# Hardware Implementations of Denormalized Numbers


Eric M. Schwarz
IBM Server Division
2455 South Rd., MS:P310
Poughkeepsie, NY 12601 USA
eschwarz@us.ibm.com

Martin Schmookler
IBM Server Division
11400 Burnett Road
Austin, TX 78758 USA
martins@us.ibm.com

Son Dao Trong
IBM Server Division
Boeblingen, Germany
daotrong@de.ibm.com



## Abstract

*Denormalized numbers are the most difficult type of numbers to implement in floating-point units. They are so complex that some designs have elected to handle them in software rather than hardware. This has resulted in execution times in the tens of thousands of cycles, which has made 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 discussed have only been discussed in filed or pending patent applications.*


## 1. Introduction

The IEEE 754 binary floating-point standard [1] defines a set of normalized numbers and four sets of special numbers. The special numbers are Not-a-Numbers(NaNs), infinities, zeros, and denormalized numbers which are sometimes referred to as subnormals or denormals. Operations on the first three special numbers require no computation. The only type of special number which requires computation for an arithmetic operation is denormals. Normalized numbers are represented 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 (127, 1023, and 16383, for single, double and quad). Denormals are represented by the following:

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

There is no implied bit, and the exponent is not equal to $X_e - bias$, but instead has to be forced up by 1, to *Emin* which is equal to -126, -1022, and -16382 depending on the format. Typically the dataflow of a Floating-Point Unit (FPU) is optimized for normalized numbers since they are the most common but there must be some mechanism to handle denormals.

There have been many variations as to how much of the denormalization handling should be done in hardware versus software. Some 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]. Some implementations take either a trap to software or stall in hardware when a denormalized operand is detected and transform the denormal to a normalized number with a greater exponent range. The Motorola G4 vector unit traps on denormal inputs and underflow results in Java mode, while it also has a non-compliant mode where it forces denormal inputs and results to zero. The problem with this technique is that it stalls dispatching instructions and does not solve all the execution issues of denormals and underflow. This paper will present techniques for handling denormal input as well as underflow cases which require "denormalization" in hardware.

There are two obvious areas where denormals must be handled, 1) as input to an arithmetic operation, and 2) when an intermediate result underflows and traps are disabled, a denormal number needs to be produced. First, architecture variations will be discussed which affect the handling of denormals. Handling denormal input will be discussed in the following four sections. Section 3 will discuss using tagging of register files to indicate denormal input and its effect on loads and store instructions. Section 4, 5, and 6 discuss denormal input for the operations of addition, multiplication, and fused multiply-add. The following two sections address the handling of an intermediate result

which underflows. Section 7 discusses denormalizing an intermediate result, and Section 8 addresses how to prevent denormalization from being needed. Section 9 presents a case study, the Power4 FPU and shows how it combined several of these techniques to implement denormals in hardware with very little area and keeping most of the execution in hardware.

## 2. Architecture Variations

The handling of denormal numbers is dependent on both the instruction set architecture and constraints set by the overall processor microarchitecture. The important attributes of the architecture include the different precisions which are supported, how they are represented internally in the registers, and the operations which can be performed on each of these data types.

For some architectures, all data types are converted internally to the widest supported format. For example, the Intel IA-32 architecture [4] as well as some other architectures [5] support single, double and double-extended formats in memory, but all data is automatically converted to the double-extended format when loaded from memory. This 80-bit format includes 15 bits of exponent and 64 bits of significand. Single and double denormals are normalized during this conversion. Similarly, the PowerPC architecture [6] supports single and double in memory, but converts all single format data to normalized double format in the floating-point registers (FPRs). When storing data to memory, both of these architectures must also denormalize some data values that have a lower precision than the internal format. An actual implementation can vary from this architectural model. Some PowerPC implementations, for example, provide extra bits called tags with each register which allow denormals and sometimes other special values to be quickly recognized during instruction processing. Use of these tags would allow a denormal single to be loaded into the FPRs unnormalized. The constraints of the microarchitecture, such as timing considerations of load and store operations, how instructions are dispatched to each unit, whether registers are renamed, and so on, may affect such design choices.

Despite the similarities described above for handling denormal data from memory, the IA-32 and PowerPC differ in how they handle results which correspond to denormal values. The IA-32 architecture supports both single and double precision instructions when the results are normalized. However, the precision control only affects the fraction length. The exponent range for all operations is the same as for double extended operations. Thus, results may differ from an architecture which fully implements both single and double precision as specified in the IEEE 754 standard [1], although this difference would usually result in greater accuracy. Therefore, the IA-32 arithmetic operations can only produce a denormal result in its double extended range. Single and double denormals are only produced when storing such values to memory. The PowerPC, on the other hand, when executing single precision instructions, must round all denormal results at the proper boundary. This requires that it first produce an unnormalized result with its exponent clamped at $Emin$, and then convert it to the normalized double format. If tags are used which permit unnormalized single precision representations in the FPRs, then conversion after rounding can be avoided.

The Intel IA-64 [7] illustrates yet another variation in which all precisions are represented only in the widest format, which is 82 bits. It 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-32 disallows unnormalized representations other than true double extended denormals. The IA-64, on the other hand, represents the integer bit, or implied bit, so that single and double denormals which are loaded from memory or which result from single or double operations are represented in the registers with unnormalized significands and with the exponent set to $Emin$ of the corresponding precision. Thus, a scheme for avoiding normalization corresponds to non-architected representations in the PowerPC and in some implementations of IA-32, but corresponds to architected representations in the IA-64. Thus, the IA-64 provides full IEEE compliance for single and double precision. A separate range control bit is provided which either sets the exponent range to 17 bits, or to the range which corresponds to the specified precision.

The IBM zSeries architecture[8] (which is the new name for the 64-bit version of S/390[9]) illustrates a very different choice for internal representation of the supported precisions. Both single and double precision data are represented in the FPRs with the same format as in memory. Thus, single precision data only occupy the left half of the 64-bit registers, and all denormals must be represented in the denormalized format within these registers. However, since the execution dataflow is optimized for double precision, the single format data is converted to an internal format which supports six different data types. There are single, double, and quadword data types for both Binary Floating-Point (BFP) and Hexadecimal Floating-Point (HFP). The architecture allows instructions defined for any of these data types to operate on the contents of

any of the FPRs, whether it makes sense or not. Therefore, creating tags for special operand values, as in the PowerPC, is not too useful for this architecture.

Predetermining whether data is denormalized and creating tags has limited value even for architectures such as the PowerPC. It may solve some problems while creating new ones. In some cases, tagging can save hardware and eliminate critical paths since the detection of denormalized operands is done prior to arithmetic calculation. But then creating the tags themselves may present difficulties. On the other hand, even without tagging it is possible to have implementations which can handle denormalized operands at speed. In the next section, more details concerning tags are discussed. Then the following sections describe handling of denormal operands for various operations without use of tags. The case study in section 9 then illustrates how tags are used in the Power4 FPU and in related implementations.

## 3. Register Files with Tags

In the previous section, for architectures such as the PowerPC, we showed that tagging the data that is loaded into the FPRs can simplify the conversion of single denormals to double format. For normal data, this conversion only requires adding three exponent bits corresponding to the complement of the exponent high order bit, and padding zeros to the low order bits of the fraction. But for denormalized operands, the data either needs to be normalized or else must be represented in a non-architected format. This format might consist of just a tag bit and an unnormalized significand which must be dealt with in later operations. The tag bit is needed to distinguish this value from a normalized double precision number having the same exponent.

If normalization is to be done at the time the data is loaded, then either special hardware must be added for counting the number of leading zeros and then shifting the fraction, or else one must pass the data through the floating-point adder dataflow, making use of existing hardware for doing these tasks. The FPRs may have separate write ports just for data loaded from memory, so using the adder dataflow would be difficult. In many cases, tagging does not eliminate having to normalize the data before instruction processing, but merely delays it until the data is needed and the adder dataflow can more easily be used.

Once tagging is added, it can be exploited to help simplify instruction processing in other ways. A tag could be used not just for single precision denormals but for double precision denormals as well in determining the value of the implied bit. Also, if processing of an instruction requires that the operands be first normalized, then subsequent instructions must be prevented from being issued to the unit. It is important to recognize these cases quickly, and tag bits can help in their detection. Additional tag bits can also be used to quickly detect other special values, such as infinities and NaNs. If instruction issuing needs to be stalled for prenormalizing operands, then it is also desirable to distinguish denormals from zeros, which do not require prenormalization. Another use for tagging, which was mentioned in the previous section, is to avoid renormalizing single precision denormals after they are rounded. Also, if they are later stored to memory in single format, denormalizing them could also be avoided.

There are some costs and disadvantages to using tags also. First, there is the extra circuitry and delay to determine the tags. This may be much more significant if the fraction must also be examined to distinguish zeros from denormals. If tags are also used for double precision denormals, then all instructions must also produce correct tags. This could complicate instructions such as convert-to-integer, since the architecture may allow floating-point instructions to operate on the result, although it would be nonsense. There is also the cost of providing the tag bits to all of the registers, but that is not significant.

Another disadvantage is that multiple formats may be used for the same data values. Double precision store instructions may have to normalize unnormalized single precision data, while single precision store instructions may have to denormalize data which is normalized but in the single denormal range. All double precision instructions may become more difficult to execute when an operand may be an unnormalized single denormal. In some PowerPC processors, all unnormalized operands are prenormalized before the instruction is executed, while in some other PowerPC processors, prenormalization is done only in special cases.

Also, design verification becomes more difficult, because testcases may need to verify correct processing for each format of a particular data value.

As previously noted, the use of tag bits may take several forms. One method is to merely set a tag bit for data from a single precision load, or possibly for the result from any single precision instruction. When the data is later read from the FPR, the tag and eight exponent bits can be used to determine if the data is a single precision denormal in a non-architected form. This form of tagging requires no extra delay when loading data from memory. If more time is available during the load operation, the exponent can be examined for all zeros, and the tag bit could then correspond to the

implied bit itself. This method is used in several implementations. The implied bit is determined during all load, arithmetic and conversion operations, and stored explicitly with the data in the FPR. The exponent LSB is also set to one, so that the exponent corresponds to the proper *Emin*. This simplifies the execution of subsequent arithmetic operations for both single and double precision. Considerably more circuitry and delay is encountered if the fraction is also examined to set tags which distinguish zeros and infinities from denormals and NaNs. In the IA-64 architectural model, the exponent is also forced to all zeros when the data is zero. In the Power4 implementation of the PowerPC, the exponent is not forced for these values, but the tags override the exponent contents when the data is used.

## 4. Denormal Input for Addition

Floating-point addition of denormal operands is more complex than for normalized operands. Floating-point addition involves an exponent comparison and difference calculation, alignment, conditional complementation, addition of significands, normalization, and rounding. If there are denormal input operands the exponent difference calculation will be off by one and this needs to be corrected. The exponent difference drives the aligner and is timing critical. Typically implementations compute the exponent difference, $D$, and $D-1$, and $D+1$.

$$\begin{aligned} Sum &= A + B \\ A &= (-1)^{A_s} * (a_0 + A_f) * 2^{A_e + \overline{a_0} - bias} \\ D &= A_e + \overline{a_0} - (B_e + \overline{b_0}) \\ &= A_e - B_e + \overline{a_0} - \overline{b_0} \\ &= A_e - B_e + z \quad , z \epsilon \{-1, 0, +1\} \end{aligned}$$

A late detection of operands equal to denormals then selects between the exponent differences. An alternative implementation is to add a stage to the aligner to perform a late correction shift based on which operand is denormal.

Also implied ones of the significands must be correct too. Since the critical path is not the significand input, this is not a problem. The most timing critical path is the exponent difference calculation and the implied bit should be correct by the end of this calculation. One significand is shifted by the aligner while the other is not needed until the carry propagate addition. Therefore the significand can easily be corrected for an implied bit.

## 5. Denormal Input for Multiplication

Floating-point multiplication typically involves a Booth decode, partial product generation (Booth multiplexing), a counter tree, a carry propagate addition, normalization and rounding. Multiplication can be performed with a Booth decode which reduces the number of partial products or 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 [10]:

$$\begin{aligned} X &= x_0 + \sum_{i=1}^{n-1} x_i * 2^{-i} \\ Y &= y_0 + \sum_{j=1}^{n-1} y_j * 2^{-j} \\ P &= x_0 * y_0 + x_0 * \sum_{j=1}^{n-1} y_j * 2^{-j} + y_0 * \sum_{i=1}^{n-1} x_i * 2^{-i} \\ &\quad + \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} x_i y_j * 2^{-(i+j)} \end{aligned}$$

$$\begin{aligned} P &= loc1 + loc2 + P' \quad , \qquad where \\ P' &= \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} x_i y_j * 2^{-(i+j)} \\ loc1 &= x_0 * (y_0 + \sum_{j=1}^{n-1} y_j * 2^{-j}) \\ loc2 &= y_0 * \sum_{i=1}^{n-1} x_i * 2^{-i} \end{aligned}$$

Williams [11] separated the partial products that are dependent on an implied one as denoted by *loc*1 and *loc*2 (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 just two others (*loc*1 and *loc*2) dependent on the implied ones 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 less counters in their path. Also, some counter designs are tapered having varying propagation delay based on input and output. Thus, it is possible to delay the two late arriving partial products while the exponent is examined for all zeros to see if the implied bit should be a one or a zero.

A similar technique is used in the next zSeries FPU which uses a Booth radix-4 multiplier [12]. Rather than adding in a leading ones correction, leading zero correction terms are subtracted from the partial product

array.

$$
\begin{aligned}
X &= x_0 + \sum_{i=1}^{n-1} x_i * 2^{-i} \\
Y &= y_0 + \sum_{j=1}^{n-1} y_j * 2^{-j} \\
Y &= \sum_{j=1}^{\lfloor \frac{n-1}{2} \rfloor + 1} W_j * 4^{-j} \\
W_j &\epsilon \{-2, -1, 0, +1, +2\} \\
P &= \sum_{j=1}^{\lfloor \frac{n-1}{2} \rfloor + 1} W_j * X * 4^{-j} \\
X' &= 1 + \sum_{i=1}^{n-1} x_i * 2^{-i} \\
X &= X' - \overline{x_0} \\
P &= \sum_{j=1}^{\lfloor \frac{n-1}{2} \rfloor + 1} W_j * X' * 4^{-j} - Y * \overline{x_0} \\
lzc1 &= -Y * \overline{x_0}
\end{aligned}
$$

Two terms could be used to correct for the multiplier and multiplicand if two late inputs are available in the counter tree and the extra counter area is not a concern. Note for each added term there will be one additional 3:2 counter. A second solution is to correct the Booth decode ($W_1$ term) prior to creating the partial product dependent on the multiplier's implied bit. This would necessitate a delayed partial product but would not add an additional partial product. Only one correction term would be needed to correct for the multiplicand's implied bit. Actually $W_1(y_0 = 0)$ and $W_1(y_0 = 1)$ are calculated in parallel and multiplexed after $y_0$ is known.

Thus, there are multiple ways to correct for denormal input into a multiplier. Additional partial products are needed with delayed inputs. In implementations where the counter tree can accept more rows without adding stages, this type of design is non-timing critical. It adds area of 1 or 2 - 3:2 counters but this is small in comparison to the overall counter tree area.

## 6. Denormal Input for Fused Multiply/Add

Several architectures including the PowerPC floating-point architecture are optimized for the fused multiply-add operation. This operation is a multiply-add or multiply-subtract instruction, e.g. A*C + B or A*C - B, where the AC product is not rounded before the addition or subtraction. In all implementations, the pipeline structure is optimized to exploit this operation, even though it may increase the latency of Add, Subtract and Multiply instructions. For Add and Subtract, the C operand is set to 1.0, and for Multiply, the B operand is set to zero. Alignment of operands is one function whose design is significantly different for this architecture.

The usual method of aligning operands for Add and Subtract is to compare the operand exponents and then shift the operand with the smaller exponent to the right. In a multiply-add operation, the B operand is the only operand aligned and complemented. The alignment is to any position up to a little over 53 bits greater than the AC product or to the least significant bit of the product. To minimize latency, the alignment of B is done at the same time as the product is developed, then merged with the product in the final carry save adder.

If the addend, B, is denormalized the main problem is correcting the shift amount. The significand can easily be corrected before reaching the aligner. The exponent correction is timing critical and will probably require multiple adders to compute the exponent difference based on any of three operands being denormalized.

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

Note that there is no need to consider both A and C denormalized ($z = -2$) since this will severely underflow and thus will not have to be aligned with B even if it is a denormal too.

If the multiplier or multiplicand are denormalized, the exponent shift amount calculation is affected and the significand calculation needs to be adjusted. Either of the two techniques mentioned in the previous section can be used to correct the significand.

A multiply-add dataflow can also have difficulty with underflow if underflow traps are enabled. The result written is a normalized significand rounded and with a rebiased exponent prior to invoking the exception handler. The problem is that it is difficult to produce 53 bits of significance for the case of a denormalized addend and when the product is less than the least significant bit of the addend. The dataflow does not separate the addend from the product properly and instead incorrectly concatenates the addend with a cou-

ple guard bits to the product. To handle this case some implementations like Power3 and Power4 prenormalize input denormals while others trap to software when they detect the "disjoint case" such as the latest zSeries processor [13].

## 7. Denormalization

Once an intermediate result completes normalization it can be determined whether the operand underflows. If the underflow trap is disabled, then the intermediate result needs to be aligned to an exponent equal to *Emin* and then rounded. This alignment and subsequent rounding operation is called denormalization. The problem with denormalization is that by the time underflow is detected it is too late in the pipeline to utilize the normalizer. The data either needs a denormalization unit, or somehow needs to wrap back to the top of the pipeline avoiding other instructions, or somehow needs to avoid denormalization altogether as discussed in the next section.

There have been several designs that assumed denormalization units such as by AMD [14, 15]. Basically, rather than stalling the FPU pipeline, an intermediate result requiring denormalization would be sent to another unit. The complication with this type of an implementation is that it requires an out-of-order execution design since subsequent operations would pass the instruction requiring denormalization. There would have to be a checkpoint ordering buffer to re-order instructions coming from both the FPU and the denormalization unit. This buffering is already available in an out-of-order execution design. However, the denormalization unit requires a large right shifter.

An alternative to adding a dedicated unit for denormalization is to utilize the existing shifters in the FPU. In a non-pipelined design it is simple to feed data back to the top of the pipeline to utilize the aligner or normalizer for denormalization [16]. But in a pipelined design there needs to be a mechanism to squash or reorder subsequent instructions following the instruction requiring denormalization. Schwarz [17] shows a mechanism for feeding back an intermediate result to an early stage in the pipeline if it is detected that there are no other instructions in the pipeline. In the case there is another instruction in the pipeline, all instructions are flushed from the pipeline and the one requiring denormalization is re-issued in non-pipelined mode. If the second execution requires denormalization, the instruction is guaranteed to have the FPU pipeline to itself and be able to perform denormalization. No results are saved from the first execution and if another processor in the configuration happens to write over the instruction or data, then the second execution may no longer require denormalization. This technique was used on the 1998 S/390 G5 FPU [18] and only added a little control logic since there was already a mechanism for conditionally issuing an instruction in a pipelined or non-pipelined manner.

Other similar techniques to this proposal have been used. Another FPU detected underflow early enough in the pipeline to stall a subsequent instruction from reaching the normalizer. And then it fed back the normalizer output back to its input to effectively right shift the intermediate result. The key to this technique is detecting possible underflow in an early stage and forcing stalls to separate instructions.

Another variation that the authors have seen of this technique is to detect underflow very late and provide a small shifter at the bottom of the dataflow. Underflow is detected in the normalizer. The pipeline is stalled at this point until denormalization is complete. The latch feeding the rounder also has a feedback path with a small multiplexor which enables a hold of the latch or up to a 4 bit shift. The data is right shifted 4 bits each cycle until it reaches the point where the exponent is equal to *Emin*. Then the stall is released and the data rounded properly to complete denormalization. This technique only requires a very small shifter and not much detection logic. It does create stalls in the pipeline which can be timing critical and therefore is not implemented frequently. The denormalization process for double precision can require up to 13 cycles but this is much less than trapping to software. The main drawback is the late detect of a stall can cause timing critical control signals. This type of technique was implemented in the 1998 S/390 G5 FPU to handle quad precision denormal numbers since the feedback paths in the dataflow were only double precision.

## 8. Preventing Denormalization

Denormalization can be prevented. The trick is to prevent normalizing past the radix point of a denormalized result. Urano [19] shows the simple technique of comparing the shift amounts for a denormal result versus the shift amount from a leading zero anticipator, and selecting the least shift amount. Goshtein and Khlobytev [20] 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 dataflow is slow and supports the maximum shift amounts. Grushin and Vlasenko [21] also suggest creating both shift amounts but they go into a reduction of the equation of the shift amount. They reduce the

comparison and selection of the lesser shift amount into one equation. All of these techniques create a separate shift amount for denormals and for complete normalization, and have different techniques for choosing the lesser of the shift amounts.

Some high speed designs get rid of the choice between two shift amounts and combine the maximum shift amount of a denormal back earlier in execution. Naffziger and Beraha [22] determine the bit of the LZA which corresponds to the position of most significant bit of a denormal and force this bit to a one. All bits are examined in parallel and basically a decode of the maximum shift amount is done and used to force the LZA bit to a one. Bjorksten, Mikan, and Schmookler [23] in Power3 create a vector corresponding to the denormal maximum shift amount, using a monotonic mask. It has ones in every bit starting with the most significant bit of a denormal. This denormal vector is logically ORed with a monotonic LZA vector, and the resulting vector is used to encode the shift amount. The following shows a similar method without using a monotonic mask:

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

where $V$ is a commonly used LZA vector which 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 a denormal can be shifted, and + represents the logical OR function, and juxtaposition represents a logical AND function. $M$ is the combination of the LZA vector and denormal vector, and the resulting shift amount should be based on a Leading Zero Detect of $M$. The shift amount calculation can be made a little simplier with a monotonic mask.

Handlogten [24] in the PowerPC A50 moved the logical ORing process one step further back. Handlogten ORs the denormal vector with both the carry and sum input to the LZA. And then just the LZA output and the resulting LZD is used to create the shift amount.

## 9. 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 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 rare cases involving a denormal addend with the fused multiply-add instructions which were too difficult to handle without first normalizing it. The Power3 prenormalizes the addend just for those cases, passing it through the pipeline, utilizing the leading zero anticipator (LZA) and normalizer and then returning it 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 denormal operands are very rare, it was also decided that all operands would be prenormalized for both single and double precisions instructions. For the multiply-add instructions, which may have three denormal operands, they 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 adds them also 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, denormal results occur, or rebiasing of the exponent is needed for trapped underflow or overflow. Power3 was able to avoid stalls for these cases but with difficulty. In a multiply-add operation, a 108-bit LZA and normalizer is needed, and the normalizer must also be limited when the intermediate exponent is near *Emin*. Detection of a *possible* unusual result would cause a *back-end stall* two cycles before the actual stall, thus allowing the instruction queue to halt and the upper stages of the pipeline to also halt. The stall would allow the output data to recycle back through the last two stages, which are the normalizer and the rounder. Thus, most stalls are only two cycles. If a denormal result is needed, the data is normalized 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 the LZA and normalizer to be much smaller. Even with stalls occurring at times when the result is normal, the two-cycle delay does not have a very significant 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.

There is one other interesting mechanism that is used for denormals which has not yet been described. 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 of *Emin* and the exponent of the normalized operand.

## 10. 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 be handled for multiply and add operations by performing multiple corrections of the exponent difference calculation for alignment and by correcting the multiplication result by adding correction terms. 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 denormalized number. This is accomplished by modifying the leading zero anticipate logic to prevent an indication of more than this radix point.

Also, shown is a case study for the new Power4 FPU which handles denormalized numbers in hardware. It uses a combination of tagging and prenormalization to prevent denormalization and to handle denormal input. The result is a processor which executes 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 Institute of Electrical and Electronic Engineers, Inc., New York, Aug. 1985.

[2] R. Yu and G. Zyner. "167 MHz Radix-4 Floating Point Multiplier," In *Proc. of Twelfth Symp. on Comput. Arith.*, pp. 149–154, Bath, England, July 1995.

[3] A. Naini, A. Dhablania, W. James, and D. D. Sarma. "1-GHz HAL SPARC65 dual floating point unit with RAS features," In *Proc. of Fifteenth Symp. on Comput. Arith.*, pp. 173–183, Vail, Colorado, June 2001.

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

[5] M. D. V. Dyke-Lewis and W. Meeker. "Method and apparatus for performing fast floating point operations," *U.S. Patent No. 5,966,085*, p. 7, Oct. 12, 1999.

[6] C. May, E. Silha, R. Simpson, and H. Warren, editors. *"The PowerPC Architecture: a specification for a new family of RISC processors,"* Morgan Kaufman Publishers, Inc., San Francisco, CA, 2002.

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

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

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

[10] S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan. "A general proof for overlapped multiple-bit scanning multiplications," *IEEE Trans. Comput.*, 38(2):172–183, Feb. 1989.

[11] T. Williams. "Method and apparatus for multiplying denomalised binary floating point numbers without additional delay," *U.S. Patent No. 5,347,481*, p. 16, Sep. 13, 1994.

[12] C. A. Krygowski and E. M. Schwarz. "Floating-point multiplier for de-normalized inputs," *U.S. Patent Application No. 2002/0124037 A1*, p. 8, Sep. 5, 2002.

[13] G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess. "High performance floating-point unit with 116 bit wide divider," In *Proc. of Sixteenth Symp. on Comput. Arith.*, Spain, June 2003.

[14] S. Gupta, R. Periman, T. Lynch, and B. McMinn. "Normalizing pipelined floating point processing unit," *U.S. Patent No. 5,267,186*, p. 10, Nov. 30, 1993.

[15] S. Gupta, R. Periman, T. Lynch, and B. McMinn. "Normalizing pipelined floating point processing unit," *U.S. Patent No. 5,058,048*, p. 11, Oct. 15, 1991.

[16] M. P. Taborn, S. M. Burchfiel, and D. T. Matheny. "Denormalization system and method of operation," *U.S. Patent No. 5,646,875*, p. 8, Jul. 8, 1997.

[17] E. Schwarz, B. Giamei, C. Krygowski, M. Check, and J. Liptay. "Method and system for executing denormalized numbers," *U.S. Patent No. 5,903,479*, p. 6, May 11, 1999.

[18] E. M. Schwarz and C. A. Krygowski. "The S/390 G5 floating-point unit," *IBM Journal of Research and Development*, 43(5/6):707–722, Sept./Nov. 1999.

[19] M. Urano and T. Taniguchi. "Method and apparatus for normalization of a floating point binary number," *U.S. Patent No. 5,513,362*, p. 15, Apr. 30, 1996.

[20] 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," *U.S. Patent No. 5,963,461*, p. 22, Oct. 5, 1999.

[21] A. I. Grushin and E. S. Vlasenko. "Computer methods and apparatus for eliminating leading non-significant digits in floating point computations," *U.S. Patent No. 5,732,007*, p. 34, May 24, 1998.

[22] S. D. Naffziger and R. G. Beraha. "Method and apparatus for bounding alignment shifts to enable at speed denormalized result generation in an FMAC," *U.S. Patent No. 5,757,687*, p. 13, May 26, 1998.

[23] A. A. Bjorksten, J. D.G. Mikan, and M. S. Schmookler. "Fast floating point results alignment apparatus," *U.S. Patent No. 5,764,549*, p. 9, Jun. 9, 1998.

[24] G. H. Handlogten. "Method and apparatus to perform pipelined denormalization of floating-point results," *U.S. Patent No. 5,943,249*, p. 10, Aug. 24, 1999.