the carry and sum registers, respectively. This case requires three cycles, since the normalization cycle may be required if the operand with the larger exponent is unnormalized.

Conditional one's-complement:
$(\overline{SHIFT\_GT15} \cdot EFF\_SUB \cdot EXP\_EQ)$
The remaining cases are more difficult and involve an effective subtraction with shift amount less than 16. Determination of the operand to conditionally complement and align is difficult, especially considering that the input fractions could be unnormalized. The case in which the exponents are equal (signaled by *EXP_EQ*) is called conditional one's-complement. For timing reasons, the fraction comparator receives the unaligned operands. When the exponents are equal, the fraction comparison gives a true indication of the relative magnitude of the two operands, but it is determined too late to complement the smaller of the two before addition. However, this is not necessary to obtain the correct result. In [3], a method is described of always complementing operand B and still computing the correct result. Note, for $A - B$,

$$|R| = |A| - |B| = |A| + \overline{|B|} + 1 \text{ ulp},$$

where $\overline{|B|}$ is the one's-complement of B and ulp refers to a unit in the last place of operand B. For $B - A$ the following can be derived [4]:

$$|R| = |B| - |A| = \overline{(|A| + \overline{|B|} + 0 \text{ ulp})}.$$

Thus, B can always be complemented, if for $B < A$ the carry-in to the adder is set (1 ulp) and the true sum is the result, and for $A < B$ the carry-in is zero and the complement of the sum is the result. This causes the critical path to be the setting of one bit, the carry-in to the adder, rather than requiring conditionally complementing all the fraction bits of Small and Big and conditionally selecting them to the carry and sum registers. The control signal for the FC1 register which receives the adder output can be determined to be the true or the complemented output.

Thus, this case requires three cycles of execution: the first cycle, in which B is complemented and the carry-in to the adder conditionally set based on the fraction greater than signal, the second cycle of a 2-to-1 addition and conditional selection of the true or complemented output into the FC1 register, and the third cycle of post-normalization.

Conditional two's-complement:
$(\overline{SHIFT\_GT15} \cdot EFF\_SUB \cdot EXP\_EQ \cdot \overline{UNNORM})$
When the exponents are not equal and the data are unnormalized (signaled by *UNNORM*), this is called the conditional two's-complement. For this case, a prior determination of the larger operand is not possible, so a post-adder determination is made. Four cycles of execution are needed; two cycles use the adder. The second cycle through the adder is used to create a two's-complement of the sum. The first time through the adder, the carry-out can be latched along with the guessed true sum, and in the following cycle the carry-out can be used as a select signal either to hold the true sum if the original guess of the greater operand is correct, or to select the complemented sum to be gated into the FC1 register. Thus, there is a pipeline stall for this case. It was estimated that this case occurs only about 3 percent of the time.

Simple subtract:
$(\overline{SHIFT\_GT15} \cdot EFF\_SUB \cdot \overline{EXP\_EQ} \cdot \overline{UNNORM})$
If the input operands are both normalized and the shift is less than 16, the exponent compare of the least significant four bits truly indicates the smaller of the two operands. Thus, the correct operand can be identified and complemented in the first cycle of execution. This is called the simple subtract case, and requires only three cycles of execution.

Thus, in summary for addition, there exists only one rare case, conditional two's-complement, which requires four execution cycles; all of the other cases are completed in three execution cycles. The adder dataflow has been optimized to be pipelined one instruction per cycle, and each cycle has been optimized to meet cycle time. The most complex of these cycles is the first add cycle, which makes many of its decisions based on only four or five bits of the exponents. This enables a very fast cycle time, but at the cost of complexity of design, which is evident in the five separate cases of control signal selection. To reduce timing, the control design was implemented in custom circuits; it was designed into the dataflow early in the design phase.

- *Multiplication*

Both fixed-point and floating-point multiplication are performed on the FPU's multiplier. In addition, the multiplier is used by the division and square-root routines. These high-order routines require greater precision for intermediate results than is supported by the long format. Thus, additional bits of precision are necessary, but the critical timing of long format for a 56-by-56-bit multiplication must not be exceeded. To accomplish this, only one operand is extended with additional bits to be 64 bits. The other operand dictates the cycle time, since it specifies the number of partial products which determine the number of stages in the counter tree. Other designs [4] have increased both operands, creating a 60-by-58-bit multiplier which increases the performance of division and square root. On the G4 microprocessor chip, though, this type of implementation would create a longer cycle time, affecting the performance of all instructions. Thus, a 56-by-64-bit multiplier was implemented in the G4 FPU.

The counter tree design in the FPU is an important part of the FPU design; it is timing-critical and it is very large, consuming 15 percent of the G4 FPU. Our design goal for the multiplier was to have it pipelined every cycle and have an approximate latency of two or three cycles. One full cycle is required for 2-to-1 addition, so the counter tree had to complete in one or two cycles. Radix-4 Booth algorithms [5, 6] are interesting because of their simplicity, but for a 56-bit operand they require a 29-to-2 partial product array, and Booth multiplexing requires a 2-to-1 true/complement multiplexor. This counter tree requires eight levels of 3/2 counters, which did not meet our cycle time objective. Another option is to implement the counter tree with latches in the middle, but this is prohibitive to implement because of the large number of signals that must be latched.

Two other alternatives are to create a two-cycle path that is unlatched or to perform an iterative multiply. The first option was rejected by our test personnel, who did not like the large number of paths that had to be specially tested for cycle time. This would have added a huge amount of time to test the chip, and was determined to be too costly. The other alternative is an iterative multiply [7] in which half the multiplication is computed each iteration, e.g. by using a 28-by-64-bit multiplier. This has a very small area and very fast critical path, but it does not have the performance benefit of being able to pipeline a multiply every cycle.

The solution that has been implemented in the G4 chip is a higher-radix Booth algorithm. In particular, a radix-8 Booth algorithm was used to simplify the counter tree [8, 9]. The counter tree for a radix-8 algorithm requires only a 19-to-2 partial product reduction which has six levels of 3/2 counters. The counters in 0.2-μm technology supporting a 300-MHz clock frequency have a delay of approximately 350 ps for the sum output and 250 ps for the carry output, for an average delay of 300 ps. The Booth multiplexing requires a 4-to-1 true/complement multiplexor which has a delay of approximately 450 ps. The delay of both the counter tree and the Booth multiplexing meets the cycle-time objective. The main problem with higher-radix algorithms, though, is creating the difficult multiples of the multiplicand which are not powers of 2. For a radix-8 algorithm, the multiples of $+4$, $+3$, $+2$, $+1$, 0, -1, -2, -3, and -4 of the multiplicand are required. The only difficult multiples to form are the $\pm 3\times$. In the first cycle of execution, the $3\times$ multiple is formed, and the Booth selects for the multiplexing are determined. In the second cycle, the Booth multiplexing between all of the possible multiples ($+4$ to -4) is performed using 4-to-1 true/complement multiplexors; then, 19 partial products are reduced to two in six levels of 3/2 counters. This cycle is shown in **Figure 3**. In the third cycle, the 2-to-1 addition is performed. A radix-8 Booth algorithm provides the best partitioning for our particular implementation. The detailed implementation of the counter tree is described in [10].

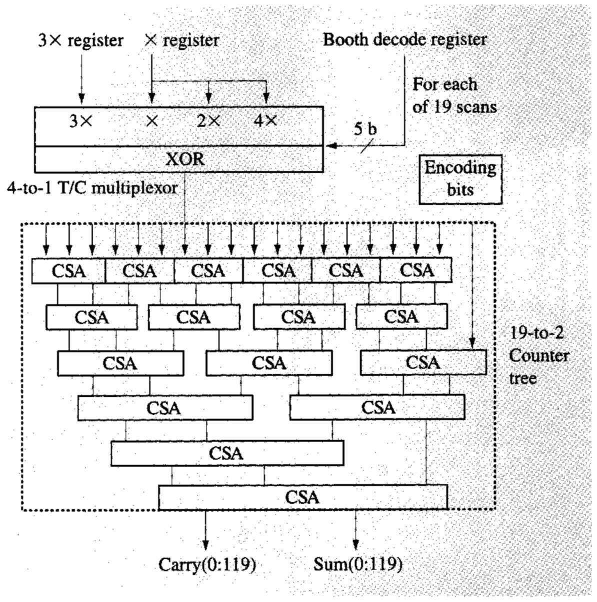

Figure 3

Dataflow of the second multiplication cycle.

The common cases for post-normalization have been designed into the dataflow of the third cycle of execution. From instruction traces, it was determined that more than 90 percent of the time both operands are normalized. If both operands are normalized, it can easily be shown that the result could have one of two normalizations. Multiplying the minimum normalized numbers and the maximum normalized numbers gives the following range of products:

$$(0.1)_{16} * (0.1)_{16} = (0.01)_{16}$$

and

$$(0.FFF\cdots)_{16} * (0.FFF\cdots)_{16} < (0.FFF\cdots)_{16}.$$

Thus, the minimum product requires a left shift of one hex digit (4 bits), and the maximum product has a shift of zero, or no shift. The determination of which shift is required is designed into the 120-bit adder to execute in parallel with the determination of the conditional sum of the most significant hex digit. An enable signal is sent to this logic to allow it to drive the select signals to the FC3 register. Both possible combinations of fraction shifts are driven to the FC3 register and are selected by this control signal, which was designed into the dataflow in custom circuits. Thus, the critical control paths were designed in custom logic. Note that if the operands are unnormalized or if the exponents are close to underflowing or overflowing, the multiply instruction requires four cycles and is driven into the post-normalizer. Hence, the most common case has a latency of three cycles with a throughput of one instruction per cycle, but some cases have a latency of four cycles with a throughput of one per cycle.

- *Load*

Loads are executed in the floating-point pipeline to take advantage of the fast bypassing between data-dependent instructions. The actual execution involves two cycles. First the operand in the B register is moved directly to the FC1 register. The second cycle passes through the normalizer with a forced shift amount (no normalization is allowed by the architecture) and is latched into the FC3 register. Thus, loads require two execution cycles.

- *Division*

The division implementation uses the Goldschmidt algorithm, which has been described in many conferences and papers [4–6, 11–16]. It is a very interesting algorithm for high performance because of its quadratic convergence and its ability to execute some of its multiplications in parallel. The algorithm was first used in the IBM System/360* Model 91 [17], but since then has not been implemented on S/390 mainframes, except for one low-end mainframe [4]. The main limitation of this algorithm is that it requires nontrivial error analysis and the avoidance of extra cycles to round the result. Other algorithms such as nonrestoring algorithms or even the Newton–Raphson quadratically converging algorithm are much easier to analyze, since they are self-correcting. Analyzing the error in one iteration and the error in the lookup table is enough to prove these algorithms for the *N*th iteration. The Goldschmidt algorithm has error which propagates each iteration, and all iterations must be analyzed.

The algorithm is based on converging the divisor (denominator) to one; then, the dividend (numerator) is equal to the quotient. Let Q equal the quotient, and the dividend of the ith iteration is N_i, the divisor D_i, and the convergence factor R_i. The following can be stated:

$$\begin{aligned} Q &= \frac{N_0}{D_0} \\ &= \frac{N_0}{D_0}\frac{R_0}{R_0} \\ &= \frac{N_0}{D_0}\frac{R_0}{R_0}\cdots\frac{R_{i-1}}{R_{i-1}} \qquad \text{if } D_i \approx 1.0, \\ &= \frac{N_i}{D_i}; \qquad \text{then } Q \approx \frac{N_i}{1.0} = N_i\,. \end{aligned}$$

The initial convergence factor is determined from a lookup table, but the subsequent convergence factors are determined by two's-complementing the current denominator:

$$\begin{aligned} R_0 &\approx \frac{1}{D_0} \\ D_1 &= D_0 * R_0 \\ &= D_0 * \left(\frac{1}{D_0} + E_{R0}\right) \\ &= 1 + D_0 E_{R0}; \end{aligned}$$

Let

$$\begin{aligned} X_1 &= -D_0 E_{R0}, \\ D_1 &= 1 - X_1, \\ R_1 &= 2 - D_1 = 2 - (1 - X_1) = 1 + X_1; \end{aligned}$$

Then,

$$\begin{aligned} D_i &= D_{i-1} * R_{i-1}, \\ N_i &= N_{i-1} * R_{i-1}, \\ R_i &= 2 - D_i. \end{aligned}$$

The error analysis was performed by 1) expanding the equations for four iterations, where RC_i is the calculated convergence factor in the ith iteration including error terms, DC_i is the calculated divisor, NC_i is the calculated dividend, $t_{\text{even}\,i}$ is the truncation error in the divisor calculation, $t_{\text{odd}\,i}$ is the truncation error in dividend calculation, E_{R0} is the error in the lookup table, and ts_i is the small error in the convergence factor truncation; and 2) performing a one's-complementation instead of a two's-complementation:

$$\begin{aligned} RC_0 &= 1/D + E_{R0}, \\ DC_0 &= D, \\ NC_0 &= N, \\ DC_i &= DC_{i-1} * RC_{i-1} - t_{2*i}, \\ NC_i &= NC_{i-1} * RC_{i-1} - t_{2*i-1}, \\ RC_i &= 2 - DC_i - ts_i. \end{aligned}$$

The analysis of the error in the final quotient, NC_4, was arduous. After substituting the values of the errors due to truncation, it was possible to prove that the implementation satisfied the error constraints.

The timing diagram of the iteration cycles of a Goldschmidt division algorithm is shown in **Table 1** for long operands. Short operands require only three iterations. In the G4 implementation there are an additional four cycles on the front end of the diagram to get the operands binary-normalized and in the proper

Table 1 Timing diagram for long divide.

Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
	R0	D1	D1	D1	D2	D2	D2	D3	D3	D3					
	N0		N1	N1	N1	N2	N2	N2	N3	N3	N3	PN4	PN4	PN4	N4

registers. And there are several cycles on the back end to hex-normalize and properly round the result.

Rounding poses an additional problem for quadratically converging algorithms, since a remainder is not determined as an intermediate product of the iteration step. The G4 implementation is able to eliminate the remainder comparison step half of the time by examining an additional bit of precision [17]. The intermediate result is formed with one additional guard bit with an error tolerance of less than the weight of the guard bit. For truncation which S/390 dictates, if the guard bit is 1, the result should be truncated. This is true since the error tolerance guarantees that the actual quotient is less than the next higher machine-representable number and not equal to it. But if the guard bit is 0, a remainder comparison is needed. If the remainder is greater than or equal to zero, the result is the truncated intermediate result; if the remainder is less than zero, the result is the decremented intermediate result. This eliminates the remainder comparison for half of the cases. If there are additional guard bits, this algorithm can be expanded to eliminate the remainder comparison in all but one of the 2^G cases, where G is the number of guard bits.

With the startup penalty of hex-normalizing and then binary-normalizing the operands and the ending penalty of hex-aligning the data, the overall division requires either 18 or 24 cycles for short operands and either 22 or 28 cycles for long operands for a guard bit equal to 1 or 0, respectively. The startup and ending penalties are very significant and make a quadratically converging algorithm only slightly better than a nonrestoring division algorithm. Thus, the G4 FPU chip implements a very aggressive division algorithm which is quadratically converging, and eliminates the remainder comparison in half of the cases.

• *Square root*

The square-root algorithm is also based on the Goldschmidt algorithm [18]. The following are the equations used, where r_i is the square root of the convergence factor in the ith iteration, SQr_i is the convergence factor, B_i is the accumulative approximation to the root of N, and X_i is an intermediate variable that approaches 1.

Initialize:

$$r_0 \approx 1/\sqrt{N},$$
$$B_0 = N,$$
$$X_0 = N;$$

iterate:

$$SQr_i = r_i * r_i,$$
$$B_{i+1} = B_i * r_i,$$
$$X_{i+1} = X_i * SQr_i,$$
$$r_{i+1} = 1 + 0.5 * \bar{X}^f_{i+1};$$

final:

$$\sqrt{N} \approx B_{\text{last}}.$$

The expression for the square root of the convergence factor, which does not take much delay to calculate, is one plus one half of the fractional part of X complemented. It is formed in the binary shifters located near the A and B fraction registers. The choice of the above formula for the square root of the convergence factor can be best understood by studying the convergence of X_{i+1} to 1.0. Let X_i be d_i different from 1.0; then,

$$\begin{aligned} X_i &= 1 - d_i, \\ r_i &\approx 1 + 1/2 * d_i \\ &\approx 1 + 1/2 * (1 - X_i) = 1 + 1/2 * (2 - X_i - 1) \\ &\approx 1 + 1/2 * (\bar{X}_i - 1) = 1 + 1/2 * \bar{X}^f_i, \\ SQ_i &\approx 1 + d_i + 1/4 * d_i^2 \approx 1 + d_i, \\ X_{i+1} &\approx 1 - d_i^2. \end{aligned}$$

Thus, there is quadratic convergence with this choice for the square root of the convergence factor.

The overall delay of the implementation is 26 or 32 cycles for short operands and 35 or 42 cycles for long operands, depending on the guard bit. This implementation eliminates the remainder calculation in half of the cases.

• *Extended-precision instructions*

Extended-precision instructions were also implemented in hardware, but in nonpipelined mode. Their performance is not critical, and thus uses simple cost-efficient algorithms.

Extended-precision addition is performed by examining the exponents and aligning the operand with the smaller

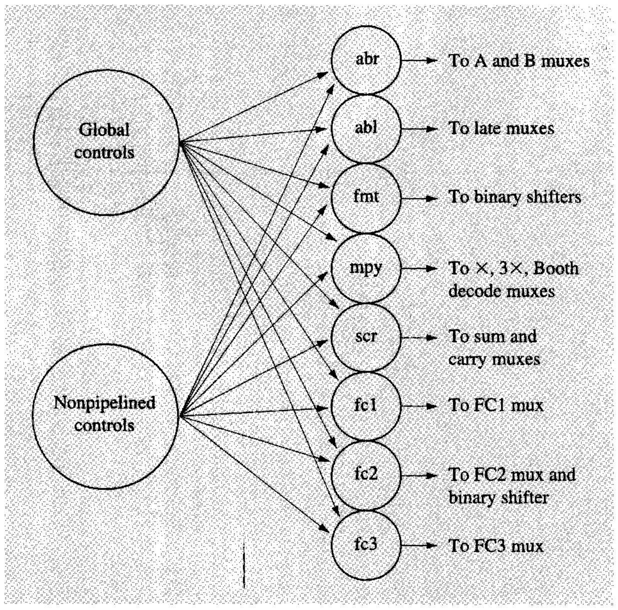

Figure 4

Overall control flow.

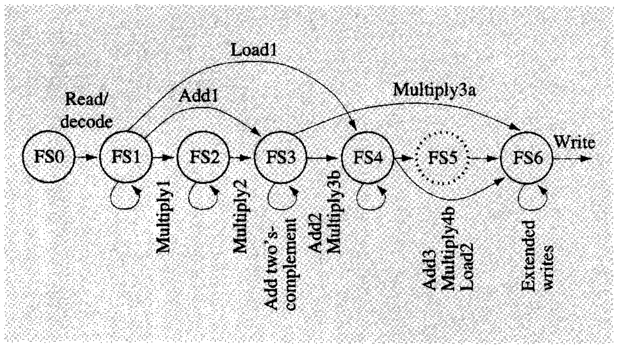

Figure 5

Pipeline state diagram.

exponent in the post-normalizer. The post-normalizer is 121 bits wide and has the capability of having the shift amount overridden by controls which then specify its own shift amount. Once data have been aligned, they are conditionally complemented and loaded in stages into the carry and sum registers. Then the actual addition requires one cycle. The result is then normalized and driven to the FC3 register, where it is written to the FPRs with two write cycles.

Extended-precision multiplication is performed by separating the extended inputs into two long fractions each. Four long multiplications and three additions are performed. Note that pre-normalization may be required if the operands are not already normalized.

Extended-precision division and square root use restoring 1-bit algorithms. Paths have been created into the carry and sum multiplexor/registers to support this. The latency is very long, but the amount of hardware invested is minimal.

Control flow

The control flow was designed with synthesized macros to allow changes late in the design phase for problems found in simulation. The overall control flow is shown in **Figure 4**. There are two major macros in the design: the global control macro and the nonpipelined control macro. The global control macro handles the state information for pipelined instructions and sequences the start of nonpipelined instructions. It also handles all handshaking with other units. The nonpipelined control macro handles the select lines for all nonpipelined instructions such as extended-precision floating-point, division, square-root, and fixed-point instructions. These routines are similar to horizontal millicode routines, since the macro implements a cycle counter and instruction decode which determine the selects to enable. The other control macros listed are collectively referred to as the pipeline-select macros; they determine the values on the select lines from global pipeline state information and from information from the nonpipelined control macro. This partitioning of controls makes it simpler to design, since the task of maintaining state information is separated from determining the select line values.

Global state information for pipelined instructions is maintained in the global control macro (see **Figure 5**). There are seven states: FS0, FS1, FS2, FS3, FS4, FS5, and FS6. FS0 corresponds to the E0 cycle, FS1 corresponds to the E1 cycle, FS2 corresponds to the E2 cycle of multiplication, FS3 corresponds to the 120-bit adder cycle, FS4 corresponds to the normalizer, FS5 is not maintained but corresponds to the shifter between the FC2 and FC3 registers, and FS6 corresponds to the write cycle. There are basically two methods for designing pipelines: up-front blocking and feed-forward. Up-front blocking prevents an instruction from entering the pipeline until it is guaranteed not to have any dependencies or resource conflicts. Feed-forward pushes the instruction as far into the pipeline as possible until contentions or dependencies make it wait. Since the G4 FPU can be considered a peripheral unit which is not aware of the global central processor state, it was best to implement a feed-forward pipeline so as not to become instruction-starved or data-starved.

In the global control macro there is a scoreboard-type implementation of the information for each state [19]. This information includes the write address, the length of the operands, whether the data must be normalized, the instruction type, etc. The status of each state is also maintained (e.g., whether the state is valid or whether it is busy and holding for this cycle). The state information from the global control macro is sent to pipeline-select macros, where decisions are made as to which selects to each multiplexor should be invoked. For example, if the FS3 state which corresponds to the 120-bit adder function is busy, the sum and carry registers should be held. This is determined by the sum and carry registers pipeline-select macro (SCR), which drives select lines to the multiplexor. Separating maintenance of the global state from determining values of select lines made the controls simpler to design.

Also, the global control macro is responsible for resolving resource conflicts. An example is a multiply instruction followed by an add instruction. Figure 5 shows that the multiply requires the FS1, FS2, and FS3 states, while the add requires the FS1, FS3, and FS4 states. There may be a contention for the FS3 state, which is the 120-bit addition cycle. In this case the add would be delayed, since the multiply was issued first. This conflict can be resolved by simply giving the transition from the FS2 to the FS3 state higher priority than the transition from the FS1 to the FS3 state. Note that this example shows a common resource conflict of using the adder for both multiplication and addition. The performance benefit of eliminating this conflict by having two adders was determined not to be worth the area cost.

- *Resolving data dependencies*

The most complex part of the control design is the resolution of data dependencies between instructions. These dependencies can be classified into four types, depending on timing and the buses used for bypassing (or wrapping) information: early wrap, late wrap, long-to-short wrap, and short-to-long wrap.

Early wrap This case has the best performance; it involves wrapping the exponents a cycle early and then wrapping the fraction into the E1, or execute-first, cycle as the data are being written to the register file. The exponent dataflow does not have late multiplexors in the E1 cycle, since they are timing-critical in the add-1 cycle. Instead, the exponent is wrapped back to the A and B exponent registers directly from the FS3, FS4, or FS6 state. This involves wrapping two separate exponents for a multiplication that is completing from the FS3 because of the different normalizations that are created on the fly. The following shows the relative timing of control signals and dataflow signals:

```
END OP        WR/E1
EXP WRAP    Fraction to late mux
```

END OP represents the end of the instruction handshake signal, EXP WRAP is the exponent wrapping, WR indicates the write cycle, and E1 the first cycle of execution. The wrapping takes two cycles.

Late wrap If the instruction that is the target of the bypass is not received early enough for the exponent to be wrapped early, the wrap is known as a late wrap. This involves wrapping the exponent and fraction during the write cycle to the A and B fraction and exponent registers. The following is the timing:

```
END OP    WR                    E1
        EXP/fraction to
         A and B registers
```

Long-to-short wrap For a long-to-short wrap, the timing is the same as for the late wrap. A longer data type is being written to the FPRs than has to be read from the FPRs. The low-order data must be masked, and there is no masking function on the late multiplexors. However, there is masking on the input to the A and B registers. Thus, the data must be wrapped to the input of the A and B registers. Another case of this wrap condition is creating a true zero (fraction, characteristic, and sign are set equal to zero) on a significance exception (result equal to zero). The true zeroing of the exponent for significance exception is performed in the FC3 exponent multiplexor. Thus, an early exponent wrap would have invalid data. The following is the timing:

```
END OP    WR                    E1
         EXP/Fraction to
         A and B registers
```

Short-to-long wrap For a short-to-long wrap, the data must be reread from the FPRs. There is no merge capability on the input to the A and B registers or in the late multiplexors. Thus, if data from a write must be combined with low-order data from the register file, a reread must take place. In addition to this case, exponent underflow, which results in a true zero, is detected very late and is bypassed in the same manner. The zeroing of the result for exponent underflow actually takes place in the C bus multiplexing of FPU_C_BUS and FXU_C_BUS prior to the register file. Thus, bypassing this exponent requires a reread of the register file; the timing is shown below:

```
END OP   WR   Read   E1
```

The early wrap is the most common of the wraps; it demonstrates an interesting design strategy. Methods of bypassing data can be divided into four categories:

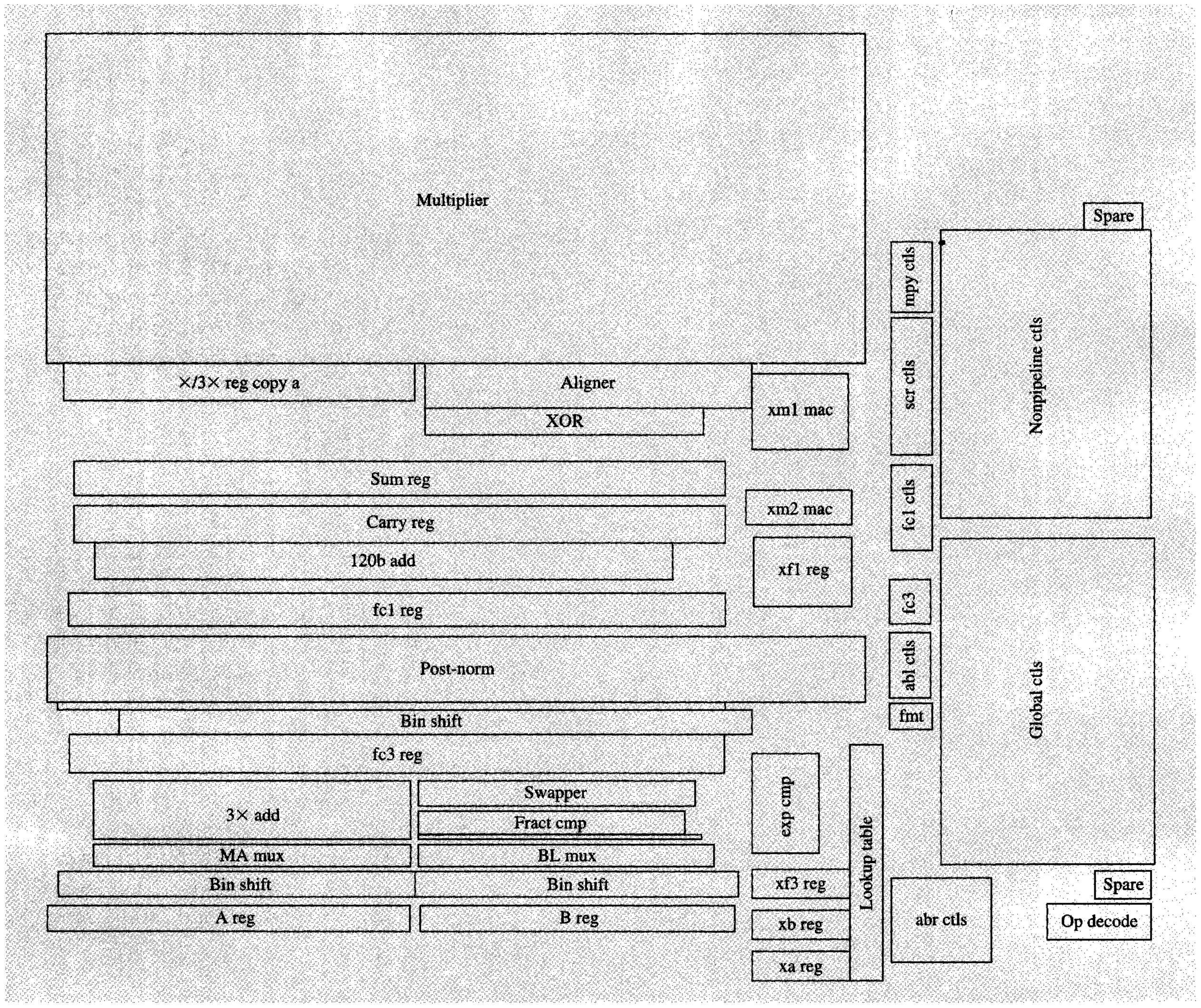

Figure 6

Macro layout.

1. Do not overlap the read and write cycle.
2. Overlap the read and write cycle.
3. Overlap the last execute cycle with the read cycle.
4. Overlap the write cycle with the first execute cycle.

The first two methods are low-performance and can be used for infrequent or complex cases. The last two methods are high-performance. However, method 4, which has been implemented for the early wrap case, has an interesting advantage in our implementation over method 3. The last execution stage can take place in several different pipeline stages: FS3, FS4, or even FS6. To implement method 3, a new bus or potentially several new buses would be needed to wrap the fraction data to the A and B registers. For method 4, only the C bus has to be used to drive into the late multiplexors. Since the C bus is an existing bus, no new tracks are needed for bypassing the fraction. Implementing method 4 reduces wiring congestion and gives good performance on data dependencies.

Physical design

The unit floorplan is shown in **Figures 6** and **7**. In effect, it is optimized around the multiplier. In order to meet the cycle time in the multiplier, we have reduced the 19 partial products generated in the fastest way possible—with six levels of CSA. This approach is very wire-intensive and consumes all of the wire resources on metal 1 to metal 4 in the heart of the multiply array. This fact was recognized early in the design, and the multiplier was placed at the very top of the unit (corner of the chip) to eliminate any wires that would have to cross it.

The dataflow was partitioned in 126-bit fraction and 14-bit exponent stacks. Some room on the sides of stacks was allocated for overflow logic required by many macros. All of the macros in the first cycle of execution are 60 bits wide (see Figures 1 and 2), while the rest of the macros are about 120 bits wide. To reduce unit area, we placed several 60-bit macros side by side. This created one horizontal wiring channel of 56 bits; however, overall we were able to reduce stack height.

Dataflow stacks were placed and wired manually. Since dataflow macro definitions were stable early in the design process, this approach led to the most compact design. Tracks were partitioned according to wire length, and wide wires were used for long nets to minimize delays and slews. Control macros were manually placed and wired with a vendor tool. A program was written that located areas for decoupling capacitor placements. The decoupling capacitors were placed both within and outside macros for a unit total of over 10 nF.

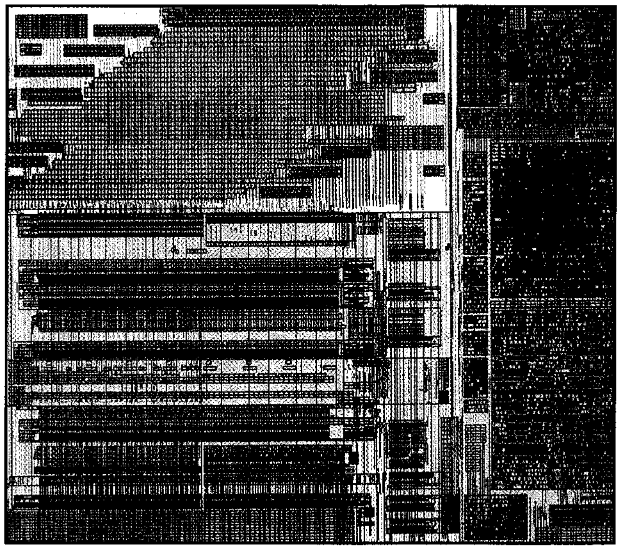
Figure 7

Physical design.

Circuit implementation
Most of the macros were implemented with static CMOS circuits. Only fraction/exponent dataflow registers and divide/square-root lookup ROS (read only storage) are dynamic [20]. The use of dynamic circuits was always an option for designers if delay goals were not met. However, through careful optimization of circuits in critical paths, we were able to meet our cycle-time objectives with what amounts to all static CMOS circuits.

Fraction/exponent dataflow macros were custom-designed. Even when common cells were used among several macros (such as register building blocks), they were custom-wired. Custom design was required to achieve both the cycle-time and area budgets in the dataflow macros. Control macros, on the other hand, were all synthesized and placed and routed with fixed-power library books. There are approximately 320K dataflow FETs in an area of 18.3 mm^2 and 61K control FETs in an area of 5 mm^2.

The control macros were implemented using a customized version of the IBM BooleDozer* logic synthesis tool. In order to meet the design schedule, it was very important to concentrate on improving our synthesis tool rather than manually tuning the results from synthesis. The design team restructured the VHDL to obtain improved synthesis implementation of the logic. This was done in parallel with focusing on improving the standard cell library and the logic transforms used within synthesis. The result of this effort was that all of the FPU control logic initially targeted for synthesis achieved the cycle-time goal and area constraints.

The timing reduction effort was facilitated by frequently running an FPU-level timing analysis to create timing reports and macro-level timing assertions used for synthesis and custom design. Unit-level timing runs were run in parallel with full-chip timing runs to create the necessary timing assertions. See [21] for more details on the design methodology.

Summary
A CMOS S/390 floating-point unit has been described which has been demonstrated at more than 400 MHz [22]. The design was optimized for frequently executed operations, and other instructions were implemented with cost-efficient algorithms. Some uncommon and aggressive algorithms were implemented (e.g., radix-8 multiplication; Goldschmidt division and square root); these algorithms were coupled with remainder-avoidance algorithms and parallel exponent calculation for normalization. Because the design caused critical control signals to be included in the dataflow, dataflow paths rather than control paths determined the cycle time. This made possible a reduced cycle time, which, coupled with a throughput of one cycle per instruction and a latency of three cycles for the most common additions and multiplications, resulted in a relatively high-performance FPU.

Acknowledgment

Design of the G4 floating-point unit would not have been possible without the combined efforts of the entire G4 project team. We especially acknowledge the contributions of Bob Averill, Robert Bunce, Rick Dennis, Dale Hoffman, Mike Mullen, Greg Northrop, Dianne Wassick, Charles Webb, Dave Webber, Barry Winter, Tom Wohlfahrt, and Fanchieh Yee.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. *Enterprise Systems Architecture/390*, Fourth Edition, Order No. SA22-7201-03, September 1996; available through IBM branch offices.
2. E. M. Schwarz, T. McPherson, and C. Krygowski, "Carry Select and Input Select Adder for Late Arriving Data," *Proceedings of the 30th Asilomar Conference on Signals, Systems, and Computers*, November 1996, pp. 182–185.
3. S. Vassiliadis, D. S. Lemon, and M. Putrino, "S/370 Sign-Magnitude Floating-Point Adder," *IEEE J. Solid-State Circuits* **24,** No. 4, 1062–1070 (August 1989).
4. S. Dao-Trong and K. Helwig, "A Single-Chip IBM System/390 Floating-Point Processor in CMOS," *IBM J. Res. Develop.* **36,** No. 4, 733–749 (July 1992).
5. S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, CBS College Publishing, New York, 1982.
6. K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley & Sons, Inc., New York, 1979.
7. R. M. Jessani and C. H. Olson, "The Floating-Point Unit of the PowerPC 603e Microprocessor," *IBM J. Res. Develop.* **40,** No. 5, 559–566 (September 1996).
8. S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan, "A General Proof for Overlapped Multiple-Bit Scanning Multiplications," *IEEE Trans. Comput.* **38,** No. 2, 172–183 (February 1989).
9. S. Vassiliadis, E. M. Schwarz, and B. M. Sung, "Hard-Wired Multipliers with Encoded Partial Products," *IEEE Trans. Comput.* **40,** No. 11, 1181–1197 (November 1991).
10. E. M. Schwarz, B. Averill, and L. Sigal, "A Radix-8 CMOS S/390 Multiplier," *Thirteenth Symposium on Computer Arithmetic*, Asilomar, CA, July 1997, pp. 2–9.
11. R. E. Goldschmidt, "Applications of Division by Convergence," Master's thesis, M.I.T., Cambridge, MA, June 1964.
12. S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Res. Develop.* **11,** No. 1, 34–53 (January 1967).
13. M. J. Flynn, "On Division by Functional Iteration," *IEEE Trans. Comput.* **C-19,** No. 8, 702–706 (August 1970).
14. A. Svoboda, "An Algorithm for Division," *Information Processing Machines* **9,** 25–32 (1963); Prague, Czechoslovakia.
15. E. V. Krishnamurthy, "On Optimal Iterative Schemes for High-Speed Division," *IEEE Trans. Comput.* **C-19,** No. 3, 227–231 (March 1970).
16. M. Darley, B. Kronlage, D. Bural, B. Churchill, D. Pulling, P. Wang, R. Iwamoto, and L. Yang, "The TMS390C602A Floating-Point Coprocessor for Sparc Systems," *IEEE Micro* **10,** No. 3, 36–47 (June 1990).
17. E. M. Schwarz, "Rounding for Quadratically Converging Algorithms for Division and Square Root," *Proceedings of the 29th Asilomar Conference on Signals, Systems, and Computers*, October 1995, pp. 600–603.
18. 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,** No. 8, 837–847 (August 1972).
19. S. Vassiliadis and E. M. Schwarz, "Controlling Unit for a Pipelined Floating Point Hard-Wired Engine," *Proceedings of the IFIP Third International Workshop on Wafer Scale Integration*, June 1989, pp. 343–351.
20. L. Sigal, J. D. Warnock, B. W. Curran, Y. H. Chan, P. J. Camporese, M. D. Mayo, W. V. Huott, D. R. Knebel, C. T. Chuang, J. P. Eckhardt, and P. T. Wu, "Circuit Design Techniques for the High-Performance CMOS IBM S/390 Parallel Enterprise Server G4 Microprocessor," *IBM J. Res. Develop.* **41,** No. 4/5, 489–503 (1997, this issue).
21. K. L. Shepard, S. M. Carey, E. K. Cho, B. W. Curran, R. F. Hatch, D. E. Hoffman, S. A. McCabe, G. A. Northrop, and R. Seigler, "Design Methodology for the S/390 Parallel Enterprise Server G4 Microprocessors," *IBM J. Res. Develop.* **41,** No. 4/5, 515–547 (1997, this issue).
22. C. Webb, C. Anderson, L. Sigal, K. Shepard, J. Liptay, J. Warnock, B. Curran, B. Krumm, M. Mayo, P. Camporese, E. Schwarz, M. Farrell, P. Restle, R. Averill, T. Slegel, W. Huott, Y. Chan, B. Wile, P. Emma, D. Beece, C. Chuang, and C. Price, "A 400 MHz S/390 Microprocessor," *ISSCC Digest of Technical Papers*, pp. 168–169 (February 1997).

Received December 5, 1996; accepted for publication April 17, 1997

Eric M. Schwarz *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (ESCHWARZ at PK705VMA, schwarz@vnet.ibm.com).* Dr. Schwarz received a B.S. degree in engineering science from The Pennsylvania State University in 1983, an M.S. degree in electrical engineering from Ohio University in 1984, and a Ph.D. degree in electrical engineering from Stanford University in 1993. He joined IBM in 1984 in Endicott, New York, and in 1993 transferred to Poughkeepsie. Dr. Schwarz is an Advisory Engineer and was FPU Logic Technical Leader for the S/390 Parallel Enterprise Server G4 processor. Currently, he is Execution Unit (FPU and FXU) Logic Technical Leader for follow-on processors. His research interests are in computer arithmetic and computer architecture. He is the author of seven filed patents, ten pending patents, and several journal articles and conference proceedings.

Leon Sigal *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (LJS at YKTVMV, ljs@vnet.ibm.com).* Mr. Sigal received a B.S. in biomedical engineering in 1985 from the University of Iowa and an M.S. in electrical engineering in 1986 from the University of Wisconsin at Madison. He worked at Hewlett-Packard's microprocessor development laboratory between 1986 and 1992. Mr. Sigal joined IBM in 1992 and has been leading the CMOS S/390 microprocessor circuit design interdivisional effort.

Thomas J. McPherson *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (MCPHERSO at PK705VMA, tmcpherson@vnet.ibm.com).* Mr. McPherson is a Staff Engineer in S/390 microprocessor development. He received a B.S. degree in electrical engineering from Rutgers University in 1990 and an M.S. degree in computer engineering from Syracuse University in 1992. Mr. McPherson joined IBM in 1990 and has worked on S/390 microprocessors and CMOS ASIC designs.