The IBM zEnterprise-196 Decimal Floating-Point Accelerator

Steven Carlough
scarloug@us.ibm.com

Adam Collura
collura@us.ibm.com

IBM Systems and Technology Group
2455 South Road Poughkeepsie, New York 12601

Silvia Mueller
smm@de.ibm.com

Michael Kroener
mkroener@de.ibm.com

IBM Deutschland Research and Development GmbH
Schoenaicher Strasse 220, 71032 Boeblingen, Germany

***Abstract*—Decimal floating-point Arithmetic is widely used in commercial computing applications, such as financial transactions, where rounding errors prevent the use of binary floating-point operations. The revised IEEE Standard for Floating-Point Arithmetic (IEEE-754-2008) defined standardized decimal floating-point (DFP) formats. As more software applications adopt the IEEE decimal floating-point standard, hardware accelerators that support it are becoming more prevalent. This paper describes the second generation decimal floating-point accelerator implemented on the IBM zEnterprise-196 processor. The 4-cycle deep pipeline was designed to optimize the latency of fixed-point decimal operations while significantly improving the bandwidth of DFP operations. A detailed description of the unit and a comparison to previous implementations found in literature is provided.**

Keywords- Decimal Arithmetic, Decimal Fixed-Point, Decimal Floating-Point, Hardware Accelerators

I. INTRODUCTION

Decimal calculations are natural to human beings and have been the standard basis for numerical calculations for thousands of years. With the advent of the computer age, binary number systems have become popular because of the ease with which binary computational mechanisms can be implemented with physical devices [1]. However, binary numerical calculations are not adequate for many applications, particularly for transaction processing where correctly rounded decimal calculations are required for financial transactions [2]. In 2008, the IEEE ratified IEEE-754-2008 Standard for Floating-Point Arithmetic which contained a format and specification for performing decimal floating-point (DFP) arithmetic.

Since the ratification of the new IEEE standard, there has been a significant amount of work in the area of DFP hardware [3]. In 2006, IBM introduced DFP execution using millicode support on their z9 Enterprise server [5]. In 2007, IBM released the first DFP hardware accelerator on its Power platform [4], and in 2008, DFP hardware support was made available on the z10 Mainframe [6]. These accelerators were considerably faster than traditional software solutions but were designed to minimize area. Consequently, they were not pipelined. Other pursuits were made in academia such as [7], but utilized software traps for special cases the hardware did not support.

This paper describes the DFP Accelerator on the IBM zEnterprise-196 (z196) processor, which is the first fully IEEE compliant pipelined DFP execution unit available in a commercial product. The DFP Accelerator also executes the fixed-point decimal instructions critical on many commercial transaction workloads.

II. ARCHITECHURE

A. Overall Description

The IEEE-754-2008 standard for decimal floating-point describes two different DFP standards. The Binary Integer Decimal (BID) format uses a binary value to represent the mantissa value, and the Densely Packed Decimal (DPD) format represents the mantissa value with encoded Binary Coded Decimal (BCD) numbers [8].

Both the BID and DPD versions of the standard each have 32-bit, 64-bit and 128-bit formats and the two versions of the standard are numerically equivalent for all three formats. The DFP Accelerator on the z196 supports the DPD version of the standard, and provides full functional support for the 64-bit and 128-bit formats. Conversion instructions for the 32-bit format are provided such that operations on the 32-bit format can be emulated using 64-bit instructions. Furthermore, the architecture provides support for variable precision arithmetic, enabling software to round to any specific precision without rounding error accumulation.

Many factors were considered in determining the depth of the DFP Accelerator pipeline. Normally, a pipeline is designed to be deep enough to streamline the most common instructions the unit will execute. However, because fixed-point decimal instructions are so critical to customer workloads, whereas DFP workloads are not yet as common, the DFP Accelerator was optimized to minimize decimal fixed-point instruction latency, resulting in a 4-stage pipeline, operating at a frequency of 5.2GHz.

Area was also a factor in the design of the DFP Accelerator. Similar to previous designs, the DFP Accelerator shares a register file with the Binary Floating-Point Unit and special issue logic prevents write back collisions from occurring between the two units. The total area of the z196 DFP Accelerator is 1.43 mm^2 in a 45 nm technology.

B. Exceptions and Special Results

The exponent and mantissa hardware have been designed to handle results in both the normal as well as extreme number ranges. Extreme results typically yield underflow or overflow exceptions and involve adjusting the bias of the resulting exponent. However, unlike normalized binary floating-point numbers, unnormalized decimal floating-point

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.27

numbers have a small range of values referred to as quasi-supernormals that require special hardware or trap conditions.

An underflow exception occurs in decimal arithmetic if the infinitely precise intermediate result has a smaller magnitude than the target format can accurately represent. Specifically, if the intermediate exponent result is less than the format's smallest exponent, E_{min}, it does not necessarily indicate an underflow. If there are enough trailing zeros in the mantissa of the result such that the mantissa can be shifted to the right until the exponent is increased to E_{min} without loss of precision, then the result will not underflow. Similarly, there are cases where the intermediate result contains an exponent above the format's largest exponent, E_{max}, but enough leading zeros exist in the mantissa to left shift the result until its exponent reaches $E_{max.}$ Referred to as quasi-supernormal numbers, these results do not trigger overflow exceptions. Numbers in these two ranges leave the application no knowledge if the result obtained accurately represents the infinitely precise result, or if the result is actually a different cohort of the same value.

On past machines, supporting decimal floating-point number applications requiring correct cohort detection for extreme results required every computational instruction be followed by a "Test Data Class" instruction that would check the result for boundary exponents (E_{min} or E_{max}), infinity, 0, etc. This is costly to performance. The availability of the new quantum exception in the z196 processor, which detects when the result is of a different cohort than the unbounded intermediate result, eliminates the need for these additional instructions [9].

III. HARDWARE IMPLIMENTATION

The DFP Accelerator in the z196 processor is capable of performing IEEE-754-2008 computations in both the 64-bit and 128-bit DFP format. All IEEE compliant exceptions, flags and rounding modes are supported by the hardware. The hardware also computes overflow results, underflow results, and supports eight rounding modes. Figure 1 and Figure 2 show the mantissa and exponent dataflow respectively.

A. Mantissa Hardware

The dataflow for the mantissa hardware of the DFP Accelerator is shown in Figure 1. Operand data arrives from the floating-point registers (FPR) or bypass buses on two 64-bit buses. Data may be bypassed from the FPR write buffer, from the DFP Accelerator output, from one or both of the two memory load pipelines, or from the fixed-point unit. 128-bit operand data arrives over two cycles and requires a 1-cycle gap in the instruction stream to allow for the extra transfer cycle into the unit.

Operand data passes though the DPD to BCD converters on the next cycle where they are converted from the IEEE-754 format to BCD data if necessary. Leading zero detectors for both operands are connected to these converters, and the data is sent to the 144-bit operand registers, A1 and B1.

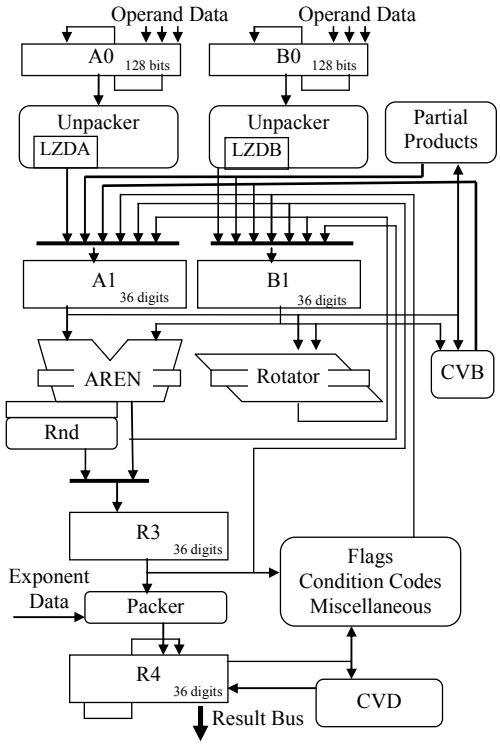

Figure 1. Decimal Floating-Point Accelerator Mantissa Dataflow.

The AREN is the Arithmetic Engine of the DFP Accelerator. It consists of a pipelined end-around carry decimal adder with injection based rounding. The AREN has a two-stage output that is capable of performing simple binary or decimal addition and subtraction operations. This faster two-cycle output is used to generate partial product accumulations during multiplication and division iterations, and for fixed-point decimal operations. For DFP addition and subtraction operations or during the rounding operations in the final stages of multiplication or division, the third cycle of the adder is used to generate a rounded result based on guard digits, a sticky bit and the rounding mode.

The operand data in the A1 and B1 registers that feed the AREN are pre-aligned based on the operation that the AREN will perform. For effective subtract operations, the data is left justified and the exponent is evaluated as though no effective cancellation occurred. If cancellation does occur, the result is left shifted one digit and the exponent is incremented. For addition operations, the data is aligned with one digit of zero padding on the left, and the exponent is set to assume that a carry out into the most significant digit will occur. If a carry out does not occur, the result is left shifted and the exponent is incremented by one. Using data pre-alignment, rounding hardware on two rather than three different decimal locations is needed, reducing logic on the timing critical path. Additionally, the need for an exponent decrementor is avoided.

The two-stage rotator is capable of performing left or right shifts up to thirty-four digits, and is pipelined over two

cycles. The AREN, rotator, and dataflow register selects can be configured as a single 36-digit pipeline, or as two parallel 18-digit pipelines, effectively doubling the hardware resources available for 64-bit DFP operations. A 6:2 compressor for decimal to binary conversion and special doubler hardware for binary to decimal conversion are included. The partial product generator is similar to that described in [4].

B. Exponent Hardware

The Exponent pipeline for the DFP Accelerator is shown in Figure 2. This 16-bit pipeline has been optimized to reduce area and power without sacrificing performance or impacting the processor's cycle time. The hardware shown is capable of handling the exponent processing for both the 16-digit and the 34-digit DFP formats.

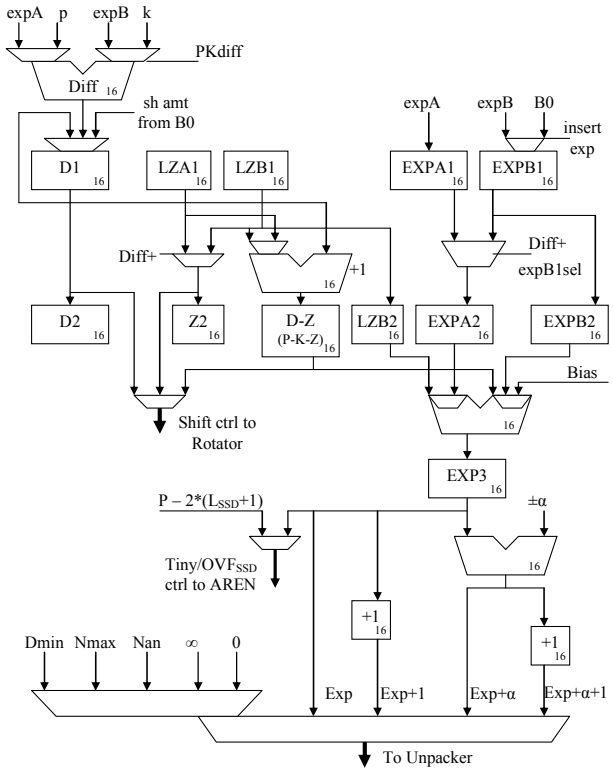

Figure 2. Decimal Floating-Point Accelerator Exponent Dataflow.

The exponent data flow calculates the rotator controls (shift amount and direction) and must also compute the correct exponent value for an operation accounting for any special conditions.

Operand exponent data arrives into the dataflow on buses expA and expB. An exponent difference calculation is computed ($D1 = Exp_{big}\text{-}Exp_{small}$) and used to determine how far the mantissa of the operand with exponent Exp_{big} must be shifted to the left to align the exponents for addition and subtraction operations. For operations that round to a smaller target significance, the precision of the format p and target precision k is multiplexed into the same adder evaluating $p\text{-}k$, which is used in the next cycle. In parallel, the results of the leading zero detectors for the operands shown in the mantissa dataflow are latched in LZA1 and LZB1.

If the difference from $D1$ is greater than the number of leading zeros in the mantissa of the operand with exponent Exp_{big}, that mantissa is normalized (shifted left by its LZD value Z) and the other mantissa is right shifted by $D1\text{-}Z$. The resulting exponent is evaluated as

$$Exp_R = Exp_{small} + D1 - Z$$

Note that for operations that round to a smaller target precision, the same adder is used to evaluate a right shift by $p\text{-}k\text{-}Z$. Reuse of these adders for both instruction types helps reduce the overall area required for the exponent dataflow.

The EXP3 latch contains Exp_R in the next cycle. Though evaluation of Exp_R has been discussed for addition and subtraction, Figure 2 shows how for multiplication or division, the same adder can be used to compute the sum or the difference of the exponents. Likewise, different bias amounts can be muxed into an operand of the same adder for operations that perform format conversion.

Exp_R is also sent to the AREN where it forms a mask to perform tininess detection used for underflow and quantum exception detection. The possibility of tininess occurs for values $Exp_R \leq p$. This is a left-aligned zero-digit detect mask. If no significant digits occur in the mask region created by these bits, the AREN will set the tininess indicator which could result in underflow or quantum exceptions. This same mechanism is used to detect fixed-point decimal overflow exceptions. The byte length of the target fixed-point decimal result (L_{SSD}) is subtracted from the number of digits of precision of the hardware (p=34). For fixed-point decimal operations, if any digits occur in the region covered by the mask, a fixed-point decimal overflow occurs.

The logic below EXP3 shows how special results, such as the smallest normal number (D_{min}) and the largest normal number (N_{max}), are multiplexed to the output stage. It also illustrates how the exponent can be adjusted by scaling factor α for underflow or overflow conditions [8]. A conditional increment is only needed for addition operations that do not overflow or for effective subtract operations that have cancellation on the most significant digit. The need for a decrement operation was removed by pre-aligning the mantissas and pre-adjusting the exponents based on the operation performed.

IV. Instruction Execution

A. Addition/Subtraction

Addition and subtraction are performance sensitive functions. Significant attention has been devoted to investigating new methods for improving DFP addition operations [7]. A method of injection based rounding is described in [10] that removes the requirement for a separate rounding stage as shown in [4]. Decimal end-around carry is also shown in [10], which allows for a single subtraction operation to evaluate the magnitude of an effective subtract operation without first determining which of the two operands are larger.

The AREN in the z196 DFP Accelerator has several unique features beyond what is found in literature. As mentioned in Section 2, special mask logic was added for tininess detection of small exponents. Based on the rounding mode, a value is added into the guard and sticky locations such that summation yields the correctly rounded result. The difficulty with using error injection for DFP operations is that it is not known exactly which digits are going to be the guard and sticky digits of the result. In addition operations, if the result of the addition operation carries out of the adder, the result is shifted one digit to the right and the exponent is incremented. Until the initial summation is complete, it is impossible to know the location of the guard digit requiring injection. For example, if the target result is p digits and the operands are aligned across p+2 digits, then the value of the carry out determines if there are p+2 or p+3 intermediate result digits. If there is no carry out, then the guard digit of the result aligns with the p+1th digit of the operands, but if the result does carry out, then the guard digit of the result aligns with the p^{th} digit of the operands.

A similar problem exists for subtraction. If the operands are aligned across more than p digits and the subtraction result contains a zero in the most significant digit, then the result must be left shifted one digit and the exponent decremented by one.

To simplify the injection based rounding process, a carry select adder was used to evaluate the sum independent of the carry into the adder. This carry select adder allows the carry selection to occur on either the p^{th} or p+1th digit of the adder. Subtract operations are normalized whereas addition operations are aligned with a pad-zero at the most significant digit making room for a possible carry digit. Based on the value of the most significant digit of the adder (non-zero on a carry out of an addition or no cancellation on subtraction) the appropriate carry-in is applied to either the p^{th} digit or the p+1th digit. By pre-aligning the mantissas this way, where effective cancellation on a subtraction is assumed to occur, the exponent hardware only needs a conditional increment; the added hardware for a conditional decrement is avoided.

B. Multiplication

The z196 DFP Accelerator employs a similar algorithm to perform 64-bit and 128-bit decimal multiplication as its predecessor [6], yet is more efficient and less complex when handling extreme numbers.

Given that a format precision is p digits in length, where multiplier

$$A = (\sum_{i=0}^{p} a_i \times 10^i) \times 10^{\mu}$$

with m significant digits is multiplied by multiplicand

$$B = (\sum_{j=0}^{p} b_j \times 10^j) \times 10^{\nu}$$

with n significant digits, the most significant p digits of the product can be calculated as

$$P = (\sum_{i=0}^{p} \sum_{j=0}^{p} a_i b_j \times 10^{j+i}) \times 10^{\mu+\nu-bias}$$

The core of the algorithm employs an iterative partial product generator to assemble the mantissa similar to [4]. It assumes the final product will be m+n digits in length. To compute the decimal mantissa of the product, the following steps are performed once the DFP operands have been converted to binary coded decimal in the Unpacker blocks shown in Figure 1.

First, m and n are determined to identify the operand with the least number of significant digits using leading zero detection logic built into the Unpackers (LZDA and LZDB). Given the format's precision p and LZD results, m and n are calculated.

Next, the operand with the most number of significant digits (m) is normalized such that the mantissa of this operand is multiplied by $1 \times 10^{p-m}$. If operand A had the most significant digits this would be expressed as

$$A' = (\sum_{i=0}^{p} a_i \times 10^{i+p-m}) \times 10^{\mu}$$

This step eliminates final product significance calculations and facilitates precision and range adjustments at extreme exponent values without result LZD hardware.

Next, the operand with the least number of significant digits (n) is stored into a linear shift register, where one digit will be extracted at a time to select multiples of the larger operand.

Generation of 1x, 2x, 5x, and 10x multiples of the operand with the larger number of significant digits is performed in the partial product block of Figure 1. These values are used to assemble partial products similar to the recoding technique in [16]. They are available on the fly; only 1x is stored in a side register, as 2x and 5x are generated using doubler and quintupler circuits, and 10x is a left shift by one.

Iteration on the n digits of the operand with the least number of significant digits is then performed. For the 64-bit format, two partial product digits are retired every two cycles for a net of one partial product digit per cycle. For the 128-bit format, one partial product digit is retired every two cycles for a net of ½ partial product digits per cycle. Exploiting the high and low parts of the AREN, which can operate as independent 18-digit adders, allowed for faster partial product generation for the 64-bit format since the amount of adder hardware effectively doubles. The number of iterations assumes m+n digits in the final product; this is corrected later if necessary.

Only the most significant p+1 digits of the intermediate product are available after the iterative process has completed. Other designs such as [4], [11], [12] and [13] store 2*p digits, resulting in additional hardware as well as various attempts to compute sticky bit information required for correct IEEE-754-2008 rounding. The z196 DFP Accelerator design shifts the intermediate partial product result to the right each iteration. If the product exceeds the

precision p, hardware examines the digits shifted out to compute the sticky bit dynamically.

Once the intermediate product is complete, it is still left aligned in the registers and must be shifted right to bring the result within the precision of the given format. Effectively, it is now divided by $1\times10^{p-m}$ plus an additional amount. The right shift amount, S, is determined according to Table 1.

Table 1 Right shift amount based on operand significance and format precision

	64-bit, n even	64-bit, n odd	128-bit
$m+n < p$	$2p+1-m-n$	$2p-m-n$	$p-m-n$
$m+n \geq p$	$p+1$	p	0

Since the result was produced left-aligned, the most significant digit of the left-aligned intermediate product is examined to determine if it is equal to 0 ($m+n$-1 digits) or non-zero ($m+n$ digits). Other designs such as [12], [14] and [6], employed a leading zero detection on the intermediate product to determine the final number of significant digits. The z196 DFP Accelerator only requires LZD hardware on the input operands.

The right shift amount is now conditionally decremented by one if it is determined that there are $m+n$-1 digits in the result product, but only if $m+n > p$, since products less than p digits suffer no loss of precision with a right shift. To summarize, there is:

no correction when

($m+n$ digits in result ***and*** $m+n>p$) ***or*** ($m+n \leq p$)

correction when

($m+n$-1 digits in result ***and*** $m+n>p$)

Additional correction to the right shift amount is factored in as *exp_delta*, which is described in detail below. This value corrects for extreme values near or just outside the range of the format. It requires no additional shift operations, but is applied as an adjustment to the single shift right amount shown in Table 1. Digits of precision lost during the right shift are collected into the guard digit and sticky bit inputs to the AREN. This least significant guard digit and sticky information are outside the precision of the given format, but participate in the rounding operation.

To compute the exponent of floating-point decimal multiplications, the multiplicand and multiplier exponents are summed into working exponent $E_{work} = \mu + \nu - bias$ and stored in EXP3 shown in Figure 2.

E_{work} is adjusted, if necessary, to account for products that are larger than the format's precision, assuming that the final result will have $m+n$ significant digits:

$m+n > p$: $E_{work} = E_{work} + ((m+n) - p)$

$m+n \leq p$: $E_{work} = E_{work} + 0$

Next, the final exponent adjustment computed as *exp_delta* is factored into final product coefficient right shift. The shift amount S before $m+n$-1 correction becomes

$$S' = S - exp\_delta$$

for the following values:

1. $E_{work} < E_{min}$: the product is a subnormal number, therefore right shift an additional amount to get the product precision and range within the format specifications:

$$exp\_delta = E_{min} - E_{work}$$

When $(E_{min} - E_{work}) > p$, all the digits are shifted to the right into the guard and sticky information.

2. $E_{min} \leq E_{work} \leq E_{min}+p$: the product is a subnormal number but is within the precision and range of the format and no exponent induced additional shifting of the coefficient is required:

$$exp\_delta = 0$$

3. $E_{min}+p < E_{work} \leq E_{max}$: the product is within the normal range of numbers and no additional exponent induced shifting of the coefficient is required.

$$exp\_delta = 0$$

4. $E_{max} < E_{work} \leq E_{max}+p$: the product is a supernormal number. If the number of significant digits in the result is less than the format precision, we may be able to bring the result to within the range of the format:

$$exp\_delta = E_{max} - E_{work}$$

This is effectively a left shift, since the right shift amount is reduced by a negative *exp_delta.* An overflow condition occurs when $(E_{work} - E_{max}) > (p - (m+n))$

5. $E_{max}+p < E_{work}$: the result is beyond the precision and range of the format and an overflow condition exists.

For fixed-point decimal multiplications, μ and ν are forced to either the value of the DFP64 or DFP128 floating-point bias, based on the fixed-point precision. This effectively disables any special exponent handling that could be invoked by the exponent hardware during the execution of the fixed-point operation.

The multiplication operation is implemented to support IEEE overflow and underflow based on the settings of these respective masks. One additional cycle is required when the IEEE overflow mask is set and the intermediate product is supernormal, as it needs to be determined whether or not to rebias the result or report the precise final product. Both results are computed and a late select is used to determine the appropriate result.

The additional algorithmic cost of nominalization was offset by the elimination of the result LZD hardware, which on z10 was about 3% of the accelerator's area footprint. The generality of the algorithm for subnormal, normal and supernormal numbers also reduced area and power by an estimated 10% [6]. Utilization of other techniques, such as parallelization of partial product summations as explored in [14] or [15], would provide an exceptional improvement at computing the product mantissa. Such techniques would reduce the latency of the multiplication process by 4 to 10 times, depending on the number of significant digits in the operands. However, these techniques require approximately 50% more area than the entire z196 DFP Accelerator currently utilizes. The multiplication technique used on the z196 DFP Accelerator was designed to minimize area and the area dedicated to logic exclusively used by multiplication is only 10% of the total footprint. The significant additional area required for parallel multiplication techniques prevented their use in the z196 DFP Accelerator design. Achieving the aggressive cycle time of the z196 processor would also be challenging for a

parallel multiplication algorithm given the CSA depth described in [13] and [15].

C. Division

The z196 DFP Accelerator uses a non-restoring decimal division algorithm to compute the mantissa of the resultant quotient, similar to that used on the Power6 decimal floating-point unit [17]. The algorithm is capable of producing correctly rounded results for both the 16-digit and 34-digit DFP formats. Likewise, the same technique is also used for the fixed-point decimal divide instructions, but rather than executing the rounding procedure, the fixed-point decimal instructions must instead perform a final subtraction to generate the exact remainder.

The algorithm employs a redundant radix-10 non-restoring division that applies pre-scaling to reduce area and overall delay. An extra partial remainder calculation is computed each iteration to reduce the number of stored multiples necessary without adding additional delay.

The quotient can be expressed as

$$Q = \sum_{i=0}^{p-1} q_i \times 10^i$$

where each q_i is determined from the equation

$$q_{i+1} = P_i \times D$$

D is the divisor and Pi is the partial remainder evaluated as

$$P_{i+1} = 10 \times (P_i - q_{i+1} \times D)$$

By pre-scaling the divisor and dividend such that the pre-scaled divisor D' is in the range of $(1.0 \le D' \le 1.1)$, the quotient selection of q_{i+1} can be reduced to the most significant digit of P_i, removing the cycle that would have been necessary for a lookup table. The pre-scaling process adds seven cycles to the startup overhead of the divide operation, but reduces the number of cycles per iteration from five to four.

A significant difference between the division operation on the z196 DFP Accelerator and previous literature is the quantum exception previously described. For exact quotient results where the exponent is close to but outside the normal exponent range, it is possible for the mantissa to be adjusted within the p digit format such that an exact result can be expressed without overflow or underflow. The quantum exception can be used to indicate to software when these cases occur.

D. Binary to Decimal Conversion

Binary to decimal conversions are used to convert binary integer data into either decimal floating-point values or may be used to convert data to the more traditional fixed-point decimal values.

The main computational engine for converting binary data into decimal data is the CVD block shown in Figure 1. The CVD block converts four bits of binary data into decimal data each cycle by using four modified decimal doubler circuits. Decimal doubler circuits are normally only a few gate delays deep, but to fit four doublers into a cycle, the digits had to be encoded and decoded on the fly so the doublers could be reduced to only two gate delays each.

A decimal format referred to as 8,6,4,2,1 is defined as follows: if X is a 4-bit BCD number with value [0-9], then the value of $X_{[8,6,4,2,0,1]}$ is encoded as shown in Table 2.

Table 2 Decimal 8,6,4,2,1 Coding used for Binary to Decimal Conversions

Value	Encode		Value	Encode
0	000010		5	001001
1	000011		6	010000
2	000100		7	010001
3	000101		8	100000
4	001000		9	100001

The encode process requires a single gate delay, but reduces each of the four subsequent doubler stages to only two gate delays each as shown from the doubling equations as follows:

If $Y_{[8,6,4,2,0,1]} = 2*X_{[8,6,4,2,0,1]}$ then:

$$Y_{[8]} = X_{[8]} \text{^} X_{[1]} + X_{[4]} \text{^} X'_{[1]}$$
$$Y_{[6]} = X_{[2]} \text{^} X_{[1]} + X_{[8]} \text{^} X'_{[1]}$$
$$Y_{[4]} = X_{[6]} \text{^} X_{[1]} + X_{[2]} \text{^} X'_{[1]}$$
$$Y_{[2]} = X_{[0]} \text{^} X_{[1]} + X_{[6]} \text{^} X'_{[1]}$$
$$Y_{[0]} = X_{[4]} \text{^} X_{[1]} + X_{[0]} \text{^} X'_{[1]}$$
$$10*Y_{[1]} = X_{[8]} + X_{[6]} + X_{[4]} \text{^} X'_{[1]}$$

where $10*Y_{[1]}$ becomes the $X_{[1]}$ in the next higher digit in the next doubler stage. A single gate delay is required to convert each digit of the result $X_{[8,6,4,2,0,1]}$ to the BCD format $X'_{[8,4,2,1]}$ as follows:

$$X'_{[8]} = X_{[8]}$$
$$X'_{[4]} = X_{[4]} + X_{[6]}$$
$$X'_{[2]} = X_{[2]} + X_{[6]}$$
$$X'_{[1]} = X_{[1]}$$

E. Decimal to Binary Conversion

Binary to decimal conversions are used to convert DFP mantissas or fixed-point decimal numbers to binary integers. Similar to other techniques described in literature, the DFP Accelerator uses an iterative algorithm to perform the conversion process.

The algorithm converts three decimal digits every iteration. The conversion of a k-digit decimal number D with digits $d_0 d_1 \ldots d_{k-1}$ to binary integer B can be expressed as

$$B = \sum_{i=0}^{\lceil (k-1)/3 \rceil} \left(d_{3i} + d_{3i+1} \times 10 + d_{3i+2} \times 100 \right) \times 1000^i$$

Three digits per iteration provides the best performance with a reasonable amount of hardware resources. Converting more digits per cycle require binary multiplies of 10,000 and were found to be too costly for the area resources available. Past implementations such as those used on [18] have also used three-digit per cycle iterations where each iteration can be expressed as

$$sum_i = sum_{i-1} \times [1000] + d_{3i+2} \times [100] + d_{3i+1} \times [10] + d_{3i}$$

The past implementation performed a binary multiplication of 1000 using multiples of 1024, -16, -8 as a series of shifts and additions over three cycles. Lookup tables were used to generate the $d_{3i+2} \times 100$ and $d_{3i+1} \times 10$ terms. Instead, the CVB process in the z196 DFP Accelerator

creates a multiplication of 1000 using multiples of 1024, -32, and 8. The $d_{3i+2} \times 100$ term is realized with multiples 64, 32, 4, and $d_{3i+1} \times 10$ is realized with multiples 8 and 2. By utilizing different sets of partial product terms, three decimal digits can be converted and accumulated each cycle using a simple 6:2 compressor with some modifications to the carry terms. The expression for the redundant summation from the 6:2 compressor each iteration can be expressed as

$$S_i + C_i = \begin{Bmatrix} S_{i-1} \times \left[2^{10} - 2^5 + 2^3\right] + C_{i-1} \times \left[2^{10} - 2^5 + 2^3\right] \\ + X \times \left[2^6 + 2^5 + 2^2\right] + Y \times \left[2^3 + 2^1\right] + Z \end{Bmatrix}$$

Table 3 shows how the partial product terms can be arranged in the 6:2 compressor hardware. For the purposes of formatting, the table illustrates how the bits are arranged for the second iteration where $d_{a,b}$ refers to the a^{th} digits of *D* and *b* refers to the bit weight of each bit of *d* in 8,4,2,1 BCD notation. *S* and *C* refer to the sum and carry terms output from the 6:2 compressor on the previous iteration.

Table 3 Bit organization of a 6:2 compressor for Decimal to Binary Conversions

Least significant bits of 6:2 compressor													
S	*S*	*S*	*S*	$d_{5,8}$	$d_{5,4}$	$d_{5,2}$	$d_{5,1}$	$d_{5,8}$	$d_{5,4}$	$d_{5,2}$	$d_{5,1}$	0	0
C	-	-	-	$d_{5,8}$	$d_{5,8}$'	$d_{5,4}$	$d_{5,2}$	$d_{5,1}$	$d_{4,8}$	$d_{4,4}$	$d_{4,2}$	$d_{4,1}$	0
S'	*S'*	*S'*	*S'*	*S'*	*S'*	*S'*	*S'*	*S'*	-	$d_{3,8}$	$d_{3,4}$	$d_{3,2}$	$d_{3,1}$
C'	*C'*	*C'*	*C'*	*C'*	*C'*	-	$d_{4,8}$	$d_{4,4}$	$d_{4,2}$	$d_{4,1}$	0	0	0
S	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	0	0	0
C	*C*	*C*	*C*	*C*	*C*	*C*	*C*	1	0	0	0	0	0

The sum and carry terms from the 6:2 compressor are sent to the AREN where a binary addition and conditional complementation is performed to evaluate the final result. Although the 6:2 compressor requires more area than the design implementation described in [18], the number of cycles per iteration was reduced by a factor of three.

V. Error Detection

The reliability, availability, and serviceability (RAS) of the z196 and for any Enterprise mainframe system is extremely important. Any computational error resulting from system noise, alpha particle impact, beta particle decay, circuit failure, or other factors must be detected by the hardware [19]. If an error is detected in the DFP Accelerator hardware, or any other part of the z196 processor, an error signal is sent to a Recovery Unit where action can be taken by the system to flush the erroneous data and then restart the instruction stream from an error clean state.

To detect errors between the DFP Accelerator and the surrounding units, every input and output signal is protected with 8-bit parity. Once the data and instructions enter the Accelerator logic, multiple layers of error checking are employed to ensure the hardware does not produce an erroneous result.

The exponent dataflow and the control logic detect errors through logic duplication. An identical copy of each of the pipeline control and exponent dataflow macros were added to the hardware and were sourced by staging latches containing the state of the signals driving the functional copy of the macros. The exponent logic was duplicated because it contains a significant amount of control state (such as masks and shift controls) which are difficult to properly cover with computational methods such as RAS. Because the error detection copies of the duplicated macros ran one cycle behind the functional copies, timing was not impacted by the extra load and wiring that was needed. This "brute force" method of error detection is highly effective but too expensive in area and power to do to the entire unit. The net impact to area for logic duplication on the z196 DFP Accelerator was less than 14%. Control state that was not in duplicated macros, such as the sequencer logic for the multiplication, division, and addition operations, was either protected by parity or through duplication of selected critical hold latches, compared against each other.

Residue modulus 3 (Res3) and residue modulus 9 (Res9) are used to detect errors that occur in the dataflow using two overlapping methods described below. Single bit errors that occur in decimal computations are detected with Res9, whereas Res3 is utilized to detect single bit errors in operations that entail conversions between binary and decimal numbers [20]. Res9 is evaluated for each of the operand registers A1, B1, R3, and R4 as shown in Figure 1. The Rotator computes the Res9 for any digits that are shifted out. Furthermore, Res9 is computed and stored to provide error detection for the values contained in the Partial Products macro used for multiplication and division. Residue provides significant error coverage for minimal power and area resources.

The first method of error detection, referred to as component detection, checks the Res9 at the destination latch against the expected Res9 value based on the Res9 of the source data and the function applied. For example, when data is simply moved from one register to another, the Res9 of the source register is compared to the Res9 of the target register. When data is shifted through the rotator, the Res9 of the data shifted out of the Rotator is subtracted from the Res9 of the source register and is compared to the Res9 of the target register. When the AREN is used, the predicted Res9 is computed as a function of the Res9 of the A1 and B1 registers along with the AREN operation (addition or subtraction). This predicted value is then compared to the target register Res9. If the rounded result is used, the predicted Res9 is conditionally adjusted based on the rounding mode, guard and sticky bit values, which are all duplicated in the residue prediction logic for the AREN.

The second method of error detection used in the DFP Accelerator is instruction level detection. Based on the input operands, rounding mode, sign information and operator, the Res9 for the entire addition or subtraction operation can be computed. The operation can then be computed in the hardware and the Res9 of the result can be compared to the predicted value to check for an error. When data is shifted into the guard and sticky locations, the predicted Res9 must be adjusted for these digits that will not participate in the final summation. Similar to how the rotator is checked in the component error detection described above, the Res9 of the

digits shifted out of the rotator is used to adjust the predicted Res9 of the addition/subtraction operation.

Multiplication operations are also checked using this technique. Based on the Res9 of the input operands, the Res9 of the final product can be determined. If the final product contains more digits then the target precision, they are shifted into the sticky bit location. As each digit is shifted into the sticky bit, its Res9 is computed and the predicted Res9 of the result is adjusted accordingly.

Instruction level checking is the only method used to detect errors in the CVB and CVD hardware. The Res3 of the binary integer input for a CVD operation is compared to the Res3 output of the CVD operation. The Res3 of the decimal input for a CVB operation is compared to the Res3 of the binary output. If the operation executes without error, the two residues will match.

VI. COMPARISONS

Providing a workload comparison for the DFP Accelerator against previous implementations is difficult as there are not many mainstream benchmarks available. This analysis compares the number of cycles necessary for common functions to previous implementations and software libraries [21]. Table 4 shows the results of this comparison:

Table 4 Comparison of Frequent DFP Operations

Operation	z196	z10	Power6	Software Libraries
Add DFP64	6	12-28	13-21	154
Add DFP128	8	16-31	15-23	233
Mult DFP64	13-39	16-55	24-39	296
Mult DFP128	15-87	17-104	25-93	453
Compare	8-9	11-14	14-20	289-580
Divide	16-140	17-193	36-154	627-940
CVB	10	14-19	11-27	N/A
CVD	12-16	15-24	12-43	N/A
Format	1-3	9-11	5-8	N/A
Quantize	8-10	12-24	14-22	138-211
Reround	9-11	13-28	13-14	178-269

VII. CONCLUSION

This paper described the DFP Accelerator on the zEnterprise-196 processor, the first fully IEEE compliant pipelined DFP execution unit available in a commercial product. The DFP Accelerator also executes the fixed-point decimal instructions critical on many commercial transaction workloads. The 4-cycle deep pipeline was designed to optimize for the latency of traditional Fixed-Point Decimal operations at the cost of DFP instruction throughput. The shorter pipeline also limited the area requirements of the accelerator hardware to 1.43 mm^2 in a 45 nm technology.

The DFP Accelerator for the zEnterprise-196 processor was designed to meet the aggressive frequency of 5.2GHz while at the same time providing significant error detection.

REFERENCES

[1] W.S.Brown, P.L.Richman, "The Choice of Base," *Communications of the ACM*, vol. 12, pp. 560-561 ,Oct 1969.

[2] M.F.Cowlishaw, E.M.Schwarz, R.M.Smith, C.F.Webb, "A decimal floating-point specification," *proc. 15th IEEE Symposium on Computer Arithmetic*, 2001, pp 147-154

[3] L.K.Wang, M.A.Erle, C.Tsen, E.M.Schwarz, M.J.Schulte, "A survey of hardware desgins for decimal arithemetic," *IBM J. Res. & Dev.* Vol. 54, no. 2 pp 8:1-8:15

[4] L.Eisen, J.Ward, H.Tast, N.Mading, J.Leenstra, S.Mueller, C.Jacobi, J.Preiss, E.Schwarz, S.Carlough, "IBM POWER6 accelerators: VMX and DFU," IBM J. Res. & Dev., vol. 51, no. 6, pp. 663-684, Nov. 2007

[5] A.Y.Duale, M. H.Decker, H. G. Zipperer, M. Aharoni, T. J. Bohizic, "Decimal floating-point in z9: An implementation and testing perspective," *IBM J. Res. & Dev*., vol. 51. no. 1/2, pp. 217-228, Jan. Mar. 2007

[6] E.M. Schwarz, J.S. Kapernick, M.F. Cowlishaw. "Decimal Floating-point Support on the IBM System z10 processor." *IBM Journal of Research and Development*, Vol.53 No.1, 2009.

[7] L.K.Wang, M. Schulte, "Decimal Floating-Point Adder and Multifunction Unit with injection-Based Rounding," *IEEE Symposium on Computer Arithmetic*, pp. 56-68, 2007.

[8] IEEE Standard for Floating Point Arithemetic, IEEE Std. 754-2008, Aug. 2008.

[9] IBM Corporation, *z/Architecture Principles of Operation*, Aug. 2010

[10] A.Vazquez, E.Antelo, "A high-performance Signicand BCD Adder with IEEE 754-2008 Decimal Rounding," Proc. 19th IEEE Symp Computer Arithemetic, 2009, pp. 135-144.

[11] M.A. Erle, M.J. Schulte. "Decimal Multiplication via Carry-Save Addition." *Proc. IEEE Int'l Conference on Application-Specific Systems, Architecture, and Processors*, pp. 348-358, June 2003.

[12] C. Minchola, G. Sutter. "A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier." *Int'l Conference on Reconfigurable Computing and FPGAs*, pp. 59-64, 2009.

[13] M.A. Erle, B Hickmann, A.Krioukov, M.J. Schulte. "A Parallel IEEE P754 Decimal Floating-Point Multiplier." *Int'l Conference on Computer Design*, pp. 296-303, 2007.

[14] R. Raafat, A.M. Abdel-Majeed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, H.A.H. Fahmy. "A Decimal Fully Parallel and Pipelined Floating Point Multiplier." *IEEE Asilomar Conference on Signals, Systems and Computers*, pp. 1800-1804, 2008.

[15] A. Vazquez, E. Antelo, P. Montuschi. "A New Family of High-Performance Parallel Decimal Multipliers." *IEEE Symposium on Computer Arithmetic*, pp. 195-204, 2007.

[16] M.A. Erle, E.M. Schwarz, M.J. Schulte. "Decimal multiplication with efficient partial product generation*." Proc. IEEE Symposium on Computer Arithmetic*, pp. 21-28, 2005.

[17] S.Carlough, E.Schwarz, "Power6 Decimal Divide," *Proc. 18th IEEE Symp. Application Specific Signals, Systems, Architectures and Processors*, 2007, pp. 128-133.

[18] F.Busaba, C.Krygowski, W.Li, E.Schwarz, S.Carlough, "The IBM z900 Decimal Arithemetic Unit," Proc 35th *Asilomar Conf Signals Systems and Computers*, vol.2, pp. 1335-1339, Nov. 2001

[19] L.Spainhower, T.Gregg, "IBM S/390 Parallel Enterprise Server G5 fault tolerance: A historical perspective," IBM J. Res. & Dev., vol. 43, no. 5/6 Sept/Nov 1999

[20] D. Lipetz, E. Schwarz "Self Checking in Current Floating-Point Units," *IEEE Symposium on Computer Arithmetic*, 2011.

[21] M.Anderson, C.Tsen, L.Wang, K.Compton, M.Schulte, "Performance Analysis of Decimal Floating-Point Libraries and Its Impact on Decimal Hardware and Software Solutions," IEEE Int. Conf on Computer Design, pp.465-471, 2009