The S/390 G5 floating-point unit

by E. M. Schwarz
C. A. Krygowski

The floating-point unit of the IBM S/390® G5 Parallel Enterprise Server represents a milestone in S/390 floating-point computation. The S/390 G5 contains the first floating-point unit (FPU) to support both the S/390 hexadecimal floating-point architecture and IEEE Standard 754 for binary floating-point arithmetic. The S/390 G5 FPU supports the new S/390 floating-point architecture, which contains six operand formats, including the IEEE 754 standard singleword, doubleword, and quadword formats, which are all supported in hardware. An internal hexadecimal-based dataflow is implemented to support both hexadecimal- and binary-based architectures. The S/390 G5 server is generally available at 500 MHz. The microprocessor chip is fabricated in IBM CMOS 6X technology, with a device size of 0.25 μm as drawn and 0.15 μm effective length. The design of the G5 FPU is based upon that of its predecessor, the G4. All of the custom dataflow macros from the G4 hexadecimal FPU were utilized with only minor modifications, and only a few additional macros for format conversion were required. This paper discusses the changes that were required to support the new S/390 binary floating-point architecture.

Introduction

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) [1] was developed in 1985 to standardize computation among the various computer manufacturers. This standard has been adopted by virtually all PC, workstation, and midrange computer manufacturers. Mainframes have been using proprietary floating-point formats which are incompatible with and vastly different from the IEEE 754 standard. To facilitate participation in nontraditional markets, the S/390* architecture is expanding to support multiple floating-point formats. The new S/390 architecture [2] defines a superset of the union of the IEEE 754 standard and the S/390 hexadecimal format.

The floating-point unit of the S/390 G5 server represents a milestone in the history of IBM mainframe servers. The G5 is the first S/390 processor to support IEEE 754. The G5 FPU implements the new S/390 architecture, which has six floating-point formats, 175 instructions, and five rounding modes. Please see the paper by Abbott et al. in this issue [3] for additional details on the S/390 binary floating-point facility.

The G5 FPU relies heavily on the design of its predecessor, the G4 FPU [4]. The floating-point dataflow was "retrofitted" with a minimal amount of additional hardware to support the new binary floating-point architecture. This implementation strategy was chosen to remain consistent with the overall central processor (referred to as the microprocessor, or CP) design schedule. The G5 CP design schedule was relatively short, which is typical of an incremental design. In evolving from its G4 predecessor, the G5 CP contains a technology change to a denser and faster technology, some performance enhancements, such as increased cache size and a branch target buffer, and functional enhancements (most notably, the addition of binary floating point) [5–8].

0018-8646/99/$5.00 © 1999 IBM

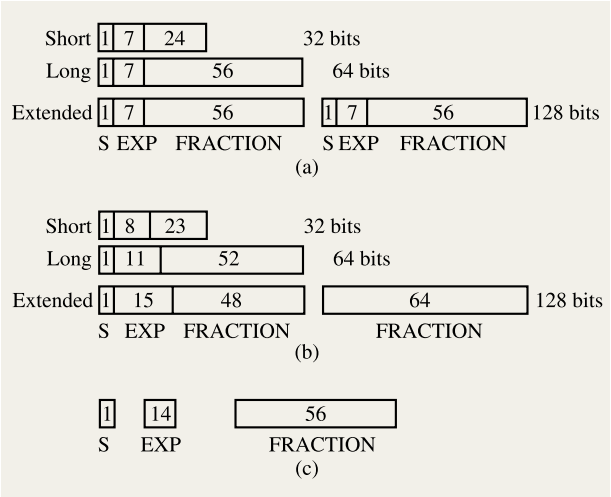

Figure 1

G5 FPU-supported formats: (a) Hex-architected; (b) binary-architected; (c) hex-internal.

The G5 processor comprises four units: the instruction unit, the execution unit, the recovery unit, and the L1 cache unit. IBM S/390 mainframes are more reliable than PCs and workstations, and this tradition is continued with the G5 microprocessor. Both the execution unit (E-unit) and the instruction unit (I-unit) are duplicated on chip and execute on the same instruction stream, and the results are compared by the recovery unit and L1 cache unit. The processor contains a unified cache design comprising both instructions and operands. The I-unit fetches, decodes, and dispatches instructions to the E-unit. The I-unit also fetches operands from storage for the E-unit. Instructions are fetched with a two-level branch prediction that uses a branch target buffer. The execution unit comprises two subunits, the fixed-point unit and the floating-point unit. The fixed-point unit (FXU) executes general types of instructions which operate upon integer data. The G5 E-unit can accept a single instruction for execution each cycle. The register files are all located in the FXU, including general-purpose registers (GPRs) and floating-point registers (FPRs). Since the FXU and FPU cannot execute in parallel, the FXU is used for the starting of floating-point instructions to access data from the register files and align storage data for the FPU. The G5 FPU receives two operands on 64-bit buses from the FXU.

The G5 FPU implements most short- and long-precision instructions in hardware in a pipelined manner. Extended-precision instructions are also implemented in hardware, but in a nonpipelined manner [9]. Other nonpipelined but hardware-executed instructions include divide, square root, and multiply-then-add. These instructions are executed in a nonpipelined mode, since the control sequencing complexity of these instructions would not be worth the potential performance increase. The only instructions which are implemented in low-level software called millicode are the control instructions which operate on the floating-point control (FPC) word, the divide-to-integer instruction, and conversion instructions between fixed-point and floating-point formats. All other instructions and special cases are implemented directly in hardware. The FPU also executes fixed-point multiply and divide instructions. Floating-point store instructions, which were previously executed in the fixed-point unit on the G4, are now executed in the FPU. The store instructions are executed in a pipelined manner and can exploit wrap paths internal to the FPU to enhance the performance of data-dependent store instructions.

The key feature of the G5 FPU is its ability to implement both the hexadecimal and binary floating-point formats on one FPU. This is accomplished with small changes to the G4 FPU by adopting an internal format which is hexadecimal-based but supports the wider-range numbers of the binary floating-point format [10]. This feature is described in more detail, along with some of the other changes which enhanced performance. Also detailed are subtleties of the IEEE 754 architecture which are difficult to implement correctly.

In the first section of this paper, we provide a brief description of the different S/390 floating-point data formats. The next section reviews the G4 floating-point dataflow and discusses the modifications that were necessary to support the binary floating-point facility. The final section discusses the performance of the G5 floating-point unit.

S/390 floating-point data formats

Six architected formats are supported by the G5 FPU, as shown in **Figure 1**. The figure shows the partitioning of each format into sign (S), exponent (EXP), and fraction bits. The hex and binary extended formats both utilize a full quadword format. The hex extended format has two sign bits and two exponent fields, which differ by 14. This allows the hex extended-format number to be separated into two contiguous long-format numbers. The binary extended format is similar to other manufacturers' binary floating-point quadword format, which is a subset of the IEEE 754 floating-point format double-extended. All instructions which input an extended-format operand are implemented in a nonpipelined manner and partition the operands into multiple parts of 56 or fewer bits. The hex formats have a fixed exponent size of 7 bits, whereas the binary format exponents can have 8, 11, or 15 bits.

A floating-point number can be represented in hex floating-point (HFP) format by the equation

$$X_{\text{hex}} = (-1)^{X_s} \cdot 16^{X_c - bias_{16}} \cdot X_f \qquad 0.0 \le X_f < 1.0,$$

where X_{hex} is the value of the hex format number, X_s is the sign bit, X_f is the fraction which is less than 1.0, X_c is the biased hex exponent referred to as the characteristic, and $bias_{16}$ is the hex bias, which is fixed at 64.

A floating-point number can be represented in normalized binary floating-point (BFP) format by the equation

$$X_{\text{binary}} = (-1)^{X_s} \cdot 16^{X_c - bias_2} \cdot (1 + X_f) \qquad 1.0 \le (1 + X_f) < 2.0,$$

where X_{binary} is the value of the binary format number, X_c is the biased binary exponent, $bias_2$ is the binary bias, which equals 127, 1023, or 16 383; and the 1 is implied if the biased binary exponent is nonzero. Special numbers of the IEEE 754 standard are supported in hardware without trapping to software. Even handling of denormalized input and output operands is implemented under hardware control.

Both the HFP and BFP formats are supported by a single internal format. With the use of a single internal format, the execution dataflow can be the same for both formats. The internal format is optimized for hexadecimal performance, which also allows for minimal changes to the base G4 FPU dataflow, which is hexadecimal-based. The internal format has a 56-bit fraction, which is the same as the HFP format, and the BFP short and long formats can easily be supported. BFP long format requires 53 bits of significand. In transforming the exponents from binary- to hex-based, which adjusts the significand from binary normalization to hex digit normalization, there can be up to three leading-zero bits. A 53-bit binary normalized significand requires 56 bits to represent in a hex-normalized format. Thus, both BFP and HFP long significands require at most 56 bits.

The exponent range of BFP format (15 bits) is greater than that of HFP format (7 bits). A hex unbiased exponent (16^X) can be represented by a binary unbiased exponent by multiplying it by four (2^{4X}). This shows that the hex exponent notation requires two fewer exponent bits to represent numbers of the same magnitude. If biased exponents are considered, an additional bit is needed in the hex notation to account for the slight biasing differences. The BFP format chooses its zero exponent point differently than the HFP format. The BFP format bias has the form $2^X - 1$ (i.e., 127), whereas the HFP bias has the form 2^X (i.e., 64). To account for this centering point of the exponent range being different, an additional exponent bit is added. This bit also helps for some of the range of overflow and underflow of intermediate results but does not completely account for the full range of intermediate results. Thus, a 14-bit exponent format is chosen for the internal format (15 − 2 + 1). The following equation shows the format:

$$X_{\text{internal}} = (-1)^{X_s} \cdot 16^{X_c - bias_{\text{internal}}} \cdot X_f \qquad 0.0 \le X_f < 1.0,$$

where X_{internal} is the value of the internal format number. The internal bias has a fixed value of 8192, which is similar to the HFP format. The transformation of a number from the binary format to the internal format is discussed in detail in the section of this paper on format conversion.

G5 FPU dataflow

The G5 FPU fraction dataflow is illustrated in **Figure 2**. As mentioned earlier, this dataflow is essentially that of the previous G4 FPU, with some additional hardware to support the BFP architecture. The dataflow is five stages deep and supports computation on 56-bit hex-format fractions. The dataflow contains a floating-point multiplier which utilizes a radix-8 Booth encoding algorithm [11–15]. The dataflow also contains a 120-bit carry-propagate adder which is utilized in both multiplication and addition instructions.

The first stage of the pipeline contains the input registers, the input multiplexing, the Booth encoding, and the formation of the 3× multiple of the multiplier, as well as the compare and swap and aligner for the adder. The input registers receive the input operands from the FPRs, GPRs, and operand buffers, and the output of FPU dataflow. The MAL, AAL, and BL bypass multiplexors are used to allow wrapping of the output of the dataflow back into the first stage of the dataflow. This stage also contains the format-conversion hardware for binary operands. The details of this hardware are discussed subsequently.

The second stage of the dataflow contains the $3X$, X, and Booth encode registers which are used to stage the intermediate results of the multiplication process. The Booth multiplexors and the 19-to-2 counter tree are also in this stage. The Booth multiplexors are controlled by the Booth encode register to select the proper multiple of the multiplicand, X. These 19 multiples are then reduced to two partial products in the counter tree.

The third dataflow stage contains the carry and sum registers, which are 120 bits in length. These registers receive either the final two partial products of a multiplication operation or the two operands for an addition operation which originate from the first add-cycle hardware. These two registers drive the input to the 120-bit carry-propagate adder. The output of the adder is driven to the FC1 and FC3 registers.

The fourth stage of the dataflow comprises the FC1 register, the leading-zero detect (LZD), the post-normalizer with sticky detection, and the store rotator. The FC1 register drives 117 bits to the post-normalizer for

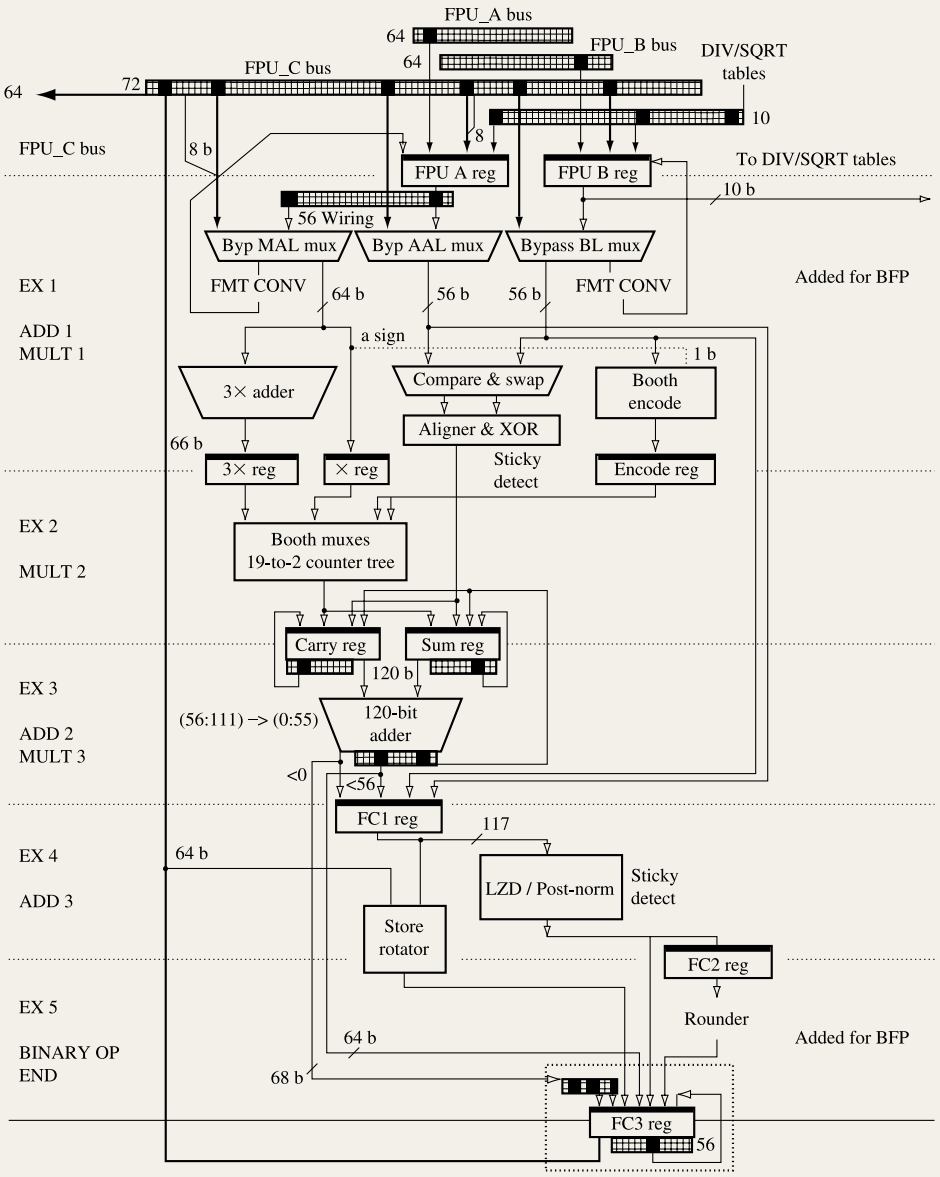

Figure 2

G5 FPU fraction dataflow.

the third addition execution cycle. The post-normalizer determines the shift amount by performing a leading-zero detect 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 store rotator is a 64-bit-wide byte rotator, and its input is driven by a multiplexing of either the FC1 or FC3 registers. The store rotator is used to byte-align the source of the store within a doubleword, 64-bit memory boundary. The FC3 register input is used for early data dependency resolution with the previous instruction.

The fifth stage of the dataflow consists of the FC2 register and the binary rounder. The FC2 register drives to a binary rounder, which performs the rounding function for the binary architecture. The rounder also is used to convert the internal hex data format back to a binary format. The binary rounder can also 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. This stage is used by all binary format instructions; hex division and square root also use its binary shift capability. The rounder output is connected to the FC3 register.

The FC3 register receives the result of the arithmetic computation and drives the results to the register files or back to the FPU dataflow by way of the FPU_C bus.

The execution of instructions in this pipeline is shown in **Figure 3**. 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 corresponds to the rounder cycle, and FS6 corresponds to the write cycle. The primary difference between this diagram and one for the G4 FPU is the addition of feedback paths in the FS1 cycle for binary format conversion and the addition of the FS5 stage for rounding. The pipeline for hex operations remains the same as in the G4 FPU, and binary operations add two pipeline stages to avoid major design changes.

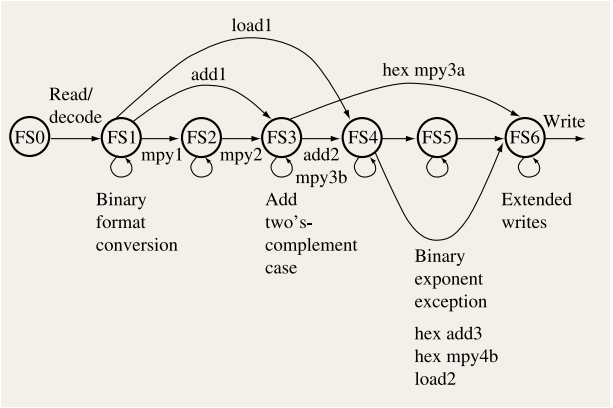

Figure 3

Pipeline state diagram.

• *G5 FPU dataflow modifications for BFP support*

As mentioned earlier, the G5 FPU evolved directly from the G4 FPU, which implements only HFP format and is optimized for long operands. The G4 FPU performs many common operations, such as shifting to hex digit boundaries in support of the HFP format. To utilize this dataflow without affecting critical timing paths and area, most operations for the G5 continue to operate on hex digits. These operations include alignment, leading-digit detection, and post-normalization. This presents the challenge of implementing the binary floating-point instructions with minimal impact to the existing dataflow. The overall implementation strategy of transforming the BFP into hex data also has an impact on the HFP format. The resulting hex internal format must differ from the architected HFP format to allow for the greater range of binary floating-point numbers. The hex internal exponent is 14 bits versus the 7-bit HFP format, requiring a transformation for even HFP format to be represented in this internal format. However, this operation will be shown to be trivial.

Once the data is converted to the internal format, most of the computation is carried out as though the data were a hex-based number. A minimal number of changes during computation are necessary to satisfy the increased precision required by the binary floating-point facility in comparison to its hex-based counterpart. At the end of the dataflow, the result data must be transformed back into its binary representation. This process entails being able to normalize the result within the hex boundary and converting the hex-based exponent back to a binary exponent. The dataflow changes for binary support are designed not to affect the execution of the hex-based input data.

Given our implementation strategy, six additional dataflow functions for BFP support are identified:

1. Format conversion of hex-architected data to internal format and back.
2. Format conversion of binary-architected data to internal format and back.
3. Loss of precision detection or sticky detection.
4. Binary rounding.
5. Binary floating-point special number handling.
6. Binary exception detection and handling.

The changes to the dataflow are highlighted in Figure 2. At the top of the dataflow are two format converters, which receive data from the two input registers. The primary responsibility of the format converters is to transform the binary-based data into the hex-based internal exponent and fraction formats. These format converters also perform special binary number detection, which is used to control the execution of instructions when special numbers are encountered as input data. Farther down in the dataflow, sticky detection is added to the fraction aligner. This logic detects loss of precision during the pre-alignment operation that is typically required during the execution of addition and subtraction. Sticky detection is also added to the fraction post-normalizer, since this process can also produce a loss of precision. In the final stage of the pipeline, a binary rounder is added to perform the rounding function and to transform the result from the hex-based internal format back to the

binary format. The rounder also has the capability of creating special number results. Finally, binary exceptional conditions such as exponent overflow and underflow are detected through the combined use of the post-normalizer and the rounder.

Format conversion of hex data

The format conversion from hex-based architected format to internal hex format is trivial and can be performed within the input multiplexor of the input operand register. The only difference between the hex-architected and hex-internal format is the bias and widths of the exponents. The transformation of exponents from architected to internal is similar to a sign extension, as shown by the following equations:

$$16^{(Exponent_{\text{internal}} - bias_{\text{internal}})} = 16^{(Exponent_{\text{architected}} - bias_{\text{architected}})};$$

$$16^{(Exponent_{\text{internal}} - 8192)} = 16^{(Exponent_{\text{architected}} - 64)};$$

$$Exponent_{\text{internal}} - 8192 = Exponent_{\text{architected}} - 64;$$

$$Exponent_{\text{internal}} = Exponent_{\text{architected}} + 2^{13} - 2^{6}.$$

The term $2^{13} - 2^{6}$ is a string of ones from bit 1 of the internal exponent representation (bit 0 being the most significant bit) to bit 7. Bit 7 is the location of the most significant bit of architected exponent, as shown by the following:

$$\begin{aligned} Exponent_{\text{internal}} &= 0000000E_0E_1E_2E_3E_4E_5E_6 \; + \\ &\quad 01111111000000_2 \\ &= E_0\overline{E}_0\overline{E}_0\overline{E}_0\overline{E}_0\overline{E}_0\overline{E}_0\overline{E}_0\overline{E}_0E_1E_2E_3E_4E_5E_6 . \end{aligned}$$

Only the most significant bit of the architected exponent participates in the addition. This range-expansion operation is similar to a sign-extension operation in which the most significant architected exponent bit becomes the most significant internal exponent bit followed by its complement replicated seven times. This is a simple operation to perform, requiring only an inverter and additional fan-out on the most significant exponent bit.

This function of taking a signal and placing its true phase in bit 0 of the exponent followed by its complement replicated several times will be seen to be useful for other format conversions. Therefore, it is defined to be *SIGNEXT*(X, Y), with *SIGNEXT* representing its similarity to sign extension; let X equal the number of bit positions occupied by the sign-extension encoding, and let Y equal the input signal. In the case above, the operation can be described as *SIGNEXT*(8, E_0) $\|$ $E(1{:}6)$. In a further simplification, we let *SE*(Y) represent the overall function of sign-encoding the most significant bit, followed by a concatenation with its least significant bits. In this case, the following is the reduction in notation, *SE*[$E(0{:}6)$], or *SE*(E). This simplifies the notation for further discussion.

To convert from internal format back to hex-architected format is also simple. It can be shown that an internal exponent within the valid range of HFP format will have the exact same relationship; the most significant bit is followed by its complement repeated in seven bit positions. Thus, the resulting most significant bit of the HFP exponent can be created by using the most significant internal exponent or complementing any of the next seven exponent bits. It is preferred to invert the least significant of these seven exponent bits because it gives the correct result for exponent overflow or underflow. For this case, the resulting exponent should wrap, and only the least significant of these upper exponent bits is affected by an overflow or underflow. Note that the least significant six bits of the internal exponent are directly equal to the least significant bits of the hex-architected exponent. To simplify notation, let *RSE*(Y) be the reverse sign encoding to the target format. Thus, the transformation back to hex-architected format is also trivial.

Format conversion of binary architected to hex internal

There are two format-convert macros in the first stage of the dataflow. These format converters are used to transform the input operand from the architected binary format to the internal format. The format conversion from an architected binary-based format to the internal hex-based format is not trivial, since both the fraction and exponent require modification. The transformation of the binary exponent to the hex exponent is equivalent to dividing the binary exponent by 4, which is equivalent to a right shift of 2. The fraction is then shifted by an amount equal to the remainder of the division. The following illustrates this transformation if the unbiased exponents are considered rather than the biased exponent:

$$(1 + X_{\text{f}}) \cdot 2^{0} \cdot 2^{4X} \Rightarrow (0.0001_2 \,\|\, X_{\text{f}}) \cdot 16^{X+1},$$

$$(1 + X_{\text{f}}) \cdot 2^{1} \cdot 2^{4X} \Rightarrow (0.001_2 \,\|\, X_{\text{f}}) \cdot 16^{X+1},$$

$$(1 + X_{\text{f}}) \cdot 2^{2} \cdot 2^{4X} \Rightarrow (0.01_2 \,\|\, X_{\text{f}}) \cdot 16^{X+1},$$

$$(1 + X_{\text{f}}) \cdot 2^{3} \cdot 2^{4X} \Rightarrow (0.1_2 \,\|\, X_{\text{f}}) \cdot 16^{X+1},$$

where the binary exponent equals $4X + y$ and y is between 0 and 3.

However, in reality the conversion is more difficult, since the exponents are biased by unequal amounts. The following equations illustrate the conversion for normalized nonzero BFP numbers when biased exponents are considered where r = *binary biased exponent mod* 4, the binary bias is represented by $2^{N} - 1$, and X_{if} is the internal hexadecimal format fraction.

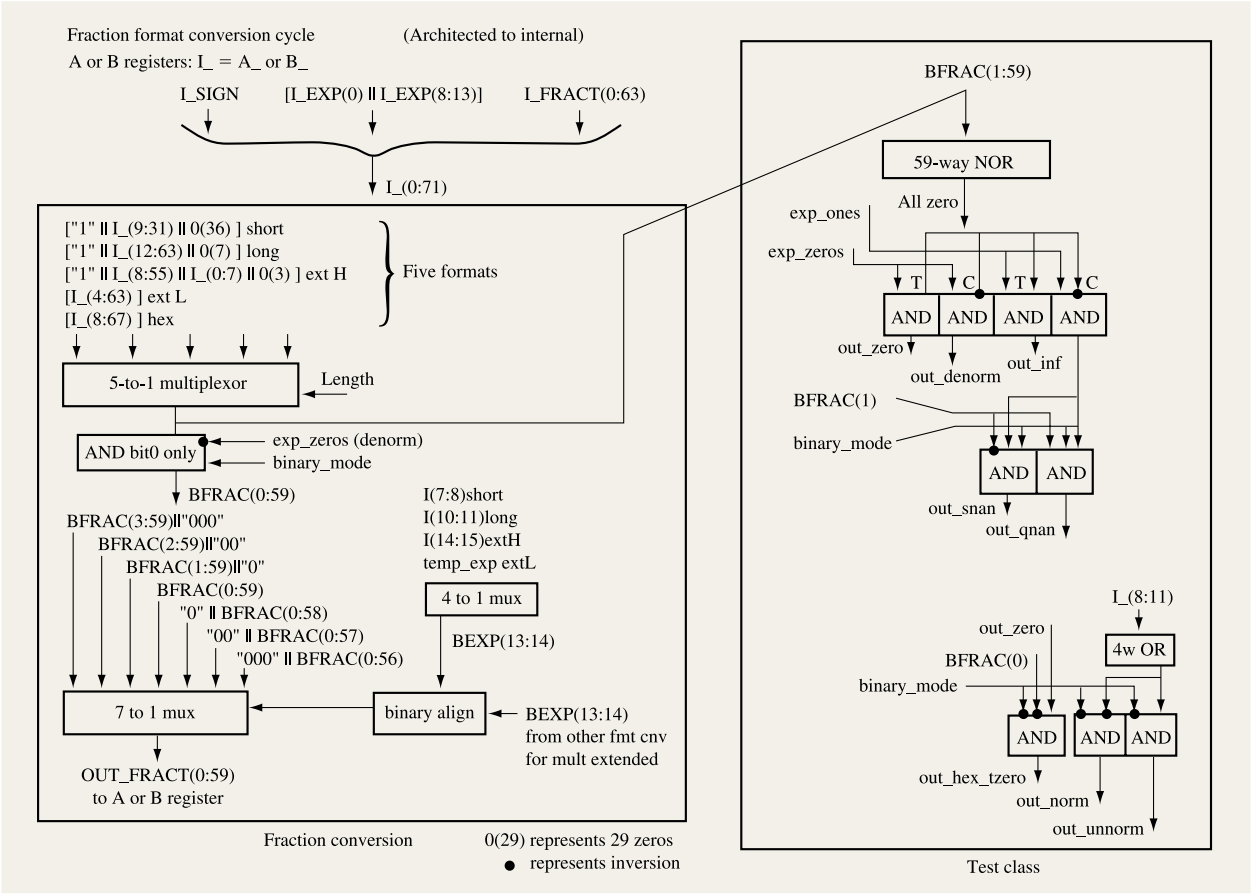

Figure 4

G5 FPU fraction format conversion.

$$(1 + X_f) \cdot 2^{binary\ exp.-bias} \Rightarrow X_{if} \cdot 16^{X-8192} \text{ and } 0.0001_2 \le X_{if} < 1.0$$

$$= (1 + X_f) \cdot 2^{[binary\ exp.-(2^N-1)]}$$

$$= (1 + X_f) \cdot 2 \cdot 2^{(binary\ exp.-2^N)}$$

$$= (1 + X_f) \cdot 2 \cdot 16^{[(binary\ exp/4)-2^{(N-2)}+8192]-8192}$$

$$= (1 + X_f) \cdot 2 \cdot 2^r \cdot 16^{[\lfloor binary\ exp./4 \rfloor-2^{(N-2)}+8192]-8192}$$

$$= [(0.001 \parallel X_f) \cdot 2^r] \cdot 16^{[\lfloor binary\ exp./4 \rfloor+1+2^{13}-2^{(N-2)}]-8192}.$$

The $2^{13} - 2^{(N-2)}$ results in a sign-extension term of $SIGNEXT\{[14 - (N - 2)], Y\}$. Below is a table for these conversions, where $\gg$ represents a right-shift operation:

$$(1 + X_f) \cdot 2^{binary\ exp.-bias} =$$

$$\begin{aligned}
&(0.0001 \parallel X_f) \cdot 16^{[SE(binary\ exp.\gg 2)+2]-8192} && \text{for } r = 3,\\
&(0.001 \parallel X_f \parallel 0) \cdot 16^{[SE(binary\ exp.\gg 2)+1]-8192} && \text{for } r = 0,\\
&(0.01 \parallel X_f \parallel 00) \cdot 16^{[SE(binary\ exp.\gg 2)+1]-8192} && \text{for } r = 1,\\
&(0.1 \parallel X_f \parallel 000) \cdot 16^{[SE(binary\ exp.\gg 2)+1]-8192} && \text{for } r = 2.
\end{aligned}$$

Figure 4 displays the design of the fraction format conversion macro. In the first stages of the fraction format converter, the format is multiplexed to separate the exponent bits from the fraction bits. The multiple fraction formats are multiplexed, and the implied one is added if the exponent is not all zeros. The fraction is then shifted depending on the least significant two bits of the exponent.

Figure 5 is a high-level illustration of the exponent format conversion macro. First the exponent is sign-extended to a 16-bit format, in which the least significant two bits are examined to determine the fraction shift amount and the exponent increment amount. These bits can be overridden for extended-precision operations such as binary multiply extended (MXBR). The output of the first format multiplexor drives the adder, and 0, 1, or 2 is added to the exponent upper 14 bits. In parallel, the exponent is examined to detect the special case in which the exponent equals all ones or all zeros.

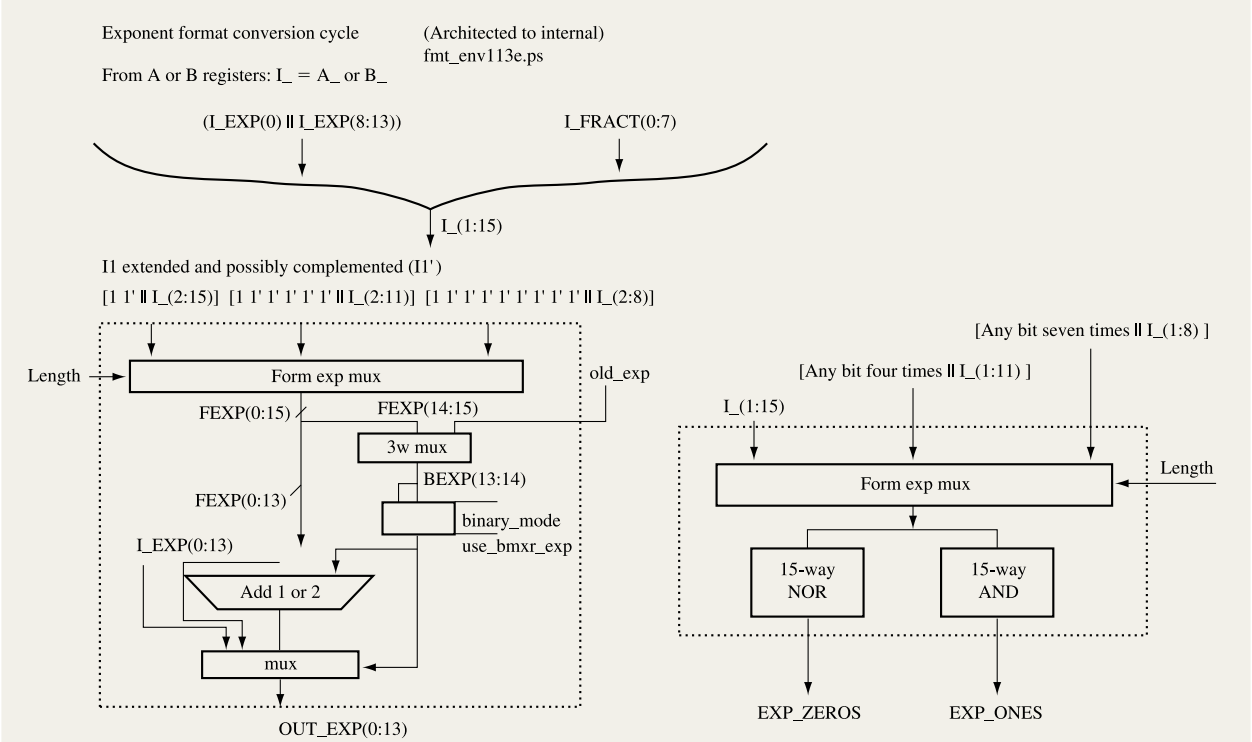

Figure 5

G5 FPU exponent format conversion.

Format conversion of hex internal to binary architected

The conversion back to binary architected format from hex internal is performed in the rounder and involves the following formulation:

$$X_{\text{if}} \cdot 16^{exponent_{\text{internal}}-8192} \Rightarrow (1 + X_{\text{f}}) \cdot 2^{exponent_{\text{binary}}-b'},$$

$$X_{\text{if}} = X_{\text{if}} \cdot 2^{r} \cdot 2^{-r},$$

$$X_{\text{if}} = (1 + X_{\text{f}}) \cdot 2^{-r};$$

$$\begin{aligned} X_{\text{if}} \cdot 16^{exponent_{\text{internal}}-8192} &= (1 + X_{\text{f}}) \cdot 2^{-r} \cdot 2^{4 \cdot exponent_{\text{internal}} - 4 \cdot 8192} \\ &= (1 + X_{\text{f}}) \cdot 2^{-r} \cdot 2^{4 \cdot exponent_{\text{internal}} - 2^{15}} \cdot 2^{(2^N - 1)} \cdot 2^{-(2^N - 1)} \\ &= (1 + X_{\text{f}}) \cdot 2^{-r} \cdot 2^{4 \cdot exponent_{\text{internal}} - 2^{15} + 2^N - 1 - (2^N - 1)} \\ &= (1 + X_{\text{f}}) \cdot 2^{4 \cdot exponent_{\text{internal}} - 2^{15} + 2^N - (r+1) - b'}; \end{aligned}$$

$$exponent_{\text{binary}} = 4 \cdot exponent_{\text{internal}} - 2^{15} + 2^N - (r + 1);$$

$$exponent_{\text{binary}} = RSE[4 \cdot exponent_{\text{internal}} - (r + 1)].$$

Exponents within the range of representable numbers are guaranteed to have E_0 followed by $\overline{E_0}$ for the bit locations from the most significant exponent position of the internal format to the bit weighted by 2^N. Thus, the reverse sign extension can easily be accomplished by just wiring the most significant bit of the hex internal exponent to the most significant bit of the binary architected format. Or the subtractor width can be minimized to the target exponent length and the most significant bit can be inverted, since it will be one of $\overline{E_0}$ bits. Here is the mapping between formats for different binary normalizations within the hex format, or, stated another way, various values of r:

$$(0.0001 \parallel X_{\text{m}}) \cdot 16^{exponent_{\text{internal}}-8192} \Rightarrow (1 + X_{\text{f}}) \cdot 2^{RSE(4 \cdot exponent_{\text{internal}} - 5) - b'},$$

$$(0.001 \parallel X_{\text{m}}) \cdot 16^{exponent_{\text{internal}}-8192} \Rightarrow (1 + X_{\text{f}}) \cdot 2^{RSE(4 \cdot exponent_{\text{internal}} - 4) - b'},$$

$$(0.01 \parallel X_{\text{m}}) \cdot 16^{exponent_{\text{internal}}-8192} \Rightarrow (1 + X_{\text{f}}) \cdot 2^{RSE(4 \cdot exponent_{\text{internal}} - 3) - b'},$$

$$(0.1 \parallel X_{\text{m}}) \cdot 16^{exponent_{\text{internal}}-8192} \Rightarrow (1 + X_{\text{f}}) \cdot 2^{RSE(4 \cdot exponent_{\text{internal}} - 2) - b'}.$$

Sticky detection

Sticky detection is necessary with the inclusion of the new rounding modes of the S/390 binary floating-point

architecture. The binary architecture dictates that the operation must appear as if infinite precision is maintained throughout the execution. Since it is physically impractical to maintain infinite precision, a sticky bit is utilized during execution to represent any loss of significance. When a number is represented in a binary format, shifting out any nonzero bit is a loss of significance. However, since our internal format is hex-based, all shifting is on a hex-digit boundary, and loss of significance is detected on a hex-digit boundary. Loss of significance can occur when significant bits of an operand are not carried throughout the entire calculation. In G5 FPU dataflow, this can occur in three places: during the pre-alignment process and during the truncation of the intermediate result which occurs in the post-normalizer and the rounder.

The pre-aligner is used in the execution of addition and subtraction instructions. The fraction of the operand with the smaller exponent is aligned to the operand with the larger exponent. Since the dataflow has a width which is much smaller than the maximum exponent separation, there is the possibility that some significant digits of the smaller operand might be lost during the right shifting of the smaller operand. The infinitely precise result of the addition might be slightly larger than the result of the reduced-precision dataflow. However, the infinitely precise result will be rounded to a finite-precision result. The loss of these shifted bits is remembered in a sticky bit so that they can participate in the rounding process operation which yields the final result. The generation of the pre-aligner sticky bit is the logical OR operation of all of the bits that are lost in the pre-aligner.

Figure 6 illustrates the pre-aligner sticky detection. SMALL_OUT(0:55) is the smaller operand of the addition and comes from the fraction swapper. EXP_DIFF(0:3) is the exponent difference as calculated by the exponent comparison. So as not to create a critical path, the sticky-bit detection is performed in parallel with the alignment of the fraction. Each digit of SMALL_OUT is checked to see whether it is nonzero; this forms the signal Sticky_Digit(0:13). When the exponent difference is determined, it is converted to a 14-bit mask, Sticky_Mask(0:13), which is input along with the corresponding sticky-digit information to a 2-input by 14-bit logical AND gate. Then a logical OR of the resulting vector shows whether there is any loss of significance during this addition alignment cycle.

Loss of significance can also occur when the infinitely precise result is truncated to the target format. The rounder requires the sticky-bit and guard-bit information to determine whether the intermediate result requires incrementation. To achieve cycle time in the rounder, it is better to perform the majority of the stickiness calculation of the intermediate result (which is lost to truncation) in

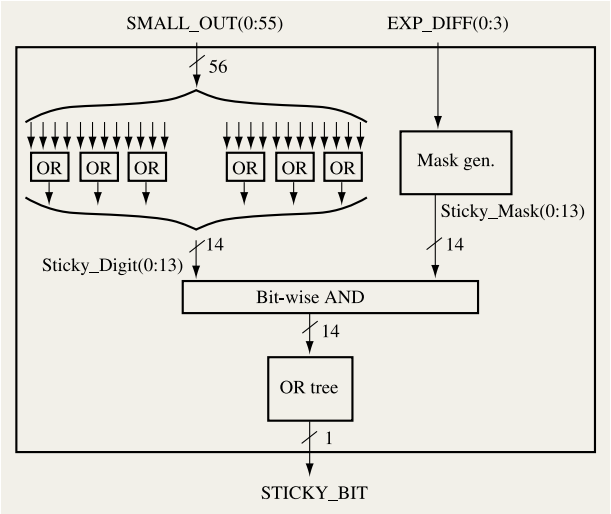

Figure 6

G5 FPU pre-aligner sticky calculation.

the post-normalizer, the cycle prior to the rounder. The post-normalizer performs a leading-zero detect on the intermediate result, generates the corresponding shift amount, and then left-shifts the result by that amount. In parallel with the shifting of the result, the post-normalizer sticky logic receives the shift amount and the target format length to determine which bits will participate in the sticky calculation. **Figure 7** illustrates the high-level hardware design of the post-normalizer sticky calculation [16]. Effectively, the sticky bit is determined for all of the possible shift amounts; once the shift amount is known, selection of the sticky information is similar to multiplexing of the fraction for normalization. The first level of logic performs a hex-digit sticky calculation which is a 4-bit logical OR for each hex digit of the result from digit 7 to digit 30. Then several groups of sticky detects are created for the different binary formats. The binary short format creates six 4-bit sticky groups, S0, S4, S8, S12, S16, and S20. Each group represents the four possible degrees of stickiness for the four possible shifts within the group of four digits. For example, S0(0) represents the stickiness of digits 7 through 30, S0(1) the stickiness of digits 8 through 30, and S0(2) and S0(3) the stickiness of digits 9 through 30 and 10 through 30, respectively. The binary long format has five 4-bit sticky groups, and the binary extended format has just one sticky group. The 4-bit format-specific sticky groups are then selected on the basis of the target format of the result and are then multiplexed on the basis of a coarse shift amount from the leading-zero detection in the post-normalizer which is available earlier than the lower bits of the shift

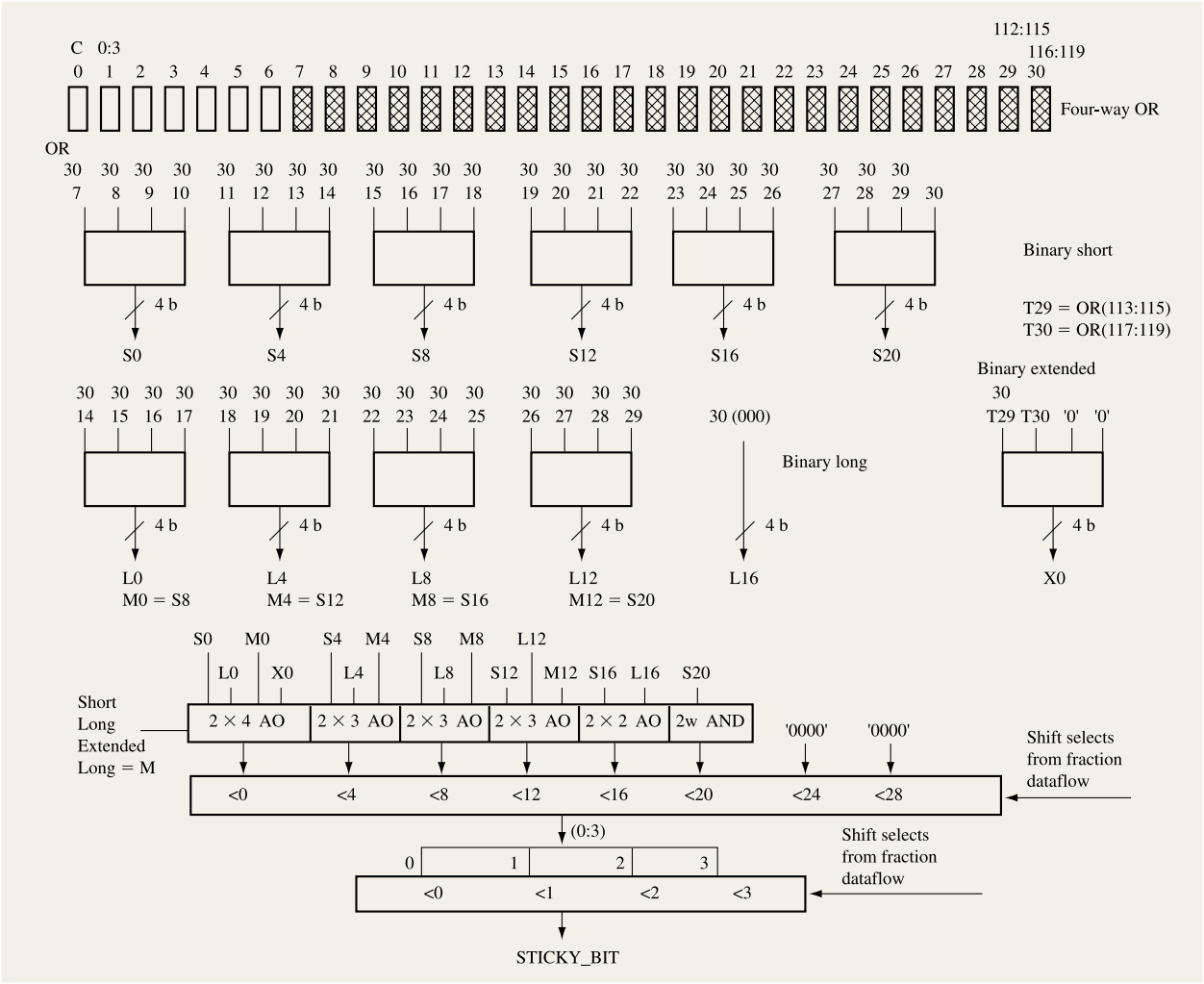

Figure 7

G5 FPU post-normalizer sticky calculation.

amount. Then, in the second multiplexing level, a finer sticky selection is done on the basis of the least significant two bits of the shift amount.

Rounder

The rounder macro has two main responsibilities: The first is to perform the binary rounding as specified by the S/390 binary floating-point architecture. The second is to convert the internal hex data format back to the binary architected format.

The S/390 binary floating-point architecture specifies five rounding modes: round to nearest, round toward zero, round toward positive infinity, round toward negative infinity, and biased round toward nearest. The first four are controlled by bits 30 and 31 of the 32-bit floating-point control word (FPC). There are a few instructions such as conversion to and from fixed-point format and the divide-to-integer instructions which can explicitly use any of the five rounding modes overriding the FPC rounding mode.

A high-level illustration of the rounder macro is found in **Figure 8**. The rounder calculates an incremented value of the result in parallel with the determination of whether the result should be truncated or incremented. Since the incremented value of the result may require an increment to the exponent, an incremented value of the exponent is also precalculated. The rounding action is dependent on the least significant digit, the guard digit, the pre-aligner's sticky bit, the normalizer's sticky bit, and a sticky bit within the least significant hex digit, which is determined within the rounder itself.

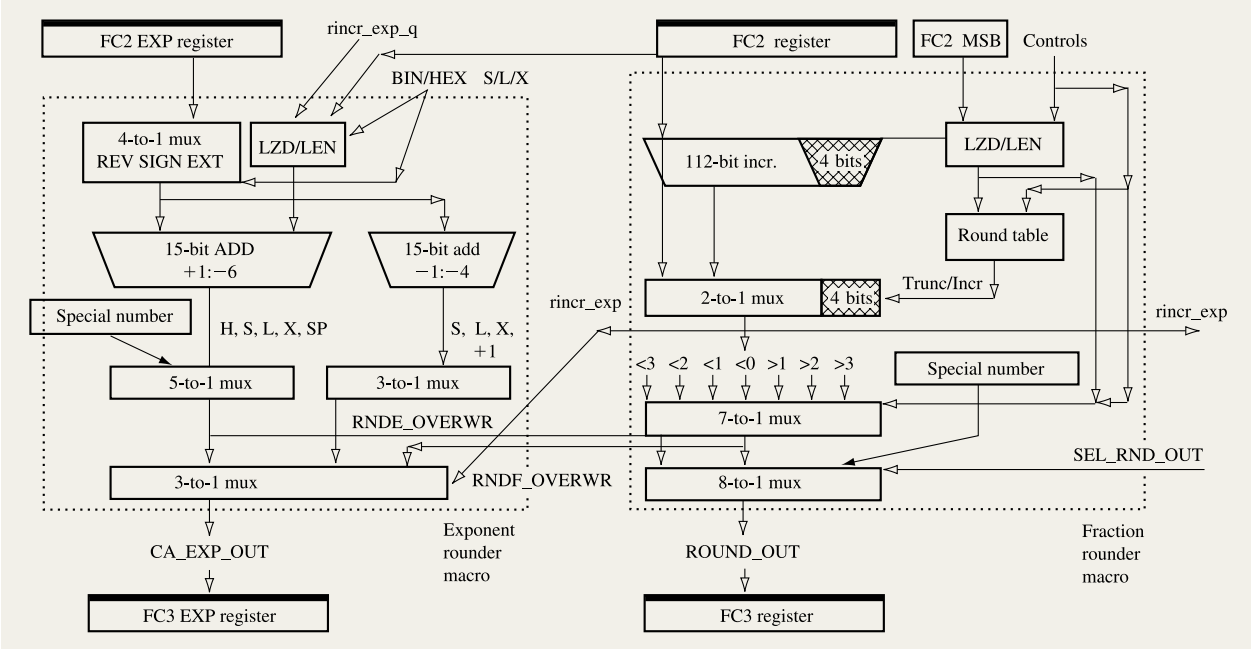

Figure 8

G5 FPU rounder.

The rounder fraction dataflow also has a multiplexor to shift right or left up to 3 bits. This is used to perform the binary alignment of the hex internal fraction. The final fraction 8-to-1 multiplexor is used to create multiple output formats such as binary short, long, extended first part, and second part. It can also select a forced special number such as zero or infinity. The rounder exponent dataflow converts the internal exponent to binary by reverse-sign-extending it and then adding a constant to it. Note that the expected exponent is calculated and it is also incremented by 1 in case the rounding of the fraction results in incrementing the exponent. There are also buses between the exponent and fraction rounder macros to support overwriting the fraction with exponent data and vice versa. Thus, the rounder performs rounding and format conversion of hex internal format to binary architected format.

Binary floating-point special number handling

The S/390 binary floating-point architecture specifies four special numbers: zero, infinity, not-a-number (NaN), and denormalized numbers. These special numbers are represented uniquely with specific exponent and/or fraction characteristics. The S/390 binary floating-point facility requires that the floating-point dataflow be able to operate upon these special numbers and also be able to produce these special numbers. Most arithmetic instructions are defined by the S/390 binary floating-point facility to treat these numbers specially. For example, NaN input operands are typically preserved throughout execution. The floating-point dataflow contains hardware to perform special number detection at the top of the dataflow within the format converters. This early detection allows the execution of the instruction to be controlled correctly.

Figures 4 and 5 show the format-conversion logic that is utilized for input special number detection. The exponent is checked for the all-zeros or all-ones case, and this information is sent to the fraction format conversion macros, where it is used with the results of a ones detection on the fraction field to determine whether the input operand is a zero, infinity, NaN, or denormalized number. In most cases when the input is an infinity or a NaN, the operation is treated as an effective load in which one of the input operands is forwarded through the dataflow.

Machines in other architectures such as PowerPC* have been able to modify the internal register file to keep special number information bits. This is possible in a load/store-type architecture, but is much more difficult in the S/390 architecture, which is optimized for register and memory (RX-type) input operands. The critical path of loading in data from memory would be affected by this type of design. Also, S/390 allows six formats for numbers,

and the loads and stores are generic and do not specify the format type but only the length of the operand. To avoid any complex conversions between internal data types, special number detection results are not saved in the register file. Instead, special number detection is performed on any input binary data.

The dataflow is also capable of producing special numbers as a result of execution. The operation on certain special input operands is defined to result in a special number. For example, multiplication of zero and infinity is defined to produce a result which is a default not-a-number, dNaN. Also, the result of rounding may create a special number result. The largest normalized number, NMax, or the smallest denormalized number, DMin, may be produced when the result prior to rounding is within certain ranges. These special numbers are produced by the rounder macro. The rounder macro is capable of producing infinity, zero, NaNs, NMax, and DMin. The creation of these numbers can either be forced explicitly by the floating-point control or created from the rounding table.

A binary floating-point calculation can also result in a denormalized result, which provides a means for gradual underflow. A denormalized floating-point number is a tiny number whose exponent is smaller than the smallest representable exponent. Unlike binary normalized floating-point numbers, denormalized numbers do not have an implied 1. The dataflow detects exponent underflow in the post-normalizer and rounder, as is subsequently discussed in detail. The dataflow does not include special hardware to denormalize the result, but instead reuses existing shifting capabilities in the normalizer. Denormalization presents an additional pipeline sequencing complexity, since the need for denormalization is not known until the final stage of the dataflow, and the data must be wrapped around to the top of the dataflow. This action must be performed without interfering with other instructions which may also coexist simultaneously in the other stages of the dataflow.

In order to minimize this complexity, a simple control method is used to produce a denormalized result. This control method utilizes two different control sequences. The first is chosen if there are no other instructions in the dataflow. The second is chosen if other instructions are detected inside the floating-point dataflow or possibly entering the dataflow when the need to denormalize a result arises.

If there are no other instructions in the dataflow,

1. The normalizer indicates a possible exponent underflow, and the exponent and fraction are held in the FC2 registers.
2. The rounder confirms the exponent underflow condition, and the floating-point dataflow accepts no further instructions until the current instruction completes.
3. The rounder then truncates the fraction and passes the resulting fraction and exponent to the FC3 register.
4. The fraction is moved to the B input register at the top of the dataflow, and the exponent is passed to the controls.
5. The fraction is then moved to the FC1 register.
6. The controls examine the exponent and perform an effective right-shifting of the fraction in the normalizer. The denormalized fraction is then moved to the FC2 register.
7. The fraction in the FC2 register is then rounded and the exponent is forced to all zeros. The result is placed in the FC3 register.
8. The denormalized result is written to the FPR, and the floating-point dataflow begins accepting new instructions.

If there are other instructions in the dataflow,

1. The normalizer indicates a possible exponent underflow, and the exponent and fraction are held in the FC2 registers.
2. The rounder confirms the exponent underflow condition. The presence of other instructions is detected in the floating-point dataflow.
3. The floating-point unit then forces an interrupt request for serialization. The execution of the denormalizing instruction and all other instructions in the dataflow are canceled.
4. The denormalizing instruction is then decoded and executed alone following the sequence outlined above. Subsequent instructions are not dispatched to the FPU until the denormalizing instruction completes.

The instruction unit is able to issue instructions in a nonoverlapped mode. This method preserves sequential execution order of the instruction stream. It is robust in that the normal path for execution does not require auxiliary registers, and only the control-issuing and canceling mechanisms are modified. The feedback path, in taking the result from the rounder back to the shifter, uses existing paths that are used for data dependencies between instructions. This mechanism ensures that a feedback path even to the first execution pipeline stage is acceptable, since there will be no other instructions executing in the pipeline if the wrap is allowed to take place.

Exception detection and handling

The S/390 binary floating-point facility defines five types of exceptions that can result from binary execution: invalid operation, divide by zero, exponent overflow, exponent

Table 1 Latency and throughput performance of common floating-point instructions.

Instruction	*Execution cycles*					
	Short		*Long*		*Extended*	
	L	*T*	*L*	*T*	*L*	*T*
Load	2	1	2	1	—	—
Store	3	1	3	1	—	—
Add HFP	3	1	3	1	12	12
Add BFP	5	2	5	2	20	20
Multiply HFP	3	1	3	1	16	16
Multiply BFP	6	2	6	2	27	27
Multiply/Add BFP	13	13	18	18	—	—
Divide HFP	20, 27	20, 27	24, 30	24, 30	135	135
Divide BFP	23, 27, 30	23, 27, 30	27, 31, 34	27, 31, 34	135	135
Sqrt HFP	27, 34	27, 34	36, 43	36, 43	129, 131	129, 131
Sqrt BFP	27–36	27–36	37–46	37–46	133, 135	133, 135

underflow, and inexact result. Each of these exceptions is maskable to cause a trap through five bits of mask contained in the floating-point control (FPC) word. If the exception does not cause a trap, execution is completed, and a corresponding flag bit in the FPC is set to record the occurrence of the exception. Invalid operation and divide-by-zero exception detection rely on the special number detection, which is performed in the format converters. Exponent overflow and underflow are detected with a combination of the post-normalizer and the rounder. The inexact result exception is detected in the rounder and utilizes the sticky information collected during execution.

Exponent underflow and overflow detection is a two-step process. The first step in this process is performed by the post-normalizer. Since the leading-zero-detection logic in the post-normalizer operates on hex digits, the binary alignment within the leading digit is not known. This is necessary for performing accurate exponent underflow detection and is not determined until the rounder. Since exponent overflow detection is performed on the rounded result, it is also finalized in the rounder. The post-normalizer creates multibit signals for the different binary exponent decrement values and sends these to the rounder. The rounder then uses these signals along with the binary normalization of the leading digit of the post-normalized fraction to precisely determine underflow and overflow.

The post-normalizer also outputs a signal to the controls which indicates the potential for overflow or underflow because of the maximum binary shift and incrementation condition in the rounder. The controls use this signal to cause the following rounder cycle to stall. Stalling the rounder cycle is necessary for overflow and underflow conditions, because special results may have to be generated. For example, in the event of underflow, the rounder may be required to produce a denormalized result if a trap does not occur. The first rounding cycle is used to confirm that a definite overflow or underflow condition exists. Subsequent rounder cycles are then utilized to produce the special result.

Performance

Table 1 shows the performance of common floating-point instructions. Both the latency (L) of execution and the throughput (T) are listed for HFP and BFP formats. Hex short and long, add and multiply are fully pipelined three-cycle latency operations. The binary short and long operations are slower because of the format-conversion and rounding cycles. The throughput of binary operations is also lower than hex because of the bus-wiring strategy of the format converter. The wiring of the input operand buses is timing-critical, and it was decided not to fan them out to multiple registers. This resulted in A and B registers driving the format converters rather than another staging register. The cost is one cycle of throughput to BFP operations so as not to affect the cycle time.

Another interesting design decision is the implementation of multiply-then-add, which is sometimes referred to as fused multiply add. From the performance figures it would appear that this function is not useful, but it was designed as a preliminary step to allow compilers to include this as an optional instruction. Also, its performance is faster for the case in which only one rounding can be tolerated. The equivalent operation would require a multiply long to extended, followed by an add extended, and a load rounded back to long format. The latency of multiply-then-add is much faster than this,

but it is not faster than a multiply long followed by an add long.

The execution times are also given for division and square root, which use a Goldschmidt algorithm [17–24] for short and long operands and a restoring radix-2 scheme [14, 15] for extended operands. This has been detailed for the G4 FPU [4, 24]. Several numbers or a range is given for these operations, since the latency is variable. It is dependent on the guard-bit combinations of the intermediate result; in most cases, extra cycles to perform a remainder comparison are eliminated. Also, certain exponent values close to overflow or underflow may take additional cycles.

The binary floating-point implementation gives reasonable performance, a little less than half the equivalent in HFP format, for the arithmetic operations. However, in many workloads the performance of the arithmetic operations is overshadowed by the loads, stores, and branches, and in this case the performance of BFP is much closer to that of HFP. The BFP implementation is intended to replace a software implementation of BFP for such applications as Java**, and for this case it is substantially faster, approximately 100 times faster. This is to be expected for a hardware implementation compared to a software implementation, especially when the basic data types are not available in hardware. In the future, it is expected that BFP performance will be as important as or more important than HFP, and this will determine how future machines will be designed. Currently, the main workload is still in HFP format; for this reason the BFP design was added in a manner that would not affect the HFP performance.

Summary

The IBM S/390 G5 Parallel Enterprise Server is the first S/390 machine to implement the IEEE 754 standard in hardware. The S/390 G5 FPU adds the functionality of the 121 new instructions to support the IEEE 754 standard. Because the development cycle of the G5 microprocessor was very short, a full redesign of the FPU was not possible. Instead, the G5 is an incremental change to the G4 microprocessor core. Most of the G4 FPU custom dataflow macros were reused without modification. Given this design limitation, a strategy of using an internal format that has hexadecimal properties such as the traditional hexadecimal format was key to minimizing design changes. This new internal format supports six floating-point formats—three formats of both hexadecimal and binary. The few macros that were added to the design were the format converter macros and the rounder macro. The custom macros requiring changes included the pre-aligner and normalizer to support sticky-bit detection. The G5 FPU implements almost all operations in hardware, including some of the most difficult subtleties of the IEEE 754 standard. This includes an interesting hardware mechanism for denormalizing an intermediate result which is very robust and simple. Also, the G5 FPU implements quadword precision hex and binary arithmetic in hardware, including all special number handling. Given these aspects of handling special cases and quadword precision operands in hardware, which are not commonly found in other processors, the G5 FPU is arguably the most complete hardware solution to implementing the IEEE 754 standard.

Acknowledgments

The authors wish to acknowledge the contributions of the circuit design team of Leon Sigal, Robert Averill, Thomas McPherson, Sean Carey, Fanchieh Yee, Barry Winter, Rick Dennis, and Dave Webber. They would also like to acknowledge system technical guidance from Charles Webb and Timothy Slegel; the verification work of Michael Mullen, Mark Decker, and Jeff Li; and the implementation help of Kai-Ann Mueller and Ashok Shenoy.

References

1. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard No. 754-1985*, The Institute of Electrical and Electronics Engineers, Inc., New York, August 1985.
2. IBM Corporation, *Enterprise Systems Architecture/390 Principles of Operation*, Order No. SA22-7201-05, September 1998; available through IBM branch offices.
3. P. H. Abbott, D. G. Brush, C. W. Clark III, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A. Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, Sr., R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White, "Architecture and Software Support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point Arithmetic," *IBM J. Res. Develop.* **43,** No. 5/6, 723–760 (1999, this issue).
4. E. M. Schwarz, L. Sigal, and T. McPherson, "CMOS Floating-Point Unit for the S/390 Parallel Enterprise Server G4," *IBM J. Res. Develop.* **41,** No. 4/5, 475–488 (July/September 1997).
5. D. E. Hoffman, R. M. Averill, B. Curran, Y. H. Chan, A. Dansky, R. Hatch, T. McNamara, T. McPherson, G. Northrop, L. Sigal, A. Pelella, and P. M. Williams, "Deep Submicron Design Techniques for the 500MHz IBM S/390 G5 Custom Microprocessor," *Proceedings of the 1998 International Conference on Computer Design*, Austin, TX, October 1998, pp. 258–263.
6. G. Northrop, R. Averill, K. Barkley, S. Carey, Y. Chan, Y. H. Chan, M. Check, D. Hoffman, W. Huott, B. Krumm, C. Krygowski, J. Liptay, M. Mayo, T. McNamara, T. McPherson, E. Schwarz, L. Sigal, T. Slegel, C. Webb, D. Webber, and P. Williams, "600MHz G5 S/390 Microprocessor," *International Solid-State Circuits Conference Digest of Technical Papers*, February 1999, pp. 88–89.

7. M. A. Check and T. J. Slegel, "Custom S/390 G5 and G6 Microprocessors," *IBM J. Res. Develop.* **43,** No. 5/6, 671–680 (1999, this issue).
8. T. Slegel, R. Averill, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 Microprocessor," presented at *Hot Chips 10*, Stanford, CA, August 1998.
9. 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.
10. E. M. Schwarz and C. Krygowski, "The S/390 G5 Floating Point Unit Supporting Hex and Binary Architectures," *Proceedings of the Fourteenth Symposium on Computer Arithmetic*, Adelaide, Australia, April 1999, pp. 258–265.
11. E. M. Schwarz, B. Averill, and L. Sigal, "A Radix-8 CMOS S/390 Multiplier," *Proceedings of the Thirteenth Symposium on Computer Arithmetic*, Asilomar, CA, July 1997, pp. 2–9.
12. S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan, "A General Proof for Overlapped Multiple-Bit Scanning Multiplications," *IEEE Trans. Computers* **38,** No. 2, 172–183 (February 1989).
13. S. Vassiliadis, E. M. Schwarz, and B. M. Sung, "Hard-Wired Multipliers with Encoded Partial Products," *IEEE Trans. Computers* **40,** No. 11, 1181–1197 (November 1991).
14. K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley & Sons, Inc., New York, 1979.
15. S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, CBS College Publishing, New York, 1982.
16. 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.
17. R. E. Goldschmidt, "Applications of Division by Convergence," Master's thesis, M.I.T., Cambridge, MA, June 1964.
18. 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).
19. 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).
20. M. J. Flynn, "On Division by Functional Iteration," *IEEE Trans. Computers* **C-19,** No. 8, 702–706 (August 1970).
21. E. V. Krishnamurthy, "On Optimal Iterative Schemes for High-Speed Division," *IEEE Trans. Computers* **C-19,** No. 3, 227–231 (March 1970).
22. 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).
23. C. V. Ramamoorthy, J. R. Goodman, and K. H. Kim, "Some Properties of Iterative Square-Rooting Methods Using High-Speed Multiplication," *IEEE Trans. Computers* **C-21,** No. 8, 837–847 (August 1972).
24. 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.

Received November 20, 1998; accepted for publication May 24, 1999

Eric M. Schwarz *IBM Server Division, 522 South Road, Poughkeepsie, New York 12601 (eschwarz@us.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 a Senior Engineer; he was Execution Unit Logic Technical Leader for the G5 processor. Currently, he is the Central Processor Logic Technical Leader for follow-on microprocessors. His research interests are in computer arithmetic and computer architecture. He is the author of 18 filed patents, eight pending patents, and several journal articles and conference proceedings.

Christopher A. Krygowski *IBM Server Division, 522 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 in Fort Collins, Colorado, in August 1999. He is an Advisory Engineer with the IBM Server Division in Poughkeepsie, New York. Mr. Krygowski joined IBM in 1989 and has had various logic design and simulation responsibilities in development of the IBM S/390 CMOS and bipolar central processor units. He is currently leading the logic design of the execution unit of a future S/390 CMOS microprocessor. His current research interests include floating-point arithmetic, high-frequency microprocessor design, and fault-tolerant computing. He is the author of one filed patent and one pending patent; Mr. Krygowski has received two IBM Outstanding Technical Achievement Awards.