The IBM eServer z990 floating-point unit

G. Gerwig
H. Wetter
E. M. Schwarz
J. Haess
C. A. Krygowski
B. M. Fleischer
M. Kroener

*The floating-point unit (FPU) of the IBM z990 eServer™ is the first one in an IBM mainframe with a fused multiply-add dataflow. It also represents the first time that an SRT divide algorithm (named after Sweeney, Robertson, and Tocher, who independently proposed the algorithm) was used in an IBM mainframe. The FPU supports dual architectures: the zSeries® hexadecimal floating-point architecture and the IEEE 754 binary floating-point architecture. Six floating-point formats—including short, long, and extended operands—are supported in hardware. The throughput of this FPU is one multiply-add operation per cycle. The instructions are executed in five pipeline steps, and there are multiple provisions to avoid stalls in case of data dependencies. It is able to handle denormalized input operands and denormalized results without a stall (except for architectural **program exceptions**). It has a new extended-precision divide and square-root dataflow. This dataflow uses a radix-4 SRT algorithm (radix-2 for square root) and is able to handle divides and square-root operations in multiple floating-point and fixed-point formats. For fixed-point divisions, a new mechanism improves the performance by using an algorithm with which the number of divide iterations depends on the effective number of quotient bits.*

Introduction

This paper describes the floating-point unit (FPU) of the IBM z990 eServer* [1], a high-performance microprocessor optimized for commercial workloads. The FPU supports two architectures: binary floating-point (BFP) compliant with the IEEE 754 Standard [2] and hexadecimal floating-point (HFP) as specified by the IBM z/Architecture* [3]. There are six floating-point formats and six integer formats supported: short, long, and extended formats for the two architectures, as shown in **Figure 1**. (In the IBM PowerPC Architecture*, these formats are called *single, double,* and *quad*.)

In zSeries* microprocessors, extended-precision operations are implemented in hardware. This includes support for both HFP and BFP architectures and integer operands.

The z990 FPU had a variety of predecessors: the 1996 G3 FPU [4], the 1997 G4 FPU [5, 6], the 1998 G5 FPU [7, 8], the 1999 G6 FPU, and the 2000 z900 FPU [9]. Compared with them, the main z990 FPU goals were to optimize for BFP and have a fast multiply-add execution in order to support the increase in new zSeries workloads, particularly those utilizing Linux**. These applications are typically written in Java** or C++ and depend on BFP architecture even in commercial applications. The G5, G6, and z900 FPUs support BFP and HFP, but do not have a one-per-cycle throughput for BFP instructions. They have been in the field for five years, and customers are now heavily using BFP, so it was important to set a goal of improving BFP performance while emphasizing HFP performance. The goal was also to create a high-performance implementation, more like the IBM pSeries* workstations, which requires a dataflow that supports a fused multiply-add execution. This allows a theoretical peak performance of two floating-point operations per cycle. More details about this and a comparison with the POWER4* processor design can be found in [10].

For the divide implementation, we also looked at the zSeries predecessors and the POWER4 design. They all use a multiplicative divide algorithm, such as a

0018-8646/04/$5.00 © 2004 IBM

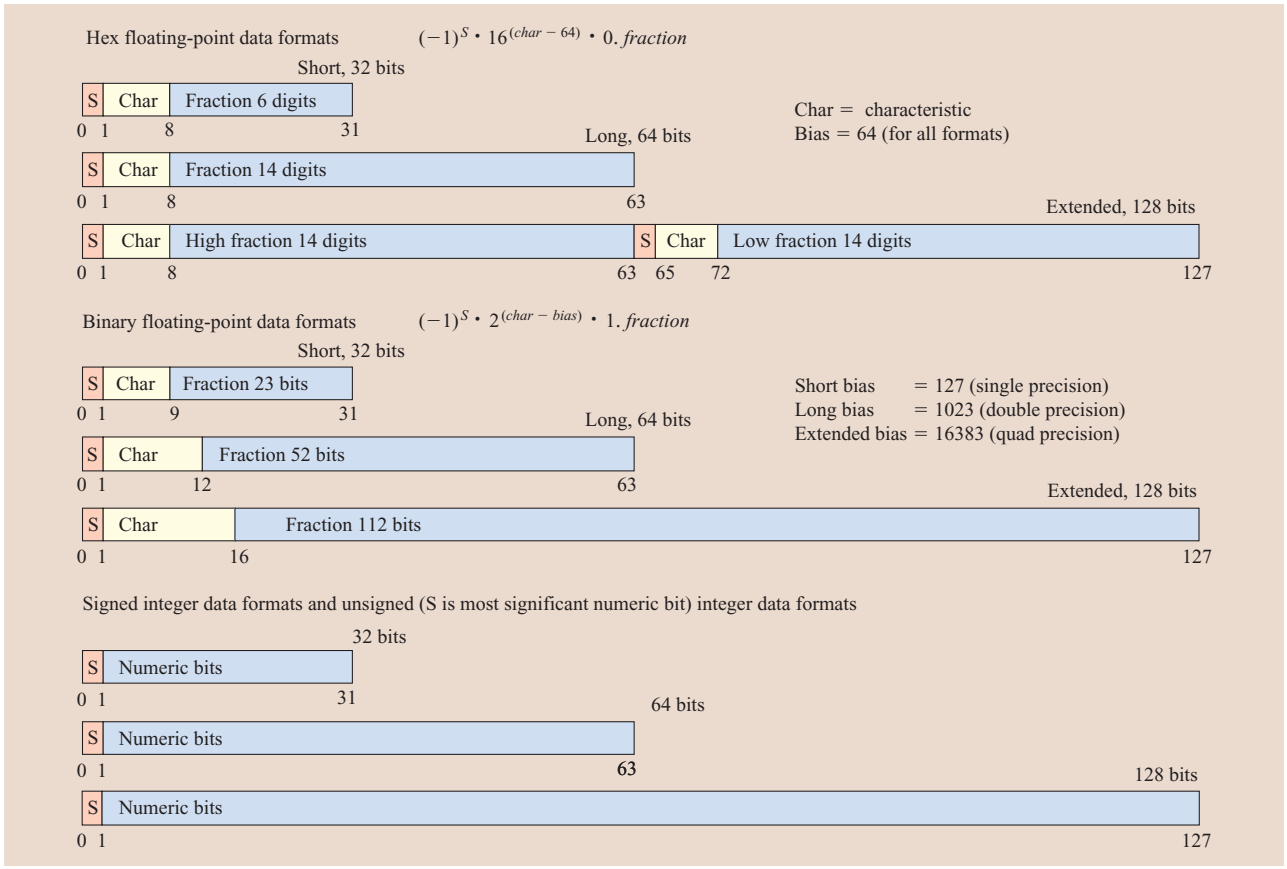

Figure 1

Data formats supported by the floating-point unit (FPU).

Goldschmidt or Newton–Raphson algorithm. Alternatives for the divide algorithm were investigated in the light of our goals to optimize the main dataflow and have a tighter cycle. We therefore looked at an SRT divide algorithm. It was named *SRT* after Sweeney, Robertson, and Tocher, who proposed it independently [11, 12]. The SRT divide algorithm has the following advantages and disadvantages for the z990 design:

1. The SRT divide hardware can be placed to the side to allow the optimization of the main dataflow for a fast fused-multiply-add implementation.
2. It can be made wide to allow extended operands, something that is not possible with a multiplicative algorithm, which is limited because of the width of the main multiplier.
3. It requires somewhat more area.
4. When no divide instruction is running, the clock of an SRT divide hardware can easily be switched off to save power.
5. The performance is better for instructions with a short operand format, since we gain two quotient bits per cycle.
6. The performance is worse for floating-point instructions with a long format.
7. The performance of integer divides can be improved, since we use a method to iterate on only the effective number of the quotient bits. This is based on the effect, according to our benchmarks, that most integer division results are only small numbers.
8. The rounding is simpler, since the partial remainder is always exact and therefore no back-multiplication is necessary.
9. The performance is better for instructions with extended format.

After consideration of these arguments, we decided upon the SRT implementation.

BFP and HFP architecture in one dataflow

Both the HFP and BFP architectures are already implemented in hardware in the 1998 IBM zSeries G5

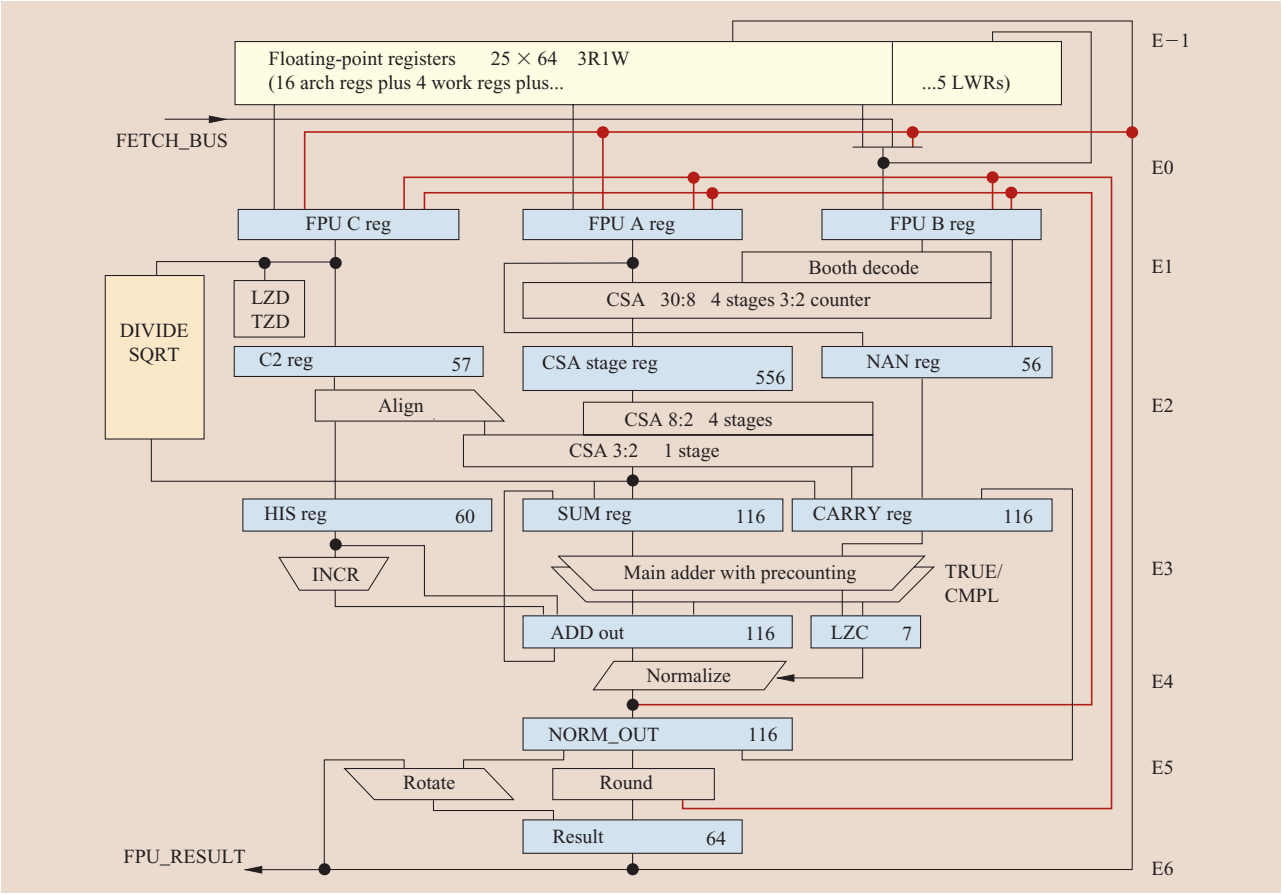

Figure 2

Main fraction dataflow of the FPU with feedback paths.

processor [7], but a hexadecimal dataflow is used. This requires binary operands to be converted to hexadecimal operands, creating an extra step. The HFP instructions are capable of performing one add or one multiply per cycle with a latency of about three cycles. The BFP instructions can be pipelined only at a rate of one instruction every other cycle because an extra cycle is necessary to convert between BFP and HFP.

The way of handling two formats in the same unit is to use an internal format that has a wider exponent and can include both formats. One problem is the different form of the bias. The exponent bias [13] in the HFP architecture is of the form 2^{n-1}, and in the BFP the architecture is of the form $(2^{n-1} - 1)$. The direct conversion requires the shifting of significands and the adding of constants to the exponents. The solution is an extended internal format with a separate internal representation for BFP and HFP:

$$X_{\mathrm{BFP}_i} = (-1)^{X_s} \cdot (1 + X_{\mathrm{f}}) \cdot 2^{X_{\mathrm{e}} - bias_{\mathrm{Bi}}},$$

$$bias_{\mathrm{Bi}_i} = 2^{n-1} - 1 = 32767,$$

$$X_{\mathrm{HFP}_i} = (-1)^{X_s} \cdot X_{\mathrm{f}} \cdot 2^{X_{\mathrm{e}} - bias_{\mathrm{Hi}}},$$

$$bias_{\mathrm{Hi}_i} = 2^{n-1} = 32768.$$

With that conversion, cycles that were necessary in the previous design can be avoided. More details about the two formats can be found in [10].

Dataflow overview and resolving of data dependencies

Figure 2 shows the pipeline-oriented structure of the FPU. The real execution cycles are E1, E2, E3, E4, and E5. In the E minus 1 (E–1) cycle, the instruction itself is decoded; in E0, a memory operand arrives and the floating-point register file (FPR) can be read out.

The main actions for these pipeline stages [10] are as follows:

E0 Load cycle.
E1 Calculate shift amount for alignment, Booth encoding, and first multiplier tree.
E2 Align addend and second multiplier tree.
E3 Main addition and counting of leading zeros (LZC).
E4 Normalization.
E5 Rounding.
E6 Write back.

When there is no dependency of operands between the instructions, a different instruction can be under execution in every pipeline stage at the same time. However, in some cases an instruction may require the result of a previous instruction as a source operand. In that case, the instruction has to wait in the E0 cycle until all source operands are available. To avoid such stalls—or at least to minimize the number of wait cycles—the feedback paths, shown in red in Figure 2, are designed into the fraction dataflow. Thus, for certain arithmetic instructions, a result of the normalizer can be forwarded directly into the A, B, or C source registers. Others may be forwarded out of the Rounder. The forwarding out of the Result register is possible in nearly every case.

The dependency of an instruction on previous loads is resolved by using the five Load-Wrap registers (LWRs). This register stack is part of the register file on the top, which contains the 16 architectured FPRs, four Work registers, and the five LWRs. All Load operands are staged through these LWRs. They can be addressed and read out like any other FPR, so for an actual instruction in E0, it is only a question of address modification to fetch the operands of previous load instructions. With this implementation, waits caused by Load instructions can be avoided.

Handling of denormalized inputs

Our architecture includes two types of instruction formats that are executed directly with memory operands: RX and RR. The RX type of instruction operates on one register operand and one storage operand, while the RR type of instruction operates on two register operands [13]. This is different from the IBM PowerPC Architecture, in which only loads can fetch memory operands. The advantage of RX instructions is that with one instruction, two operations can be executed at one time, a load and an arithmetic operation. It is the nature of such instructions that the memory operand arrives late at the execution unit, and there is no format conversion beforehand.

In the PowerPC Architecture, only RR instructions are possible, which allows the addition of tags to the operands in the register files, making it possible to recognize denormalized numbers. Such a tagging is not possible on the zSeries Architecture. Therefore, the detection of denormalized numbers must be integrated in the first execution cycle, which requires some additional effort in the multiplier and the aligner logic.

Multiplier with correction for denormalized operands

The multiplier is needed in a 56 × 56-bit width, which is required by the fraction width of the HFP long operands. This requires 29 partial products when using a radix-4 Booth encoding, which requires a counter tree of eight levels. To that, an extra correction term can be added without significantly increasing the delay.

At the beginning of the first execution cycle, it is assumed that all BFP operands are normalized and the implied unit bit has the value "1". When it is recognized that this assumption is not true, the correction is performed until the end of this first execution cycle.

The multiplier can be corrected by modifying a Booth term directly and feeding that term late into the counter tree. The correction of the multiplicand is more complicated and requires an additional correction term. When the exponent of the multiplicand is decoded during the cycle and a denormalized operand is detected, the implied unit bit of the multiplicand can be corrected when the correction term is a late entry in the counter tree.

The product P out of the multiplier Y and the multiplicand X is identical to the product out of the Booth term W_j as representation for the multiplier and the multiplicand X', when the correction term *lzcl* [14, 15] is subtracted, as shown below:

$$X = x_0 + \sum_{i=1}^{n-1} x_i \cdot 2^{-i},$$

$$Y = y_0 + \sum_{j=1}^{n-1} y_j \cdot 2^{-j},$$

$$Y = \sum_{j=1}^{\frac{n-1}{2}+1} W_j \cdot 4^{-j},$$

$$W_j \in \{-2, -1, 0, +1, +2\},$$

$$P = \sum_{j=1}^{\frac{n-1}{2}+1} W_j \cdot X \cdot 4^{-j},$$

$$X' = x_1 + \sum_{i=1}^{n-1} x_i \cdot 2^{-i},$$

$$X = X' - \overline{x_0},$$

$$P = \sum_{j=1}^{\frac{n-1}{2}+1} W_j \cdot X' \cdot 4^{-j} - Y \cdot \overline{x_0},$$

$$lzcl = -Y \cdot \overline{x_0}.$$

Aligner with correction for denormalized operands

The addend is shifted right according to the calculated shift amount D. It is simple to correct the implied unit bit of the addend directly, but the shift amount must also be corrected. To do this, the shift amount is calculated for the exponent difference $D + 1$ and $D - 1$ additionally and selected on a late multiplexer in the cycle, after the decoding for denormalized operands has been done.

With that, denormalized input operands can be handled without stalling the pipeline. More details about calculating the shift amount and alignment limitations can be found in [10].

Handling of denormalized results and limitations

The dataflow width is limited to an addend of 56 bits plus four guard bits and a product field, which is aligned with the adder of 112 bits and four guard bits for a total of a 176-bit-wide dataflow. There are certain cases of unnormalized and denormalized addends or products that require sophisticated handling in the hardware.

With regard to the alignment of the addend with the product, the radix point of the product is fixed in the dataflow. The radix point of the addend is shifted right to achieve the proper fraction alignment prior to the addition of the two. For HFP arithmetic, the radix point is always left of a field, while for BFP arithmetic, the radix points of the dataflow are partitioned as follows:

After alignment and main addition, there is a data width of 176 bits with two possible radix points for BFP operations, as shown below:

```
|Addend field      |Product field           |
|<---- 60 bits--->|<-- 116 bits  --------->|
|1.cccc . . .cGGGGGGG|xx.pppp......pGGGGGGGGGG|
  ^                  ^
  | Radix1           |  Radix2
```

Some extra guard bits `G` are appended to the fraction bits `c` of the addend and to the fraction bits `p` of the product.

In general, `Radix1`, which corresponds to the exponent of the addend E_c, is taken when the addend is greater than the product. This leads to a shift right of 60 bits before normalization. Accordingly, `Radix2`, which corresponds to the exponent of the product E_p, is taken when the addend is smaller than the product.

The normalization uses the precise leading zero count, LZC, to normalize the fraction and to calculate the normalized exponent $E_n = E_p - LZC$.

When a denormalized result is required (while underflow exceptions are masked), there is the potential possibility that $E_n < E_{min}$. This is a problem because $E_n < E_{min}$ is not allowed by the BFP architecture.

To avoid this, a mask is used in the leading zero count logic to allow limiting the leading zero count, $LZC_{max} = E_p - E_{min}$. This takes effect when E_n is in the range of E_{min} but still somewhat greater.

In one case, when an underflow exception will occur anyway (architectural exception), the dataflow is unable to deliver the correct denormalized result. This occurs when the addend is denormalized, the product is denormalized, and the underflow exception is not masked.

The dataflow is then disjointed and cannot generate the correct normalized result with a wrapped exponent. The solution is to generate a pseudo-exception by hardware and have it corrected by a routine in the millicode exception handler. Since we have an underflow condition that is not masked, this case will end in an architectural exception anyway, so there is no real performance degradation. For more details, see [10].

Rounding of different formats

Most schemes for rounding according to the IEEE 754 standard show a guard, a round, and a sticky bit. In our implementation, we have a precise normalized result; therefore, no additional guard bit is required. The rounding depends only on the least significant bit (LSB), the round bit, and a sticky bit.

Since we have three different BFP formats, we are limited by the widest format, which is the extended format with a mantissa of 113 bits. For the smaller formats (short and long), we also need the appropriate round bit and a sticky bit, which consists of a merge of the remaining bits on the low-order side of the mantissa.

With this and the actual rounding mode, we can determine via a rounding table which part of the fraction should be used incremented and which part should be taken as is for the rounded result. The LSBs of the fraction are needed for the rounding mode *round to nearest*. **Figure 3** shows how the rounder dataflow is implemented in an aggressive cycle time.

Division and square root

The architecture requires an operand width of up to 116 bits for floating-point operands of divide and square-root mantissas. An extra-wide (116-bit) divider dataflow permits that calculation to be performed in hardware. An SRT algorithm is used to gain two Bits of the quotient for divide operations and one bit of the root for a square-root operation in one iteration (one cycle).

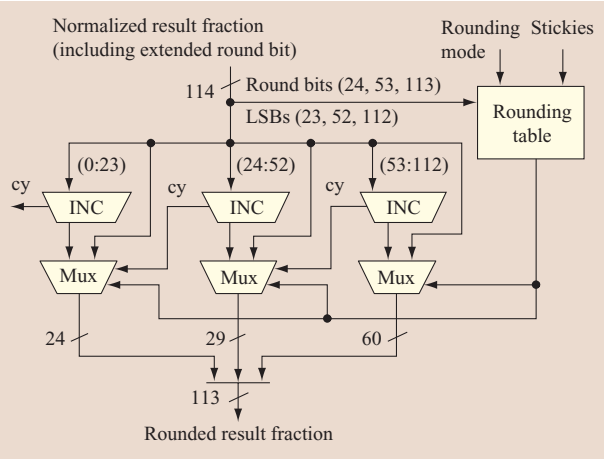

Figure 3

Structure of the fraction rounder.

A wide variety of different divide and square-root instructions and formats are supported. In addition to the six floating-point formats, there are also integer formats for divide instructions with an operand width of 32, 64, or 128. These binary integer operands may be signed or unsigned. In an unsigned operand, all bits are used to express the absolute value of the number, while for signed binary integers, the leftmost bit represents the sign, followed by the numeric field. Negative numbers are represented in two's-complement binary notation with a one in the sign-bit position [13].

SRT algorithm for division

SRT is frequently used in modern designs to implement divide and square root. It is an iterative algorithm, and one digit of the quotient can be determined in each iteration. During each iteration, the next partial remainder is calculated by multiplying the previous partial remainder with the radix of the algorithm and subtracting a multiple of the divisor. One divide iteration can be expressed as

$$P_{i+1} = r \cdot P_i - q_{i+1} \cdot D,$$

where P represents the partial remainder, q represents the quotient digit guess, D is the divisor, and r the radix of the algorithm. The final quotient is the weighted sum of all quotient digits.

Algorithm for square root

For the square-root operation, the root is both the *quotient* and the *divisor* when the divide nomenclature is applied to that operation. Since the root (divisor) is only partly available, the equation for the iteration is slightly more complicated,

$$P_{i+1} = r \cdot P_i - q_{i+1} \cdot 2Q_i - q_{i+1}^2 r^{-(i+1)},$$

where P represents the partial remainder, q represents the root digit guess, Q is the actual developed root, and r is the radix of the algorithm. The final root is the weighted sum of all root digits.

Implementation of the algorithms

Since the value for the actual quotient digit q_{i+1} is estimated, the partial remainder P_{i+1} can be negative. This can be compensated by also allowing negative values for q_{i+1}. Allowing negative values for q_{i+1} makes it possible to correct errors of the actual partial remainder in later iterations. The algorithm for division is convergent as long as the relation $P_{i+1} < (q_{\max} \cdot D)/(r - 1)$ is met. The visual representation of this equation can be shown in a so-called *P–D plot*. The ranges of q_{i+1} can be seen in **Figure 4**.

In our implementation for division, a radix of 4 is used with a maximally redundant digit set [16, 17]. This reduces the cost of the quotient estimate table lookup at the expense of an increase of the range of quotient digits. Because of the short cycle time, a full-width carry-propagate adder (CPA) would not fit into the required cycle time. Therefore, carry-save adders (CSAs) are used, which is possible when the partial remainder is not explicit, but in a carry-save redundant form. The effort for this has been reduced by using only one carry bit for every four sum bits.

With the sum and carry parts of the partial remainder and a radix of 4, the equation of one divide iteration looks like this:

$$P_{i+1} = P_{S_{i+1}} + P_{C_{i+1}},$$

$$P_{S_{i+1}} + P_{C_{i+1}} = 4(P_{S_i} + P_{C_i}) - q_{i+1} \cdot D,$$

$$q_{i+1} \in \{-3, -2, -1, 0, +1, +2, +3\},$$

$$q_{i+1} = q_{i+1,1} + q_{i+1,2},$$

$$P_{S_{i+1}} + P_{C_{i+1}} = 4(P_{S_i} + P_{C_i}) - q_{i+1,1} \cdot 1D - q_{i+1,2} \cdot 2D,$$

where P_S and P_C are the partial remainder in a sum-and-carry redundant form and $q_{i+1,1}$ and $q_{i+1,2}$ are the quotient digit guesses separated into a guess of 1 and a guess of 2, where each can take on the values -1, 0, or $+1$.

For a square root, the radix is 2 ($r = 2$), and the partial remainder and the developed root are in a redundant form, which leads to

$$P_{i+1} = P_{S_{i+1}} + P_{C_{i+1}},$$

$$Q_i = Q_{P_i} + Q_{N_i},$$

$$P_{S_{i+1}} + P_{C_{i+1}} = 2(P_{S_i} + P_{C_i}) - q_{i+1} \cdot 2Q_{P_i} + q_{i+1} \cdot 2Q_{N_i} - q_{i+1}^2 \cdot r^{-(i+1)},$$

$$q_{i+1} \in \{-1, 0, +1\}.$$

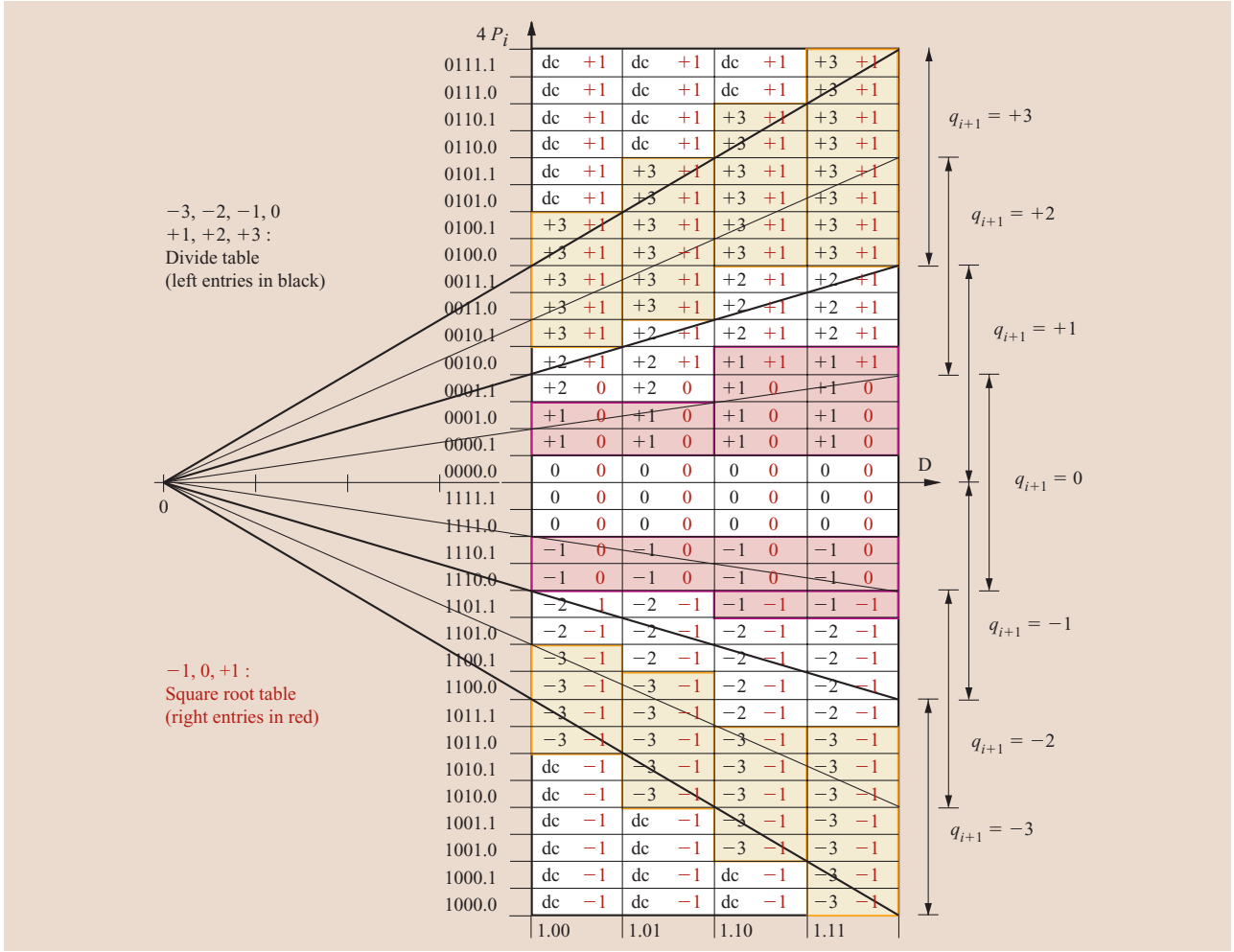

Figure 4

Lookup table for divide and square root.

The term $q_{i+1}^2 \cdot 2^{-(i+1)}$ is always a single bit in the lowest significant position. Therefore it can be handled as *carry in* to the subtractor.

Figure 5 shows the main elements of the fraction dataflow for divide and square root. The black entries are needed for the divide function, while the red entries are the additions needed for square root. It can be seen that the dataflow structures for divide and square root are very similar in this approach; it is a very effective way to add a square-root function to a divide implementation with very little additional hardware.

Divisor register

This is a simple register with the maximum width of 113 bits for a BFP extended-precision operand. The divisor is constant during the divide iterations.

Table lookup

Only a very small amount of combinatorial logic is needed to implement this function. For the divide operation, it has the five most significant bits of the partial remainder and the two most significant bits of the divisor as input. The output consists of three bits, the sign of the quotient digit, and two bits for the value of the quotient digit.

For square root, the table function is even simpler. It requires only the three most significant bits of the partial remainder as input. As output, two bits would be sufficient, a sign bit and a value bit. Since the table is merged for the divide and the square-root function, the value is expressed there in two bits as well.

Figure 4 is a combination of a P–D plot and the actual implemented lookup table. It illustrates the shifted partial remainder ranges in which a quotient digit can be selected

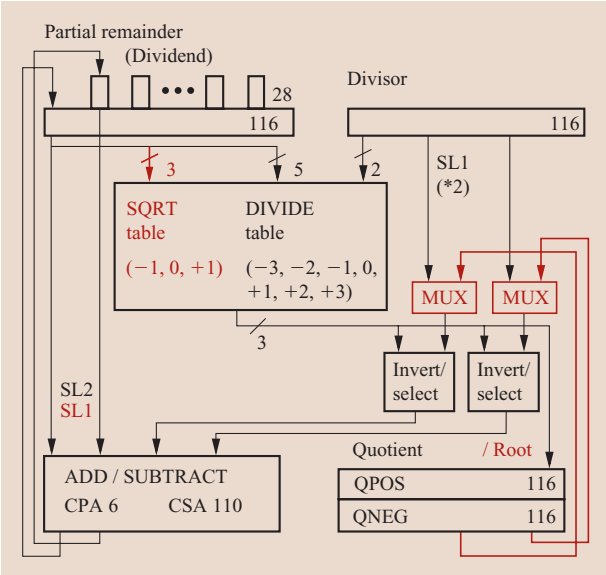

Figure 5

Dataflow structure for divide and square root.

without violating the bounds on the next partial remainder. It can be seen that the table is asymmetric with respect to +/− values because the partial remainder has a redundant form which causes an additional error. Because of this, the high-order bits of the partial remainder can be too small (by one bit in the lowest position), but can never be too large.

Generation of subtracting values

The estimated quotient digit determines the required multiples of the divisor that have to be subtracted during the actual iteration. Before selecting the multiples of one or two, the divisor is inverted, depending on the sign bit of the partial remainder. When the partial remainder is negative, we have to add divisor multiples; when it is positive, we have to subtract. The $q_{i+1,1}$ term represents divisor multiples of 1, and the $q_{i+1,2}$ term represents divisor multiples of 2. For square root, the same invert and multiplex logic can be used, but a previous multiplexer must select between the divisor (divide) and root (square root).

Partial remainder register and subtractor

The two selected multiples of the divisor (or of the root for square root) and the shifted contents of the actual partial remainder are added together, and the result is then latched into the partial remainder register. The 116-bit width of this register is required by the HFP extended format, including one hexadecimal guard digit. The register consists of a sum part of 116 bits and a carry part of 28 bits. The six high-order sum bits must be explicit without a corresponding carry because they are used in the table lookup. The most significant carry bit starts at position 6, and only every fourth carry bit is stored. This is possible because the subtractor does not use a full 4:2 reduction, but instead uses one stage of 3:2 reduction (CSAs) and one stage of CPAs with a width of four bits. On the high-order side, one CPA with a six-bit width is needed to deliver an explicit value to the table.

These four-bit-wide CPAs in the low-order range save latches, area, and power, and do not cost cycle time, since the six-bit CPA is needed anyway in the high-order range.

Quotient or root register

The quotient register requires the same width as the partial remainder (116 bits) and consists of two parts: the Q_{pos} and the Q_{neg} registers. When the sign of q_{i+1} is positive, it is stored in Q_{pos}; otherwise, it is stored in Q_{neg}. The control logic delivers a pointer that defines the position of the quotient register in which the actual q_{i+1} has to be stored. Typically, this is the output of a counter within the control logic. The final quotient or root is built after the last iteration by adding these two parts in the main adder of the FPU.

Execution and performance of floating-point divides

Before the divide operation can start, the operands must be loaded into the dividend and divisor registers. When an operand is denormalized, a normalization in the main dataflow is needed in advance. The required number of divide iterations depends on the data format (short, long, or extended). After the divider has completed enough iterations, the sum and carry parts of the remainder and the quotient are moved out into the main dataflow before the main adder. There, they are added to obtain the explicit value of remainder and quotient. The raw quotient still may have a leading zero, which is removed by normalization. The BFP quotient additionally requires rounding. For this, the normalizer and rounder of the main FPU dataflow are used. **Table 1** shows the cycles required for execution of IEEE floating-point divide instructions.

Execution and performance of integer divides

In classical benchmarks for mainframes, the results of integer divide operations are often very small numbers. Since the result width of an integer operand is fixed for a certain instruction, this means that in most cases the result contains many leading zeros, so there is a potential to improve performance when the calculation of small numbers would be faster.

Previous divide implementations are oriented to the maximum width of a divide result and always require the

same (maximum required) number of iterations to calculate the result. With our implementation, the number of divide iterations depends purely on the effective number of quotient bits. We skip the calculation of the leading zeros, which considerably improves the performance of integer divides (**Table 2**).

Integer operands must be positive and normalized for our divide calculation, which is done by a loop through the main dataflow. Afterward, the operands are loaded into the dividend and divisor registers, in the same way as for floating-point operands, but they are handled differently when the estimated q_{i+1} is stored in the quotient registers. A start pointer and a stop pointer are calculated by control and applied to the quotient register, allowing the execution of only the real required iterations.

The number of effective bits n_Q in the quotient can be calculated in advance, when the effective bits n_V and n_Q of the dividend and the divisor are known. These values are calculated during the normalization process. For a 64-bit integer division, the following equations are valid:

$$n_{Q0} = n_V - n_D \text{ for } V_{norm} < D_{norm}$$

and

$$n_1 = n_V - n_D + 1 \text{ for } V_{norm} \geq D_{norm}.$$

Since we gain two bits per cycle, the number of effective quotient bits n_{Qe} must be rounded up to the next even number. The start pointer P_{Start} and the stop pointer P_{Stop} for a 64-bit integer divide are given by

$$P_{Start} = 64 - n_{Qe}$$

and

$$P_{Stop} = 64.$$

Physical implementation

The FPU occupies an area of 3.76 mm^2, and the divide dataflow requires an area of 0.22 mm^2, which is about 6% of the FPU. This is still an acceptable part for a divide implementation, even though a multiplicative divided implementation might be somewhat smaller. The chip is fabricated in IBM 0.13-μm CMOS SOI technology, with a power supply of 1.15 V and a temperature of 50°C, and it supports a clock frequency of 1.2 GHz.

In the left-hand area of the layout shown in **Figure 6**, the fraction dataflow can be seen. This fraction dataflow is implemented in a bit-stack approach. The A, B, and C registers have a width of 56 bits. The bit stack is widened on the alignment and multiplication to 116 bits. The adder, normalizer, and rounder are also 116 bits wide. The normalizer is located on the bottom.

The layout follows the pipeline-oriented figure of the main fraction dataflow, but the pipeline stages are interleaved and folded here. Thus, the rounder is above the multiplier, since it has connections to the normalizer as well as to the FPRs, which are at the top of the layout. The exponent stack is located to the right of the A, B, and C registers.

On the right-hand side are the synthesized control-logic macros. A macro is a part of hardware that is logically and physically designed to be independent of other macros. For each execution pipeline stage, there is one separate control macro. The divider is implemented in a stack approach as well, whereby the divide table is combinatorial logic which occupies a very small area on the left-hand side of the divider macro. Since the interconnection of the divide engine with the main fraction dataflow is not timing-critical, this can be located away from the main dataflow. In this case, it is in the upper right corner of the layout.

The divider macro is completely designed in standard inverting CMOS logic. Although it has been implemented as a full custom macro, extensive use has been made of a standard cell library in order to minimize the layout effort.

As a power-saving feature, most parts of the FPU can be turned off completely when not in use. For enhanced testability, each of the master–slave latches is accompanied by an additional scan latch. Adding this extra scan latch to

Table 1 Cycles required for execution of IEEE floating-point divide instructions.

Action	*Cycles*		
	Short format	*Long format*	*Extended format*
Load operands	3	3	15
Divide loops	14	28	58
Readout remainder/quotient	4	4	4
Calculate quotient	1	1	1
Normalize	1	1	1
Round	1	1	1
Write back	1	1	2
Total latency	30	39	82
Pipelined latency	25	34	77

Table 2 Execution and performance for integer divides.

Action	*Cycles*
Load and concatenate	4
Normalize	5
Divide loops	1–32
Readout remainder/quotient	5
Invert sign (potential)	5
Write back	5
Total latency	30–61
Pipelined latency	25–56

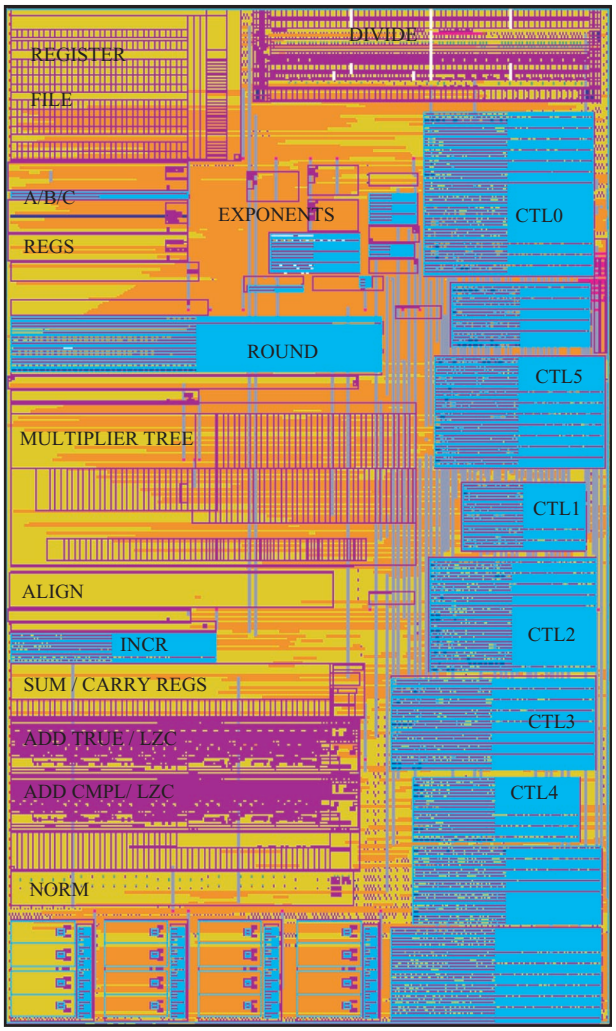

Figure 6

Physical layout of the FPU.

the scan chain configuration results in increased transition fault coverage.

Summary

The new IBM eServer z990 floating-point unit has been described. It includes a fused multiply-add dataflow in hardware that allows a throughput of one multiply-and-add in one cycle with a latency of five cycles. Two architectures, for binary and hexadecimal floating-point instructions, are supported with each short, long, and extended format. This has been accomplished by a unique method of representing the two architectures with two internal formats with their own biases. Denormalized numbers can be handled on the input side as well as on the output side in the normal flow, except for one rare case, which is destined for an underflow exception handler anyway.

Also, a fast divider dataflow is implemented which is capable of supporting an extended-precision width and achieves two result bits per cycle. This divider hardware is also used for square-root operations. For integer divides, the number of divide iterations depends on the effective number of quotient bits, which significantly improves the performance. For the redundant expression of the partial remainder only, each fourth carry bit is used, which saves about 80 latches when compared with a conventional carry-save approach.

Acknowledgments

In addition to the authors, many other individuals contributed to the success of the z990 FPU. Special acknowledgment goes to Juergen Foag and Andree Marth for their work on the design of the divide logic; to Hans-Juergen Muenster for his work on control logic and timing coordination; to Lukas Daellenbach, Dave Rude, Peter Cook, Steve Klepner, Fanchieh Yee, Harald Mielich, and Rainer Clemen for their work on custom design and integration; and to Juergen Vielfort and Klaus Keuerleber for their excellent work on verification and simulation.

References

1. T. J. Slegel, E. Pfeffer, and J. A. Magee, "The IBM eServer z990 Microprocessor," *IBM J. Res. & Dev.* **48,** No. 3/4, 295–309 (May/July 2004, this issue).
2. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard 754-1985*, The Institute of Electrical and Electronics Engineers, Inc., New York, August 1985.
3. IBM Corporation, *Enterprise Systems Architecture/390 Principles of Operation* (SA22-7201); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
4. G. Gerwig and M. Kroener, "Floating-Point-Unit in Standard Cell Design with 116 Bit Wide Dataflow," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, April 1999, pp. 266–273.
5. E. M. Schwarz, L. Sigal, and T. J. McPherson, "CMOS Floating-Point Unit for the S/390 Parallel Enterprise Server G4," *IBM J. Res. & Dev.* **41,** No. 4/5, 475–488 (July/September 1997).
6. E. M. Schwarz, R. M. Averill III, and L. J. Sigal, "A Radix-8 CMOS S/390 Multiplier," *Proceedings of the 13th IEEE Symposium on Computer Arithmetic (ARITH '97)*, Asilomar, CA, July 1997, pp. 2–9.
7. E. M. Schwarz and C. A. Krygowski, "The S/390 G5 Floating-Point Unit," *IBM J. Res. & Dev.* **43,** No. 5/6, 707–721 (September/November 1999).
8. E. M. Schwarz, R. M. Smith, and C. A. Krygowski, "The S/390 G5 Floating Point Unit Supporting Hex and Binary Architectures," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, April 1999, pp. 258–265.

9. E. M. Schwarz, M. A. Check, C.-L. K. Shum, T. Koehler, S. B. Swaney, J. D. MacDougall, and C. A. Krygowski, "The Microarchitecture of the IBM eServer z900 Processor," *IBM J. Res. & Dev.* **46,** No. 4/5, 381–395 (July/September 2002).
10. G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess, "High Performance Floating-Point Unit with 116 Bit Wide Divider," *Proceedings of the 16th Symposium on Computer Arithmetic*, Santiago de Compostela, Spain, June 2003, pp. 87–94.
11. K. D. Tocher, "Techniques of Multiplication and Division for Automatic Binary Computers," *Quart. J. Mech. Appl. Math.* **11,** Pt. 3, 364–384 (1958).
12. J. E. Robertson, "A New Class of Digital Division Methods," *IRE Trans. Electronic Computers* **EC-7,** 218–222 (September 1958).
13. IBM Corporation, *z/Architecture Principles of Operation* (SA22-7832); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
14. C. A. Krygowski and E. M. Schwarz, "Floating-Point Multiplier for De-Normalized Inputs," U.S. Patent Application No. 2002/0124037 A1, p. 8, September 5, 2002.
15. E. M. Schwarz, M. Schmookler, and S. D. Trong, "Hardware Implementations of Denormalized Numbers," *Proceedings of the 16th Symposium on Computer Arithmetic*, Santiago de Compostela, Spain, June 2003, pp. 70–78.
16. M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers, Boston, 1994.
17. D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT Division Architectures and Implementations," *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, Asilomar, CA, July 1997, pp. 18–25.

Received September 22, 2003; accepted for publication November 24, 2003; Internet publication April 6, 2004

Guenter Gerwig *IBM Server Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (ggerwig@de.ibm.com).* Mr. Gerwig received his B.S. and M.S. degrees in electrical engineering from the University of Stuttgart in 1978 and 1981, respectively. He joined IBM in 1981 to work on chip card readers for banking systems. Since 1989 he has worked on floating-point designs for CMOS microprocessors. He is the team leader responsible for the design of the G2, G3, z990, and a future floating-point unit. Mr. Gerwig received a third-plateau IBM Invention Achievement Award and holds 11 patents. He is currently the team leader responsible for a future recovery unit design.

Holger Wetter *IBM Server Group, IBM Deutschland Entwicklung GmbH, Schoenaichestrasse 220, 71032 Boeblingen, Germany (hwetter@de.ibm.com).* Mr. Wetter received his M.S. degree in electrical engineering from the University of Hannover in 1995 and joined IBM the same year. Since then he has been working in the area of custom circuit design on several generations of zSeries microprocessors. In particular, he was involved in the design of the floating-point unit of the z990 microprocessor. Mr. Wetter is a coauthor of several publications and seven patents. He is currently working on floating-point units for future pSeries and zSeries microprocessors.

Eric M. Schwarz *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (eschwarz@us.ibm.com).* Dr. Schwarz received a B.S. degree in engineering science from 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 IBM Poughkeepsie. He is a Senior Technical Staff Member and was the z900 microprocessor logic design leader. Dr. Schwarz provides support to IBM floating-point unit teams and is on the program committee of the IEEE Symposium on Computer Arithmetic. He is also actively researching computer architecture and heads two IBM/CMU microarchitecture work groups. Dr. Schwarz is currently working on future microarchitectures for IBM zSeries, iSeries*, and pSeries processors. He is an author of many patents, conference proceedings, and technical reports.

Juergen Haess *IBM Server Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (haess@de.ibm.com).* Mr. Haess received his M.S. degree in electrical engineering from the University of Karlsruhe in 1980. He joined IBM to work on I/O adapter development, which included several released products. In 1994 he spent six months at IBM Poughkeepsie, where he facilitated the transfer of the G3 FIB-chip design to that location. He also coordinated this new I/O system with the Boeblingen I/O designs. In 1997 he joined the CPU development team, where he became a member of the floating-point unit team. He was responsible for the design of the z990 new binary fused multiply-and-add instructions. He is currently working in FPU design on binary instructions for the next IBM pSeries and zSeries processors. Mr. Haess is an author or coauthor of several patents, a conference proceeding, and two journal articles. He has received a number of awards for his work, including three IBM Invention Achievement Awards.

Christopher A. Krygowski *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (cakryg@us.ibm.com).* Mr. Krygowski received a B.S. degree in electrical engineering from Clarkson University in 1989 and an M.S. degree in electrical engineering from the National Technological University. He is currently a Senior Development Manager leading microprocessor development for future zSeries systems. Mr. Krygowski joined IBM in 1989 and has had various design and management responsibilities in development of the IBM zSeries central processor units. His recent contributions are the design of the integer and floating-point execution units of the z990 processor. He is the author of 17 patents and several technical articles. Mr. Krygowski has received four IBM Outstanding Technical Achievement Awards for his work on IBM zSeries processors.

Bruce M. Fleischer *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (fleischr@us.ibm.com).* Dr. Fleischer is a Research Staff Member in the VLSI Design Department at the IBM Thomas J. Watson Research Center. He received his A.B. degree in physics from Harvard University in 1981, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University in 1987 and 1989, respectively. He joined the IBM Research Division in 1989 to do BiCMOS circuit design. From 1989 to 1992 he worked on BiCMOS SRAM and communications circuits. Since 1992, he has worked on microprocessor circuit design. Dr. Fleischer was a member of the floating-point team for the z900 microprocessor, and the lead circuit designer for the z990 microprocessor floating-point unit. He is the author or coauthor of a textbook chapter, several technical articles, and eight patents. Dr. Fleischer received an IBM Outstanding Technical Achievement Award for his work on the G7 (z990) microprocessor.

Michael Kroener *IBM Server Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (mkroener@de.ibm.com).* Mr. Kroener received his B.S. degree in communications engineering from the Fachhochschule fuer Technik in Esslingen in 1994. He joined IBM in 1994 to work on the design and verification of floating-point designs for CMOS microprocessors. In 1999 Mr. Kroener began an assignment at IBM Poughkeepsie, New York, where he joined the Fixpoint Development Team for G4. In 2001, he was invited to the IBM Academy, where he presented the most recent floating-point design. He is the author of four patents and several technical articles. Mr. Kroener recently became a floating-point representative in the IBM Development Team in Austin, where he is responsible for integrating the actual floating-point design into the pSeries microprocessor.