FPU Implementations with Denormalized Numbers

Eric M. Schwarz, *Member*, *IEEE*, Martin Schmookler, *Senior Member*, *IEEE*, and Son Dao Trong

Abstract—Denormalized numbers are the most difficult type of numbers to implement in floating-point units. They are so complex that certain designs have elected to handle them in software rather than in hardware. Traps to software can result in long execution times, which renders denormalized numbers useless to programmers. This does not have to happen. With a small amount of additional hardware, denormalized numbers and underflows can be handled close to the speed of normalized numbers. This paper will summarize the little known techniques for handling denormalized numbers. Most of the techniques described here only appear in filed or pending patent applications.

Index Terms—Denormalized numbers, subnormals, IEEE 754 Standard, floating-point hardware, underflow trap.

1 Introduction

The IEEE 754 binary floating-point standard [1] defines a set of normalized numbers and four sets of special numbers. Of the four types of special numbers, three do not require computation for arithmetic operations. These include Not-a-Numbers (NaN), infinities, and zeros. Denormalized numbers, also known as subnormals or denormals, are the fourth type of special number and do require computation. Below, the representation for normal and denormal numbers is given, as well as a brief comparison of the two formats.

Normalized numbers can be described by the following:

$$X = (-1)^{X_s} * 1.X_f * 2^{X_e - bias},$$

where X is the value of the normalized number, X_s is the sign bit, X_f is the fractional part of the significand, X_e is the exponent, and $bias$ is the bias of the format which corresponds to 127, 1,023, and 16,383, for single, double, and quad, respectively.

Denormalized numbers can be described by the following:

$$X = (-1)^{X_s} * 0.X_f * 2^{1-bias}, \quad X_e = 0, X_f \neq 0.$$

The denormal format differs from a normal number in that there is no implied bit and the exponent is not equal to $X_e - bias$, but, instead, is forced up by 1 to $Emin$, which is equal to -126, -1,022, and -16,382, depending on the format. Since the dataflow of a Floating-Point Unit (FPU) is generally optimized for normalized numbers, which are the most common, there must be a particular mechanism to handle numbers that require this subnormal format.

- *E.M. Schwarz is with the IBM Server Division, 2455 South Road, Poughkeepsie, NY 12601. E-mail: eschwarz@us.ibm.com.*
- *M. Schmookler is with the IBM Server Division, 11400 Burnett Road, Austin, TX 78758. E-mail: martins@us.ibm.com.*
- *S. Dao Trong is with the IBM Server Division, Schoenaicher Str. 220, Boeblingen, 71032, Germany. E-mail: daotrong@de.ibm.com.*

Manuscript received 1 Dec. 2003; revised 29 Jan. 2005; accepted 15 Feb. 2005; published online 16 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0247-1203.

There are two obvious areas where denormals must be taken into consideration: as input to an arithmetic operation and as output when an intermediate result underflows and exceptions are disabled. There have been several variations in designs regarding the aspects of subnormals that are managed in software versus hardware. Certain implementations force all denormalized input to software, while others handle easy cases in hardware. Several SPARC implementations support gross underflow in hardware, but force other underflow cases to software [2], [3]. Designs exist such that, when a denormalized operand is detected, either a trap to software or a stall in hardware is forced in order to transform the denormal number into a normalized number with a greater exponent range. The vector unit used in the Apple G4 and G5 processors from Motorola and IBM [4] traps on denormal inputs and underflow results when in Java mode, but, in another supported mode, noncompliant mode, forces both denormal inputs and results to zero. The problem with these techniques is the performance loss from stalling dispatch of instructions with denormal input.

This paper will present feasible, and relatively fast methods in hardware for handling both denormal inputs and underflow cases that require denormalization of the result with modest effects on area and performance. First, variations in the attributes of architecture that affect the handling of denormals will be given in Section 2. Techniques to manage denormal input will be discussed in Sections 3 and 4. This includes the tagging of data in register files (Section 3), as well as hardware methods for pipelineable instructions such as multiply, add, and fused multiply-add, which do not require tags (Section 4). In the following two sections, 5 and 6, schemes to deal with an intermediate result that underflows will be addressed. Section 5 details the denormalization of an intermediate result and Section 6 specifies how to prevent denormalization altogether. The final three sections prior to the conclusion provide concrete examples of several techniques described throughout the paper implemented in three different IBM processors.

Many of the methods, algorithms, and systems described here were previously presented [5] at the 16th Symposium on Computer Arithmetic. In this new paper, we provide

additional algorithms for handling some special cases, describe handling of denormals for some instructions not covered in the previous paper, and present more details on the Z990 floating-point unit and how it deals with denormal operands and results.

2 Architecture Variations

Both the instruction set architecture and the constraints set by the overall processor microarchitecture have an impact on the way denormal numbers are processed within hardware. Specific aspects include the different precisions that are supported, the internal representation of subnormals in the registers, and the operations that can be performed on each of these data types. In this section, we will provide several examples of the different formats that particular architectures utilize to represent a denormal number.

Intel IA-32 architecture [6], as well as a few other architectures [7], converts all data types to the widest supported format. When data is loaded from memory, whether it be in single or double precision, it is converted to double-extended format, which is 80-bits wide and includes 15 bits of exponent and 64 bits of significand. Single or double precision denormals are converted to normalized numbers. Similarly, the PowerPC architecture [8] supports single and double precision formats in memory, but converts all single precision data to a normalized double format in the floating-point registers (FPRs). In both IA-32 and PowerPC, load and store operations could require an alignment. Despite the similarities of the IA-32 and PowerPC architectures, they differ in their approach to handling denormal results within the FPU. The IA-32 architecture has precision control in its operations, but it only affects the fraction length and not the exponent length, which is in double extended format. The PowerPC architecture implements precision control with a "clamped" exponent range to the particular format desired. The results between these two platforms will differ for denormal numbers even though both abide by the IEEE 754 Standard.

The IA-64 [9] format is an 82 bit format which includes a 17 bit exponent field and a 64 bit significand. Like the IA-32, the significand includes an integer bit and a 63 bit fraction. However, the IA-64 permits all formats to represent denormal data in the registers unnormalized, whereas the IA-32 only permits the double extended format. This allows the IA-64 implementations to skip the normalization step after the load from memory which is required for the PowerPC and IA-32.

The IBM zSeries architecture [10], which is the new name for the 64-bit version of S/390 [11], demonstrates a very different choice for internal representation of the supported precisions than the three processors mentioned above. All data in the register file is represented in the same format as in memory. The exponent field is not separated from the fraction and there is no coercing of the loaded data type into an internal data type. There are single, double, and quadword data types for both Binary Floating-Point (BFP) and Hexadecimal Floating-Point (HFP). Therefore, being able to recognize denormal data prior to processing is complex.

3 Register Files with Tags

As we have discussed, certain architectures require that all data precisions be converted internally to the widest precision. A denormalized operand, however, must be normalized during the conversion so that it can be distinguished from normalized data that could potentially have the same exponent. As an alternative to normalization prior to computation, a nonarchitected format might be used consisting of a single tag bit and an unnormalized significand. These tags allow for quick data recognition during instruction processing and, at times, allow single precision denormals to be loaded directly into the FPRs, bypassing the normalization step. In some cases, tagging does not eliminate the need to normalize data prior to instruction processing, but merely delays the normalization until the data is needed. Furthermore, if tags are used, we can avoid conversion after rounding. Design choices such as tagging are influenced by constraints of the microarchitecture, which can include timing considerations of load and store operations, how instructions are dispatched to each unit, and whether registers are renamed.

If tagging is added, it can be exploited to facilitate instruction processing in several ways. A tag could be used not only for single precision denormals, but also for double precision denormals when determining the value of the implied bit. Also, if the processing of an instruction requires that the operands first be normalized, then, rather than stalling regardless of the operand value, tag bits can be used as a prompt indicator to stall subsequent instructions. Additional tag bits can be used to designate the other special value types, infinities, zeros, and NaNs. These type of indicators can be especially useful if the instruction requires normalized operands since denormals require prenormalization while zeros do not. We have already seen that, for the PowerPC architecture, tags can be exploited to avoid renormalizing single precision denormals after they have been rounded. Considerably more circuitry and delay are encountered if tagging always sets the implied bit correctly since the fraction must be examined for all zeros. In the IA-64 architectural model, the exponent is also forced to all zeros when the data is zero. In the Power4/5 implementation of the PowerPC, the exponent is not forced for these values, but the tags override the exponent contents when the data is used. A case study in Section 9 gives a detailed description of how tags are used in the Power4/Power5 FPUs, as well as in related implementations.

There are also costs associated with tagging denormals and/or other special number types. First, to determine the tags, extra circuitry and delay must be added. This can be quite substantial if the fraction is examined for an implementation that distinguishes zeros from denormals. Also, the registers must be extended to accommodate these tag bits, although this is typically insignificant. If tags are also used for double precision denormals, then all instructions may create tags. This could have the effect of complicating instructions such as convert-to-integer since the architecture may allow floating-point instructions to operate on the result even though it would be nonsense. When multiple formats are used for the same data, processing can become even more complex. For example, a double precision store may have the requirement of normalizing unnormalized single precision data, while single precision store instructions may have to denormalize

data that is normalized because it is in the single denormal range. Futhermore, every double precision instruction may become more difficult to execute when an operand is an unnormalized single denormal. Due to this, in some PowerPC processors, all unnormalized operands are prenormalized before the instruction is executed, while, in others, prenormalization is done only for special cases. Last, design verification becomes trickier because testcases may need to verify correct processing for each format of a particular data value.

Predetermining if data is denormalized and creating tags has limited value even for architectures such as the PowerPC. It may solve certain problems and, at the same time, create new difficulties. In the successful case, since the detection of denormalized operands is done prior to arithmetic calculation, tagging can save hardware and eliminate critical paths. Unfortunately, just the creation of these tags can add significant complexity. In the next section, we will show that not only is it possible to handle denormals without the overhead of tagging, but also how to do this at speed for the most common operations.

4 Operations with Denormalized Inputs

In this section, we describe how the pipelineable arithmetic instructions can be designed to handle denormalized input operands without the use of tags or the predetermination of the integer bit. Denormal inputs must be corrected for in two areas: in the alignment shift calculation, where an exponent of zero must be forced to Emin, and in the fraction data flow, where the integer bit must be correctly set. In the following three subsections, solutions for the operations addition, multiplication, and fused multiply-add are presented.

4.1 Denormal Input for Addition

There are certain aspects of denormal numbers that cause them to be more complex operands than normal numbers when executing a floating-point addition. The steps of floating-point addition can be summarized as follows: exponent comparison and difference calculation, alignment, conditional complementation, addition of significands, normalization, and rounding. There are two areas in which a denormal input can result in an incorrect result if not handled properly, the exponent difference and the implied bit of the operands.

Due to the representation of the exponent of denormal numbers, the exponent difference calculation will be off by one if the input operands are denormal. The exponent difference drives the aligner, so this error must be corrected prior to the addition. Since the exponent difference calculation is a timing critical path, adding extra stages to correct for denormals is not feasible. A typical implementation that solves this issue computes the exponent difference, $D = A_e - B_e$, and also $D-1$ and $D+1$, where A_e and B_e are the exponents of A and B, respectively. Then, the late detection of denormal operands is used to select between the exponent differences. This creates an extra level of delay for the shift amount. An alternative implementation adds a stage to the aligner to perform a late correction shift based on which operand happens to be denormal.

Less crucial than the exponent difference correction are the implied ones of the significands, which also must be corrected if operands are denormal. In floating-point addition, one significand must be shifted by the aligner while the other is not needed until the carry propagate addition. However, since the exponent difference calculation is the most timing critical path, the correction can easily be done before either significand is used.

We have shown that floating-point addition hardware can easily be modified to allow for denormal inputs; in the following subsection, multiple methods are presented to demonstrate that the same can be done for floating-point multiplication.

4.2 Denormal Input for Multiplication

The second pipelineable operation we will discuss is floating-point multiplication. It generally involves a Booth decode, partial product generation (Booth multiplexing), a counter tree, a carry propagate addition, normalization, and rounding. Although multiplication is typically implemented using a form of the modified Booth algorithm, which reduces the number of partial products, it can also be done as direct bit by bit multiplication. For floating-point multiplication, the product, P, is calculated for the multiplicand, X, and the multiplier, Y, as shown by the following [12]:

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

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

$$P = X * Y = loc1 + loc2 + P' \; where$$

$$P' = \sum_{i=1}^{n-1}\sum_{j=1}^{n-1} x_i y_j * 2^{-(i+j)}$$

$$loc1 = x_0 * \left(y_0 + \sum_{j=1}^{n-1} y_j * 2^{-j} \right)$$

$$loc2 = y_0 * \sum_{i=1}^{n-1} x_i * 2^{-i}.$$

x_0 and y_0 are the integer bits of X and Y. The primary problem with denormal inputs in multiplication is the determination of the integer bit. Williams [13] separated the partial products that are dependent on an implied one, denoted by $loc1$ and $loc2$ (for leading ones correction), from the other partial products, denoted by P'. For a 53 bit direct multiplication (non-Booth), there were 52 partial products representing P' and two others ($loc1$ and $loc2$) which were dependent on the implied one of the multiplier and multiplicand. Williams noted that, in a counter tree, it is common to have a few inputs that can have delayed arrival times since they have fewer counters in their path. Also, some counter designs are tapered to have varying propagation delay based on input and output. Thus, it is possible to delay the two late arriving partial products so that the exponent can be examined for all zeros and the implied bit can be set accordingly.

A similar technique is used in the next zSeries FPU, which uses a Booth radix-4 multiplier [14]. However, rather than adding a leading ones correction to the partial product array, leading zero correction terms are subtracted. Rather than assuming that x_0 is zero and adding a term if it is equal to one, the opposite is assumed. In general, if two late inputs are available in the counter tree, two terms can be

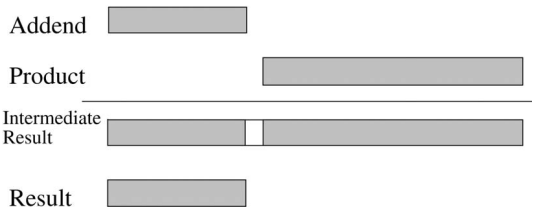


Fig. 1. Alignment for addition: addend much greater than product.

added to correct for the multiplier and multiplicand. The cost would be extra counter area since there is one additional 3:2 counter for each term. Alternatively, the most significant term of a Booth decode could be corrected prior to the partial product creation by using the multiplier's implied bit. This would necessitate a delayed partial product, but would eliminate the additional partial product. Then, only one correction term would be needed to correct for the multiplicand's implied bit. This could be done by calculating the Booth decode with $y_0 = 0$ and $y_0 = 1$ in parallel and choosing between the two after y_0 is known.

Thus, with few additional partial products that have delayed inputs, there are multiple ways to correct for denormal input into a multiplier. If the counter tree can accept more rows without adding stages, this type of design is nontiming critical. Moreover, the area added by one or two terms is small in comparison to the overall counter tree area. The final subsection will cover hardware solutions for the fused multiply-add operation.

4.3 Denormal Input for Fused Multiply/Add

Several architectures, including the PowerPC floating-point architecture, are optimized for the fused multiply-add operation [15]. This operation is a multiply-add or multiply-subtract, e.g., A*C + B or A*C - B, and the product (A*C) is not rounded before the addition or subtraction. The pipeline structure is optimized to exploit this instruction, even though it may increase the latency of individual Add, Subtract, and Multiply instructions. In this arrangement, the C operand is set to 1.0 for an add or subtract instruction and, for a multiply instruction, the B operand is set to zero.

One aspect of design that is significantly distinct for this architecture is the alignment of operands. When aligning operands for Add or Subtract, the customary practice is to first compare the exponents of the operands, then the operand with the smaller exponent is shifted to the right by the appropriate amount, and, finally, if it is an effective Subtract (i.e., A+ (-B), A - (+B)), the smaller operand is complemented. In a multiply-add dataflow, the development of the sums and carries for the product are part of the most critical path. It would require additional circuitry and longer latency if the sum and carry outputs were to be shifted and complemented for alignment with the addend. Therefore, regardless of the exponents, only the addend B is aligned and complemented. To minimize latency, the alignment of B is done at the same time as the product is being developed and then merged with the product in the final carry save adder.

When the addend B is drastically greater than the product (A*C), it is placed logically left of the product with a small gap, usually two bits, between them, as shown in Fig. 1. When the addend is much smaller than the product, the majority of bits from the addend that do not align with the product will not be significant and can be logically ORed into a sticky bit, which is added to the product. Therefore, the total width of the intermediate result that is formed is three times the precision length plus a few extra bits.

If the addend B is denormalized, the most critical issue is the correction of the shift amount, which is computed by the exponent difference calculation. As was the case in floating-point addition, the significand of B can be corrected before reaching the aligner. The exponent difference calculation, which is timing critical, becomes more complex for this operation since any of the three operands can be denormal. The calculations are shown below:

$$
\begin{aligned}
&Sum = B + A * C \; where \\
&\quad B = (-1)^{B_s} * (b_0 + B_f) * 2^{B_e + \overline{b_0} - bias} \\
&\quad P = (-1)^{P_s} * P_m * 2^{(A_e + \overline{a_0} + C_e + \overline{c_0} - bias) - bias} \\
&\quad D = B_e - A_e - C_e + bias + z, where \\
&\quad z \, \epsilon \, \{-2, -1, 0, +1\}.
\end{aligned}
$$

Note that it is not necessary to consider the case where both A and C are denormals ($z = -2$) since this product will underflow so severely that, even if B is a denormal, it will not need to be aligned with the addend.

We have seen how the shift amount calculation is affected if any of the inputs are denormal. Another aspect of a fused multiply-add instruction in which denormals can affect the result is multiplication. For this, either of the two techniques mentioned in the previous subsection can be used to correct the significands of the multiplicand and multiplier. These two problems, uncertainty in the exponent bias and in the value of the integer bit of the significands, are due to late detection of denormal operands. There are a few other minor difficulties denormals inputs cause in the calculation; the specific cases will be described in the following paragraphs.

In a fused-multiply add pipeline that supports denormal inputs, the alignment shifter must be extended two bit positions past the LSB of the product and the sticky bit can only include bits shifted right beyond these two bit positions. The following description of a very rare case explains why the two bits are needed.

Suppose that A is a minimum denorm, C is a number large enough to produce a product (A*C) that is within the range of normal numbers, and B is less than A*C in magnitude and is being subtracted from the product. For double precision, the significant bits of A*C would occupy only the low order 53 bits of the 106-bit product. If the product has several leading zeros in the fraction, then the subtraction of B could cause the most significant bit (MSB) of the result to be shifted right by one bit. Consequently, the first bit position to the right of the LSB of the product becomes part of the result and, therefore, that bit position of the alignment shifter must be retained. The second bit to the right must also be retained for the correct result when the rounding mode is round to nearest.

Certain implementation choices for a fused multiply-add dataflow could cause difficulties with underflow when the underflow exception bit is enabled. In this situation, the result must be normalized and rounded with a rebiased exponent prior to invoking the trap handler. For this scenario, there are three important cases to consider.

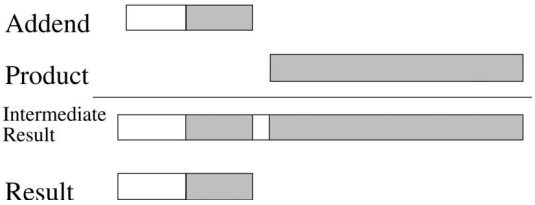


Fig. 2. Alignment: Denorm addend much greater than product, trap disabled.

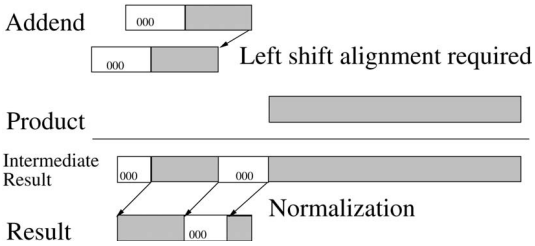


Fig. 3. Alignment: Denorm addend much greater than product, trap enabled.

The first case occurs when the addend (B) is a denormalized number that is much greater than the product (A*C) and is placed slightly to the left of A*C, rather than being correctly aligned. Fig. 2 shows the intermediate result for this case, as well as the final result when the underflow trap is not enabled. Fig. 3 shows that the gap between B and A*C is larger if they were properly aligned, which requires a larger intermediate result and shifter. When the underflow trap is enabled, it is apparent that a portion of the product will shift into the result when the result is normalized and that the alignment shown in Fig. 2 will cause an incorrect result to be produced. A design by Hitachi [16] utilizes a larger shifter and intermediate result. The aligner may shift B by up to 116 bit positions in either direction with respect to the product. The resulting symmetry simplifies the logic and circuit design. Furthermore, since the layout of the aligner is folded, it does not increase the area.

With higher frequencies causing difficulties in allowing stalls of any kind due to unusual data, we have been investigating ways of handling even this special case without any extra cycles of delay. From observation of the Hitachi design, we have come up with a method for the above case that provides the same latency, but eliminates the need for a wider output from the alignment shifter or a wider intermediate result. We see from Figs. 2 and 3 that, if the additional left shift required is less than the number of leading zeros in B, then no significant bits would shift beyond the left boundary defined by Fig. 2. Therefore, for this case, we do not need to extend the shifter, but must provide a means for shifting the addend to the left. Next, we look at the case where the additional left shift that would be required for correct alignment is greater than the number of leading zeros in the addend. It is apparent that, if we limited the left shift to the number of leading zeros, then the addend would then be normalized in the shifter and, also, that no part of the A*C product can be included in the normalized result. For this case also, neither the shifter nor the intermediate result needs to be extended. However, it appears that we would need to be able to select the number of leading zeros and use that for the alignment left shift

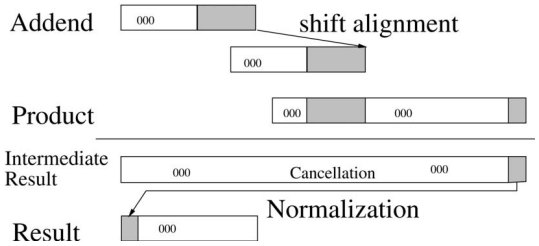


Fig. 4. Alignment: denorm addend, denorm product, trap enabled, subtraction with massive cancellation.

amount. We have found that it might be easier, instead, to force the left shift to zero, as we do in Fig. 2, for when the underflow trap is not enabled. Then, we could use the leading zero count in the normalizer instead, for the enabled underflow case. But, we must then prevent any of the A*C product from shifting into the normalized result. We avoid that by forcing the output of the adder or the right portion of the intermediate result to all zeros. For effective subtraction, the upper part of the intermediate result corresponding to the addend becomes decremented since it is complemented, incremented by the carry from the adder, and then complemented again. For this case, we force the output of the adder or the right portion of the intermediate result to all ones so that trailing ones are shifted into the normalized result. This allows the result to be properly rounded, even for the round-to-zero mode.

The three processors described in the following sections handle this same case without larger shifters and intermediate results, but all take additional cycles. The latest zSeries processor, Z990 [17], traps to internal software when this case is detected, while both PowerPC processors prenormalize the addend B.

The second case occurs when the addend is greater than the product and is aligned such that it overlaps the high order bits of the product. Most published figures of a multiply-add dataflow [18], [19] show both the adder and LZA to be the full width of the intermediate result, which is at least 160 bits for double precision. However, since the upper one-third of the intermediate result, which is 54 bits, only contains the part of the aligned addend that does not overlap the product, that portion of the adder can be replaced by an incrementer. Likewise, that upper one-third segment of LZA can be removed and replaced with the alignment distance of the B operand. When the underflow trap is not enabled and B is denormal, we can still normalize to the same position, which leaves the result denormalized. However, when the trap is enabled, the normalization must take into account the number of leading zeros in B. One way to do this with a truncated LZA is to count the leading zeros and add that amount to the alignment distance of the addend.

The final case, illustrated in Fig. 4, is a variation of the previous case described. Again, the addend is only partially shifted over the product, but this time, due to the number of leading zeros, all of the addend's significant bits overlap the product. This time the LZA must be used to determine the normalization shift amount.

We have alluded to the fact that, if underflow traps are disabled and the result is below the range of normal numbers, the result must be in a denormalized format. In the following section, we discuss what it means to

denormalize a number, the difficulties encountered with denormalization, and methods to solve it.

4.4 Denormal Input for Other Operations

Other instructions for which denormal operands are a problem are floating-point division, square root, and reciprocal estimate instructions. Since division and square root are generally multicycle, they often do not have a fixed number of cycles for execution. The main problem is that most good algorithms require that the divisor be normalized, whether for a table lookup and Newton-Raphson type algorithm or for a multibit SRT algorithm. So, if needed, extra cycles are taken to normalize the divisor and extra cycles may also be taken to denormalize the result.

The reciprocal estimate instructions are much more of a problem. Initially, these instructions were used just for graphics applications, where the exponents were in a well-behaved range. For vector applications, however, hardware divide instructions are often replaced by pipelineable software emulations, using the reciprocal instruction to get an initial approximation of the reciprocal of the divisor. The IA-64 does not include hardware divide and square root instructions, but publishes software emulations developed by Markstein [20] instead. The larger internal exponent range of that architecture allows the reciprocal instruction to be used safely with denormal operands, without overflow occurring. However, the operand must still be normalized before accessing the lookup tables.

For architectures such as PowerPC, most denormal operands cause overflow to result with the reciprocal estimate. Only a two-bit normalization is needed if overflow cases do not require use of the lookup tables. But, when the overflow trap is enabled, that would not be sufficient. Also, the reciprocal square root estimate instructions can have grossly denormalized operands without overflow occurring. One method that has been used to normalize the operand quickly without a separate shifter is to use the alignment shifter in conjunction with a leading zero counter. If the alignment shifting for a multiply-add instruction does not begin until after the exponent difference has been calculated, the leading zero count can also be obtained in about the same delay. If the complement of the 6-bit count is then used in place of the exponent difference for the alignment shift, then the leading one will always emerge at bit 63 of the shifter. Then, the bits starting from bit 64 can be sent to the lookup tables.

5 Denormalization

After normalization of an intermediate result is complete, it is determined whether the final result underflows. If the underflow trap is disabled, the intermediate result must be aligned such that the exponent is equal to $Emin$ and then rounded. This alignment and subsequent rounding operation is called denormalization. The primary problem with denormalization is that it is not detected early enough in the pipeline to utilize the normalizer. This requires either a denormalization unit or a wrap back to the top of the pipeline to utilize the normalizer. Both of these solutions have been used and will be described below.

There have been several designs that included a denormalization unit; one example is AMD [21], [22]. This technique does not require the pipeline to be stalled since the intermediate results are sent to another unit. One problem with this design is that the denormalization unit requires a large right shifter, which can be expensive. Moreover, it requires an out-of-order execution design because subsequent operations would complete earlier than the instruction requiring denormalization of the result. This entails a checkpoint ordering buffer to reorder instructions coming from both the FPU and the denormalization unit. In an out-of-order execution design, this buffering is already available.

A second solution is to make use of the existing shifters in the FPU, rather than adding a dedicated unit for denormalization. In a nonpipelined design, it is simple to feed data back to the top of the pipeline in order to employ either the aligner or normalizer [23]. But, in a pipelined design, a mechanism to squash or reorder subsequent instructions following the instruction requiring denormalization must exist. Schwarz et al. [24] show a mechanism to feed back an intermediate result to an earlier stage in the pipeline if it has been detected that no other instructions are in the pipeline. In the case that another instruction is in the pipeline, all instructions are flushed from the pipeline, then the instruction requiring denormalization is reissued for a second execution in nonpipelined mode. This method guarantees the instruction will be the sole user of the FPU pipeline and, hence, can easily wrap back to perform denormalization if necessary. Results from the first execution are not saved; if another processor in the configuration happens to write over either the instruction or the data, then it is possible that the second execution may no longer require denormalization. This technique was used on the 1998 S/390 G5 FPU [25] and, since a mechanism for conditionally issuing an instruction in a pipelined or nonpipelined manner was already in place, only a small amount of extra control logic was added.

Variations on this technique have been used in other designs. One FPU detects underflow early enough in the pipeline so that a subsequent instruction can be stalled from reaching the normalizer. The normalizer output is then fed back into the normalizer as input to effectively right shift the intermediate result. The key aspects of this method are detecting possible underflow in an early stage and forcing stalls to separate instructions. Another method allows underflow to be detected very late, but requires a small shifter at the bottom of the dataflow. In this scheme, underflow is detected in the normalizer and, at this point, the pipeline is stalled until denormalization is complete. The latch after the normalizer has a feedback path that can choose between holding the data in the latch or shifting the data up to 4 bits. The data is shifted right 1 to 4 bits each cycle until the exponent is equal to $Emin$. The stall is then released and the data is rounded properly to complete denormalization. The denormalization process for double precision operands can require up to 13 cycles, but this is drastically less than the time it would take to trap to software. Also, the only extra hardware required is a very small shifter and minimal detection logic. This detection logic, however, controls the stalls in the pipeline, which can be timing critical, and, therefore, this method is not commonly used. This type of technique was implemented in the 1998 S/390 G5 FPU to handle quad precision denormal numbers, which was necessary since the feedback paths in the dataflow were only double precision.

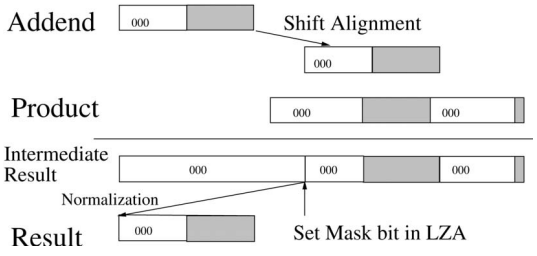


Fig. 5. Alignment: generate denorm result, using mask to limit normalization.

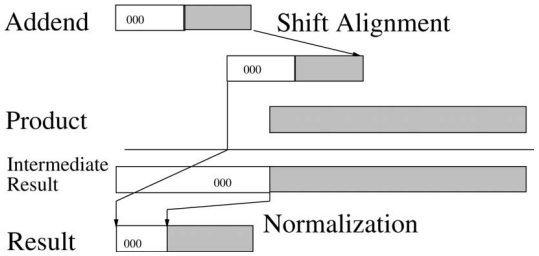


Fig. 6. Alignment: generate denorm result, exponent of addend greater than exponent of product.

Two methods for denormalization were presented in this section. Both a dedicated denormalization unit and the reuse of existing hardware add complexity to an implementation by requiring extra cycles to complete denormalization. This would not be necessary if the denormalization process could be prevented; details of this are presented in the following section.

6 Preventing Denormalization

As we have shown, denormalization of a result that has underflowed can be quite involved. Denormalization can be prevented. The trick to accomplishing this is to not permit the normalizer to shift beyond the radix point of a denormalized result. Several techniques for this have been described and we will present them throughout this section.

Urano and Taniguchi [26] show the simple technique of comparing the shift amount for a denormal result versus the shift amount computed by an LZA and selecting the smaller shift amount. Gorshtein and Khlobystov [27] also show a design of creating the two shift amounts in parallel and they go on to suggest two units for the implementation. One unit supports the normalized dataflow with limited shifts, while the other is slow and supports the maximum shift amounts. Grushin and Vlasenko [28] were another group to propose generating both shift amounts; however, they describe a method to reduce the equation of the shift amount which combined the comparison and selection of the lesser shift amount. All of these techniques create two separate shift amounts for denormals and complete normalization, but each gives a unique method for choosing the lesser of the shift amounts.

A number of high-speed designs remove the choice between two shift amounts and instead combine the maximum shift amount of a denormal back earlier in execution. Naffziger and Beraha [29] determine the bit of the LZA that corresponds to the position of the MSB of a denormal result and force this bit to a one. This is done by examining all bits in parallel, the decoded maximum shift amount is then used to force the LZA bit to a one (Fig. 5). In Power3, Bjorksten et al. [30] create a vector corresponding to the denormal maximum shift amount using a monotonic mask. A vector is generated that has a string of ones that starts at the MSB of the denormal and ends at the LSB. This denormal vector is then logically ORed with a monotonic LZA vector and the resulting vector is used to encode the shift amount. The following equations describe this method:

$$\begin{aligned} V_i &= \overline{P_{i-2}} Z_{i-1} \overline{Z_i} + \overline{P_{i-2}} G_{i-1} \overline{G_i} \\ &\quad + P_{i-2} G_{i-1} \overline{Z_i} + P_{i-2} Z_{i-1} \overline{G_i} \\ U_i &= (i \geq (E_{product} - E_{min})) \\ M_i &= V_i + U_i \\ Shift &= LZD(M), \end{aligned}$$

where V is a commonly used LZA vector that examines three bits in parallel, P is a bit propagate using an exclusive OR function, G is a bit generate, Z is a bit zero term, U is a vector of the maximum amount the denormal can be shifted, $+$ represents the logical OR function, juxtaposition represents a logical AND function, and M is the combination of the LZA vector and denormal vector. The resulting shift amount is based on a Leading Zero Detect (LZD) of M.

Handlogten [31], in the PowerPC A50, performed the logical ORing process one step further back. Handlogten ORs the denormal vector with both the carry and sum input to the LZA. Then, only the LZA output and the resulting LZD are used to create the shift amount.

Another possibility is to calculate the normalizer shift amount earlier in the pipeline by using the intermediate result exponent and the alignment shift amount. For the purpose of denormalization, the 54 high order bits of the dataflow will be zeros and can be used to generate the leading zeros of the denormal result. Fig. 6 shows how the denormal result can be generated.

7 Overview of IBM Processors

In the following sections, three different IBM implementations of a multiply-add floating-point unit are described in order to provide the reader with a complete picture of how denormal values are handled within a particular implementation. The first processor that will be presented is the Z990, which is the second implementation of the z-series architecture. Next, the Power3, which was the first 64 bit implementation of the PowerPC architecture will be described. It preceded both the Z990 and the Power4, which will be described in the last section as a case study. The Power4 and Power 5, the successors to the Power3, incorporate both the PowerPC architecture and the architecture of its i-series processors and there is little difference in their microarchitectures.

With respect to handling of infrequent but difficult cases, the Power3 design could be considered midway between the other two. Extra delays are not incurred for most cases of denormal operands, however, a small number of special cases require stalling the issue of floating-point instructions for just a few cycles. When results such as underflow or overflow occur, no extra delays are needed.

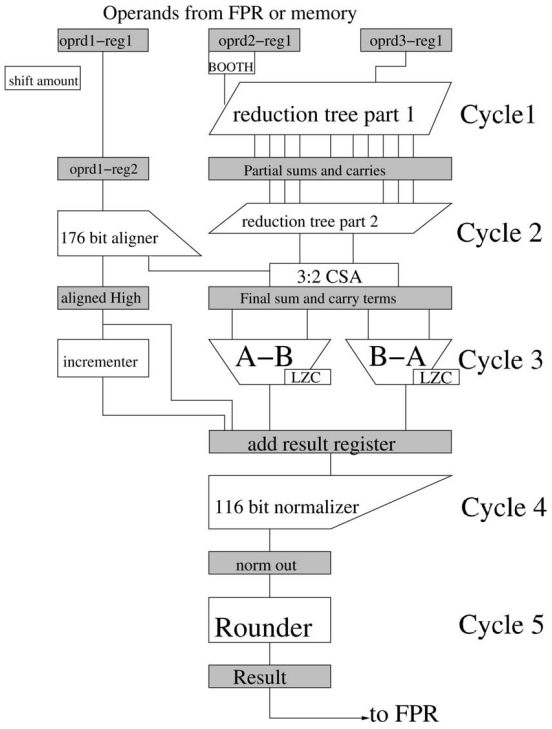


Fig. 7. Z990 dataflow.

The Z990 and Power4/5, illustrate opposite approaches to managing difficult cases. The Z990 does not introduce any delay, except for one case where an underflow trap must be taken regardless. The nontrap cases are handled in pipelined hardware. The trap cases are delayed, but the effect is negligible considering that traps require thousands of cycles. The Power4/5 avoids the tricky denormal cases by stalling the issuing of floating-point instructions for several cycles, which has insignificant effect on performance. It also stalls for massive cancellation of significant bits during subtraction, which requires large normalization shifts.

8 Z990 FP Processor

The Z990 is the implementation of the CISC architecture traditional to the IBM mainframe family. In recent years, its floating-point architecture has been extended by a considerable number of instructions to include binary floating-point instructions, as well as to make the architecture compliant with the IEEE 754 standard. Altogether, there are more than 200 instructions defined on six different formats, 93 of which are binary instructions. This is the first IBM mainframe implementation to have a fused multiply-add dataflow, which has been widely adopted by other RISC floating-point processors. Due to legacy commitments, there are quite a few constraints reflected in the design.

The dataflow (Fig. 7) is a five stage pipeline. Up to three operands can be processed, of which one can be directly from memory. This pipeline is typical for CISC processors. The fraction dataflow is 176 bits wide and can process one fused multiply-add (binary or hexadecimal) instruction directly in pipelined mode. Fraction addition is processed in two parts. In the lower 116 bits, the product is added to the aligned portion of the addend. The carry out of this addition is then used to determine if the higher 60 bits must be incremented or not. In this implementation, two 116 bits wide fraction adders are used to calculate A-B and B-A in parallel to avoid the recomplementation step that is needed when the result of a subtraction is negative. Also, a precise leading zero count for the full 116 bit results of both adders is used rather than a LZA, which may need a one bit correction. This is achieved within the cycle time and helps to simplify the detection of exponent overflow and underflow in corner cases.

The internal exponent dataflow is 16 bits wide, which is only one bit more than the architected quad binary exponent format. This is sufficient for the shift amount and exponent calculation for the quad add/subtract instructions. Two rare cases, quad multiply and divide instructions, require 17 bit exponents for special cases (overflow, underflow, denormal input, ...). For timing reasons, a choice was made to handle the special cases in the control flow. A detailed description of the Z990 FPU can be found in [17].

Note that the IBM zSeries architecture is a robust architecture that specifies many data formats. Data in all six formats (single, double, and quadword for both binary and hex) are left-aligned in the FPR and are identical to their memory formats. Denormal detection must be done while either reading data from the FPR or loading data from memory, the latter case being very timing critical. The advantage, on the other hand, is that data can be loaded/stored directly from/to memory without any reformatting.

The architecture allows instructions defined for data of one type or precision to operate on data of another type or precision and, even though the result may not be meaningful, it must be consistent and well-defined. For example, if a single precision operand is read from an FPR as a double precision operand, part of its fraction is appended to the exponent and the rest of the fraction is simply merged with the lower 32 bits to form the double precision fraction.

8.1 Denormalized Input Handling

Contrary to other implementations, the current design assumes the integer bit to be one for all binary operands and does a leading zero correction if the exponent of any operand is all zeros. Because data from FPRs or memory are left-aligned, conversion to the internal format is needed before the detection can be done. Thus, denormal decoding logic takes up the majority of the first cycle.

Before denormal detection has finished, the shift amount calculation, which is timing critical, must begin. It uses a method described earlier in which three calculations are done in parallel and both the denormal detect and case logic are used to select between the three options D, D-1, and D+1.

When the addend is a denormal, the integer bit is zeroed before the addend is passed to the aligner and it is not timing critical. Nor is the correction of the integer bit of the multiplier timing critical. This is because the multiplier array, designed to handle 56 by 56 bit multiplication, uses a radix-4 Booth encode that creates 29 partial product terms, which cannot be reduced in the first stage of the reduction tree. Therefore, the most significant product term enters the reduction tree in the second stage, allowing time for the integer bit of the multiplier to be determined. The Booth coefficient for this product term is calculated for both values of the integer bit and then the correct one is selected.

The determination of the integer bit for the multiplicand, however, is timing critical. In this implementation, the integer bit is assumed to be one, so, in the case of a denormal multiplicand, a correction term equal to the product of the integer bit position and the multiplier is subtracted from the final product. The reduction tree needs to be extended by this term, but that does not affect timing given that the reduction tree for the 29 terms, when using 3 to 2 carry-save adders as basic building blocks, has a few free inputs.

8.2 Denormalized Output Generation

The Z990 uses the technique described in Section 6 for preventing denormalization of a subnormal result. It does this by calculating the intermediate result exponent in an early stage and using this to limit normalization so that the exponent never becomes smaller than Emin. Denormal result generation only occurs when the underflow trap is disabled. There are three cases:

1. The addend exponent is smaller than the product exponent and the addend is shifted into the product range so that the intermediate result is completely contained in the adder output. The intermediate result exponent is equal to the product exponent and, when it is slightly greater than Emin, a denormal result may occur (Fig. 5). As in normal cases, the leading zero count of the adder is used to normalize the result. In these cases, a one is forced in the string that is used to encode the leading zero count at the integer bit position (of the denormal result). This causes normalization to be limited to this bit position, regardless of if there are still leading zeros. If there still are leading zeros, the result becomes a denormalized number; otherwise, the result will be normalized to its MSB and is a normalized number.
2. Same case as above, but the product exponent is smaller than Emin. Since a denormal addend has an exponent of Emin and thus would be partly to the left of the product, this case can only happen when the addend is zero. In this case, the incrementer output is included in the intermediate result, which corresponds to adding 60 zeros to the left of the adder output. The exponent is then incremented by 60; if the resultant exponent is greater than Emin, a shift amount equal to the difference is used as a shift amount for the normalizer so that it delivers the required denormalized result. If the incremented exponent is less than Emin, all significant bits are shifted out to the sticky bit position and either a zero result or the minimum denormalized number is generated. The shift amount calculation is done early in the pipeline so timing is not an issue.
3. The addend exponent is greater than the product exponent (Fig. 6) and, therefore, the addend is aligned partially, to the left of the product. A denormalized result will only occur when the addend is denormal. This case is handled in the same manner as the general case where the addend is greater than the product, except normalization is limited to the integer bit position of the addend as described in case 1.

With these mechanisms built in for denormal inputs and denormal results, processing of subnormal cases can be handled similarly to normal numbers without the use of stalls or tags. However, there are a few rare cases where a trap or slow mode execution is required.

8.3 Exceptions

For binary multiply-add instructions, there is one case that needs to be handled by a trap routine: when the addend is denormal, greater than the product, and underflow traps are enabled. In this case, the fraction result is normalized, the exponent then underflows, the leading zeros of the addend are shifted out, and significant bits of the product are shifted in to produce the full 53 bits of significance required by the architecture. The product fraction needs to be aligned correctly to the addend. The current implementation, as well as many other known ones, does not provide a way to separate the addend from the product, i.e., shift the addend to the left of the product. Therefore, a correct result cannot be produced in the dataflow.

Instead, a trap is taken in which processing is halted while the instruction is recalculated by internal software, called millicode, to produce the correct rebiased result. Then, the architected underflow trap is taken by the trap handler. The first trap is not a performance issue since the underflow trap handler must be invoked.

9 Power3 Floating-Point Processor

The Power3 stores a tag of an approximation to the integer bit in the FPRs for all data, regardless of whether that word is intended to be used as a floating-point value, a signed integer for conversion to floating-point, or for loading the FPSCR, which is the floating-point status and control register. The approximate integer bit is determined strictly from the bits in the exponent field. If the data results from a single precision arithmetic or load instruction, then only eight of the bits are used to determine its value. Otherwise, all 11 bits are examined. Whenever the approximate integer bit is set to zero, then the LSB of the exponent is forced to one so that normal and denormal operands have the same exponent bias. If the data is used for any non-floating-point operation or store instruction, then the exponent LSB is interpreted as a zero when the approximate integer bit is zero. Move type instructions, which may change just the sign bit, do not change the exponent LSB or approximate integer bit.

For most arithmetic instructions, denormal values are handled directly, regardless of whether they are normalized (for some single precision values) or unnormalized. The approximate integer bit, however, eliminates delays in the multiplier and the forced exponent LSB eliminates correction of the bias in calculating the alignment shift amount. It also enables early detection of the special cases for which an unnormalized addend cannot be handled directly. Such cases require that the issuing of instructions to the floating-point pipeline be halted, allowing the addend to be prenormalized by first cycling it through the execution pipeline alone. We refer to this action as a *stall*. There are six cases that are described here which cause a stall when one of the operands is a denormal value. Cases 3 through 6 only occur due to the mixing of single and double precision instructions and operands.

1. The architecture includes two *estimate* instructions (reciprocal and reciprocal square root) which, in some implementations, require the operand to first be normalized, regardless of whether it is a single or double precision value. In some PowerPC implementations, these instructions are not pipelineable, but, in the Power3 and Power4/5, they are.
2. In the Power3, a stall occurs whenever the addend is unnormalized, as determined from its approximate integer tag bit, and the underflow trap is enabled. This eliminates a disjoint case between a denormal addend and a smaller product which can't be aligned properly by the dataflow. Since the extra delay is minimal, there is no need to examine the other exponents to determine if underflow would actually occur.
3. When the addend is a single precision unnormalized operand used in a double precision add, subtract, or multiply-add instruction, the final result must be normalized. If the addend is greater than the product, then a problem similar to Case 2 arises in that it may not be properly aligned and unwanted bits of the product may shift into the final result. A stall to normalize the addend occurs whenever the addend is a single precision unnormalized operand and the instruction is double precision. Note that the result is within the normal range and no underflow exception occurs. Therefore, this case must be properly handled without relying on a trap to fix the result.
4. There are also three nonarithmetic instructions which require that the operand be in its architected format. Thus, a single precision unnormalized operand must first be normalized for these instructions, causing a stall. One of these instructions is a double precision store. The others should never operate on any floating-point data in a real application. One of them converts integer data to floating-point. The other moves data to the FPSCR. However, both instructions must provide a predictable and consistent result for any operand, regardless of how it was generated. Thus, the design must provide for cases which should never really occur.
5. There is also an unusual case with a single precision multiply-add where the addend is a single precision denormal value, but is in the normalized double precision architected format. If the addend is also much greater than the product, then it would be placed completely to the left of the product in the intermediate result, with no leading zeros. However, since the final result would be in the single precision denormal range, it must be denormalized to allow rounding at the proper bit position. The normalizer only shifts data to the left and, thus, the Power3 dataflow provides no way of unnormalizing it once it reaches this stage, so a stall is needed. During the stall, the addend is denormalized using the alignment shifter to shift it to the right and the exponent is set to x'381'. Then, it is processed normally in the dataflow to perform the multiply-add operation. This case is detected whenever the addend for a single precision instruction is nonzero, but its exponent is less than x'381'.
6. A case related to Case 5 above is that of a single precision store instruction with a nonzero operand whose exponent is less than x'381'. This too requires that the operand be denormalized.

It is worth noting that only Case 2 can occur with the z-series architecture, but the Z990 always traps when an enabled exception occurs. With the program execution stopped, the instruction is reexecuted in millicode to produce the correct rebiased result and then the trap is taken. The Z990 must first determine that an underflow result does occur. In Power3, since this case is handled by prenormalizing the addend, which only results in a few cycles of delay, this action may even be taken when the product is in the normal range, which would not result in an underflow trap being taken. The Z990 only impacts the underflow trap case with an extra few cycles, whereas Power3 stalls on denormal addends.

Case 1 cannot occur in z-series processors because the architecture does not include any estimate instructions or other instructions requiring table lookup. The other cases cannot occur because instructions defined for a particular precision must use operands having the same precision. All single precision data use the high order 32 bits of the architected registers, thus denormal values are always unnormalized. Instructions for converting between precisions are provided to allow operands of one precision to be used with instructions of another precision.

10 Case Study of Power4

The Power4 FPU design illustrates the use of several techniques for handling denormal operands and results. Early in the program, performance considerations forced several key decisions regarding how denormals would be handled. Each of these decisions required that certain mechanisms be provided to handle special cases. However, we then expanded on each of these mechanisms to further simplify the design or to reduce critical timing paths, without significantly affecting performance.

The first key decision was that denormal data from memory would be loaded into dedicated write ports of the FPRs without first normalizing it. This would avoid the delay and area for counting leading zeros in the fraction and then shifting it. Therefore, at least one tag bit was needed to help identify a single precision denormal value. It was determined that we would have enough time while transmitting data from cache to also determine whether the exponent field was all zeros or all ones and whether the fraction was all zeros. So, three tag bits were added, along with an integer bit corresponding to the implied bit. The tag bits allowed all special values to be quickly identified for special handling during execution of arithmetic instructions.

The second decision was that some mechanism would be needed for normalizing denormal operands for some special case arithmetic operations. The previous processor, the Power3, had attempted to eliminate stalling instruction issuing based on data values. It successfully eliminated stalls based on unusual results, such as for denormal values. However, there were several cases that required prenormalization of an operand in which it is passed through the pipeline using the LZA and normalizer and then returned to the top of the pipeline before executing the operation. During this *prenormalization stall*, the instruction queue is

prevented from issuing instructions. From this experience, it was decided that Power4 would also need a *prenormalization stall*. However, since denormalized operands are very rare, it was decided that all such operands would be prenormalized for both single and double precisions instructions. The multiply-add instructions, which may have three denormal operands, are pipelined so that each additional denormal operand takes only three more cycles. Prenormalization simplifies the execution of most instructions and the effect of the stalls on performance is negligible. The tag bits used in Power4 enable denormal operands to be detected more quickly, thereby allowing the stall signal to be sent out earlier to prevent the next instruction from entering the pipeline.

Prenormalization of a double precision operand results in an intermediate exponent which is below $Emin$ and, thus, requires another exponent bit. In Power4, two internal exponent bits are added. This allows for the product of two denormals when the underflow trap is enabled and also avoids ambiguity in the alignment shift count, which could otherwise wrap past zero or all ones. Although these extra bits are only needed in the dataflow, Power4 also adds them to the contents of the FPRs. When data is loaded from memory, these bits are determined, along with the tag bits. All arithmetic results must also produce the 13 bit exponent.

A third important decision in the design was that a short stall would also be allowed when various unusual results are produced. These include cases where very large normalization shifts might be needed due to massive cancellation and all cases of underflow or overflow. Power3 was able to avoid stalls for these cases, but with difficulty. For multiply-add operations, a 108-bit LZA and 161-bit normalizer are needed and the normalizer must also be limited when the intermediate exponent is near $Emin$. In Power4, a *back-end stall* occurs whenever a *possible* unusual result is detected two cycles before the stall must occur, thus allowing the instruction queue and the upper stages of the pipeline to halt. The stall allows the output data to then be recycled back through just the normalizer and the rounder. Thus, most stalls are only two cycles. When a denormal result is needed, the data is normalized during the first trip through the pipeline, but is not rounded. During the stall, it is then sent back to the normalizer but aligned 65 bit positions to the right. The low order bits of the intermediate exponent, which is smaller than $Emin$, provide the proper shift amount for the normalizer. The stall also allows both the LZA and normalizer to be much smaller and have less delay. Even with stalls sometimes occurring when the result is normal, the two-cycle delay has little effect on performance.

It is possible to have both types of stalls in progress within the pipeline at the same time. An instruction may begin a *prenormalization stall* in stage1. It may advance up to the fourth stage when the previous instruction reaches stage5 and begins a *back-end stall*. The *prenormalization stall* is then stopped until the *back-end stall* is completed.

Single precision denormal values may be held in the FPRs with the significand either normalized or unnormalized. If a double precision store to memory is to be executed and it is unnormalized, then a *prenormalization stall* is taken to normalize it. However, if a *single* precision store needs to be executed and it is normalized, then it needs to be denormalized. Rather than taking the data through the pipeline and denormalizing it, the alignment shifter is used. All data for stores use the Add operand input in the multiply-add dataflow. Since there are no other operands entering the multiplier, constants are forced into the exponents which correspond to those operands. If the sum of those constants is $Emin$, then the aligner will shift the significand to the right a distance equal to the difference between $Emin$ and the exponent of the normalized operand.

11 Conclusion

Implementing denormalized numbers in hardware is possible with a small amount of additional hardware. The usefulness of tagging has been discussed and its utility is partially dependent on the architecture of the processor. Denormalized input operands can also be handled for multiply and add operations by performing multiple corrections of the exponent difference calculation for alignment and by adding correction terms in the partial product array. An underflow condition with traps disabled requires a denormalized result. Denormalization can be handled in a denormalization unit or it can be prevented by stopping the normalizer from shifting past the radix point of a denormalized number. This is accomplished by modifying the LZA to prevent an indication of more than this radix point.

The Z990, Power3, and Power4 FPUs are described, which utilize some of the above techniques. These FPUs execute denormalized operands with only a few additional cycles over the execution of normalized operands.

References

[1] "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985," The Inst. of Electrical and Electronic Engineers, Inc., New York, Aug. 1985.

[2] R. Yu and G. Zyner, "167 MHz Radix-4 Floating Point Multiplier," *Proc. 12th Symp. Computer Arithmetic,* pp. 149-154, July 1995.

[3] A. Naini, A. Dhablania, W. James, and D.D. Sarma, "1-GHz HAL SPARC65 Dual Floating Point Unit with RAS Features," *Proc. 15th Symp. Computer Arithmetic,* pp. 173-183, June 2001.

[4] M. Schmookler, M. Putrino, A. Mather, J. Tyler, H. Nguyen, C. Roth, M. Pham, J. Lent, and M. Sharma, "A Low-Power, High-Speed Implementation of a PowerPC Microprocessor Vector Extension," *Proc. 14th Symp. Computer Arithmetic,* pp. 12-19, Apr. 1999.

[5] E.M. Schwarz, M.S. Schmookler, and S.D. Trong, "Hardware Implementations of Denormalized Numbers," *Proc. 16th Symp. Computer Arithmetic,* pp. 70-78, June 2003.

[6] I. Corp., "Intel Itanium Architecture Sofware Developer's Manual Volume 1 Application Architecture," ftp://download.intel.com/design/Itanium/Downloads/24531703s.pdf, Dec. 2001.

[7] M.D.V. Dyke-Lewis and W. Meeker, "Method and Apparatus for Performing Fast Floating Point Operations," US Patent No. 5,966,085, p. 7, 12 Oct. 1999

[8] *The PowerPC Architecture: A Specification for a New Family of RISC Processors,* C. May, E. Silha, R. Simpson, and H. Warren, eds. San Francisco: Morgan Kaufman, 2002.

[9] I. Corp., "IA-32 Intel Architecture Sofware Developer's Manual Volume 1 Basic Architecture," ftp://download.intel.com/design/Pentium4/manuals/24547008.pdf, 1997

[10] "z/Architecture Principles of Operation," Order No. SA22-7832-1, available through IBM branch offices, Oct. 2001.

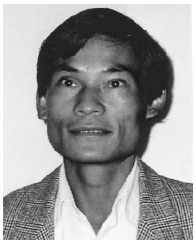
[11] "Enterprise Systems Architecture/390 Principles of Operation," Order No. SA22-7201-5, available through IBM branch offices, Sept. 1998.

[12] S. Vassiliadis, E.M. Schwarz, and D.J. Hanrahan, "A General Proof for Overlapped Multiple-Bit Scanning Multiplications," *IEEE Trans. Computers,* vol. 38, no. 2, pp. 172-183, Feb. 1989.

[13] T. Williams, "Method and Apparatus for Multiplying Denomalised Binary Floating Point Numbers without Additional Delay," US Patent No. 5,347,481, p. 16, 13 Sept. 1994.
[14] C.A. Krygowski and E.M. Schwarz, "Floating Point Multiplier for De-Normalized Inputs," US Patent Application No. 2002/0124037A1, p. 8, 5 Sept. 2002.
[15] R.K. Montoye, E. Hokenek, and S.L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Research and Development*, vol. 34, no. 1, pp. 59-70, Jan. 1990.
[16] H. Sawamoto, private presentation, no longer confidential, Jan. 2000.
[17] G. Gerwig, H. Wetter, E.M. Schwarz, and J. Haess, "High Performance Floating-Point Unit with 116 Bit Wide Divider," *Proc. 16th Symp. Computer Arithmetic*, pp. 87-94, June 2003.
[18] E. Hokenek, R. Montoye, and P.W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE J. Solid-State Circuits*, vol. 25, no. 5, pp. 1207-1213, Oct. 1990.
[19] T. Lang and J.D. Bruguera, "Floating-Point Fused Multiply-Add with Reduced Latency," *Proc. Int'l Conf. Computer Design (ICCD '02)*, Oct. 2002.
[20] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*. Upper Saddle River, N.J.: Prentice Hall, 2000.
[21] S. Gupta, R. Periman, T. Lynch, and B. McMinn, "Normalizing Pipelined Floating Point Processing Unit," US Patent No. 5,267,186, p. 10, 30 Nov. 1993.
[22] S. Gupta, R. Periman, T. Lynch, and B. McMinn, "Normalizing Pipelined Floating Point Processing Unit," US Patent No. 5,058,048, p. 11, 15 Oct. 1991.
[23] M.P. Taborn, S.M. Burchfiel, and D.T. Matheny, "Denormalization System and Method of Operation," US Patent No. 5,646,875, p. 8, 8 July 1997.
[24] E. Schwarz, B. Giamei, C. Krygowski, M. Check, and J. Liptay, "Method and System for Executing Denormalized Numbers," US Patent No. 5,903,479, p. 6, 11 May 1999.
[25] E.M. Schwarz and C.A. Krygowski, "The S/390 G5 Floating-Point Unit," *IBM J. Research and Development*, vol. 43, nos. 5/6, pp. 707-722, Sept./Nov. 1999.
[26] M. Urano and T. Taniguchi, "Method and Apparatus for Normalization of a Floating Point Binary Number," US Patent No. 5,513,362, p. 15, 30 Apr. 1996.
[27] V.Y. Gorshtein and V.T. Khlobystov, "Multiplication Apparatus and Methods which Generate a Shift Amount by which the Product of the Significands Is Shifted for Normalization or Denormalization," US Patent No. 5,963,461, p. 22, 5 Oct. 1999.
[28] A.I. Grushin and E.S. Vlasenko, "Computer Methods and Apparatus for Eliminating Leading Non-Significant Digits in Floating Point Computations," US Patent No. 5,732,007, p. 34, 24 May 1998.
[29] S.D. Naffziger and R.G. Beraha, "Method and Apparatus for Bounding Alignment Shifts to Enable At Speed Denormalized Result Generation in an FMAC," US Patent No. 5,757,687, p. 13, 26 May 1998.
[30] A.A. Bjorksten, J.D.G. Mikan, and M.S. Schmookler, "Fast Floating Point Results Alignment Apparatus," US Patent No. 5,764,549, p. 9, 9 June 1998.
[31] G.H. Handlogten, "Method and Apparatus to Perform Pipelined Denormalization of Floating-Point Results," US Patent No. 5,943,249, p. 10, 24 Aug. 1999.

Eric M. Schwarz (M'84) received the BS EngSc degree from the Pennsylvania State University in 1983, the MS EE degree from Ohio University in 1984, and the PhD EE degree from Stanford University in 1993. Since 1984, he has been employed by IBM Corporation, first in Endicott and now in Poughkeepsie, New York. Dr. Schwarz is the IBM Server Group leader of Floating-Point Units which includes responsibilities for PowerPC and zSeries mainframe development. He is currently working on the eClipz project and is the leader of FPUs for binary, hexadecimal, and decimal floating-point formats. He is the author of 28 issued patents and 20 pending applications and several journal and conference proceedings papers. He is also program cochair of ARITH17. His current research interests are in computer arithmetic, especially decimal floating-point, and also computer architecture. He is a member of the IEEE.

Martin Schmookler received the BSEE degree from Pennsylvania State University in 1956, the MSEE degree from Syracuse University in 1964, and the PhD degree in electrical engineering from Princeton University in 1968. He has worked in the IBM development laboratories in Poughkeepsie and Austin since 1956, participating in the circuit and logic design of several mainframe processors, workstations, and servers. He has been the lead logic designer for the floating-point units of the Power4 and Power5 processors. He has also been an adjunct professor at the University of Texas at Austin, teaching courses on computer design and architecture. He has coauthored several journal and conference papers, mostly in computer arithmetic. His current interest is in the analysis and design of algorithms for divide and square root operations. He is a senior member of the IEEE.



Son Dao Trong received the MS degree and PhD degree in electronic engineering from the Technische Universität Karlsruhe, Germany. He has been with IBM since 1985, working on different areas of computer design. His main interest now concentrates on the logic design of floating-point processor, mathematical analysis of used algorithms and techniques, and hardware tuning for faster cycle time.