High Performance Floating-Point Unit with 116 bit wide Divider

Guenter Gerwig, Holger Wetter, Eric M. Schwarz, Juergen Haess
IBM Server Division
ggerwig@de.ibm.com, hwetter@de.ibm.com, eschwarz@us.ibm.com, jhaess@de.ibm.com

Abstract

The next generation zSeries floating-point unit is unveiled which is the first IBM mainframe with a fused multiply-add dataflow. It supports both S/390 hexadecimal floating-point architecture and the IEEE 754 binary floating-point architecture which was first implemented in S/390 on the 1998 S/390 G5 floating-point unit. The new floating-point unit supports a total of 6 formats including single, double, and quadword formats implemented in hardware. The floating-point pipeline is 5 cycles with a throughput of 1 multiply-add per cycle. Both hexadecimal and binary floating-point instructions are capable of this performance due to a novel way of handling both formats. Other key developments include new methods for handling denormalized numbers and quad precision divide engine dataflow. This divider uses a radix-4 SRT algorithm and is able to handle quad precision divides in multiple floating-point and fixed-point formats. The number of iterations for fixed-point divisions depend on the effective number of quotient bits. It uses a reduced carry-save form for the partial remainder, with only 1 carry bit for every 4 sum bits, to save area and power.

1. Introduction

This paper describes a future floating-point unit (FPU) of a high performance microprocessor which is optimized for commercial workloads. The FPU implements two architectures: Binary Floating-Point (BFP) which is compliant with the IEEE 754 Standard [1], and Hexadecimal Floating-Point (HFP) as specified by IBM S/390 Architecture[2] which is now called z/Architecture[3]. There are a total of 6 formats supported which include single, double, and quadword formats for the two architectures as shown in the following table:

Format	bits	sign	exponent	significand	bias
BFP short	32	1	8	24	127
BFP long	64	1	11	53	1023
BFP quad	128	1	15	113	16383
HFP short	32	1	7	24	64
HFP long	64	1	7	56	64
HFP quad	128	1	7	112	64

Unlike many other microprocessors, zSeries microprocessors implement quad precision operations in hardware, and this includes support for both HFP and BFP architectures.

Prior zSeries floating-point units have included the 1996 G3 FPU [4], the 1997 G4 FPU [5, 6], the 1998 G5 FPU [7, 8], the 1999 G6 FPU and the 2000 z900 FPU [9]. Most are remaps of the G5 FPU with extensions for 64-bit integers. The G4 FPU has an aggressive cycle time and can complete a multiply or add in about 3 cycles with a throughput of 1 per cycle. The G5 FPU is the first FPU to implement both BFP and HFP architectures in hardware on one pipeline. The G5 FPU design is based on the G4 FPU so it has the same latency for HFP instructions. BFP instructions involve translating the operands to HFP format, performing the arithmetic operation including rounding and then converting back to BFP format. So, BFP operations take 5 or 6 cycles of latency with a throughput of only one BFP instruction every two cycles.

The G5 FPU was designed with only one year between its announcement and that of the G4 FPU. So, the BFP arithmetic implementation is not optimized for speed, but instead for simplicity. With a longer development schedule for the next zSeries FPU, there were a few new goals: 1) optimize for BFP, 2) optimize for multiply-add, and then 3) optimize for HFP. The first goal was chosen due to the increase of new workloads on zSeries, particularly workloads utilizing Linux. These applications are typically written in Java or C++ and, especially those written in Java, rely on BFP even in commercial applications.

Thus, the primary goal was to create a high performance implementation much like the pSeries workstations. One key element of pSeries floating-point units is that the dataflow supports a fused multiply-add which effectively yields two operations per cycle. Since this type of design is optimal for BFP architectures, a decision was made to base our design on the Power4 design.

The Power4 floating-point unit has a 6 stage binary multiply-add dataflow. It uses tags in the register file to identify denormalized data. It has only 2 data formats, BFP single and double with double format retained in the register file. The major enhancements of our new zSeries FPU to the Power4 design are:

1. Two architectures are supported (HFP and BFP) which results in 6 formats versus only 2 formats of BFP, and 200 different instructions are implemented directly in hardware.
2. The pipeline is reduced to 5 cycles.
3. Denormalized number handling is supported without tags or prenormalization.
4. The normalizer and LZA are expanded to full width.
5. Division and square root are implemented with a quad precision radix-4 SRT algorithm.

These items will be detailed in the remainder of this paper. First, implementing two architectures in one dataflow will be discussed. Then, the overall dataflow will be described along with particular enhancements including handling of denormalized operands and the divide implementation.

2. Dual Architectures

The first machine to implement both BFP and HFP architectures in hardware is the 1998 IBM S/390 G5 processor[7]. A hexadecimal dataflow is used which requires binary operands to be converted to hexadecimal operands before they are operated on. The HFP instructions are capable of performing one add or one multiply per cycle with a latency of about 3 cycles. The BFP instructions can only be pipelined one instruction every other cycle and the latency is 5 or 6 cycles due to the extra conversion cycles and rounding cycle.

The problem with optimizing the dataflow for both HFP and BFP architectures centers on the choice of an internal bias. HFP architecture has a bias of the form 2^{n-1} whereas BFP has a bias of the form $(2^{n-1} - 1)$. To choose one of the biases as the internal bias and to convert to the other format requires shifting the significands and adding constants to the exponent. To avoid a conversion cycle, a separate internal representation

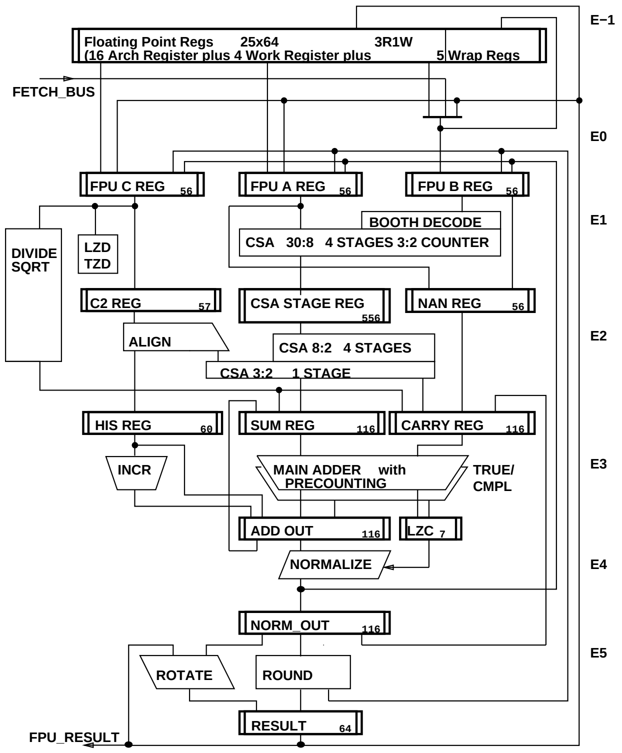


Figure 1. Main Fraction Dataflow of FPU

and bias was chosen for both architectures as shown by the following:

$$\begin{aligned}
X_{BFP_i} &= (-1)^{X_s} * \ (1 + X_f) \ * 2^{X_e - bias_{Bi}} \\
bias_{Bi} &= 2^{n-1} - 1 = 32767 \\
X_{HFP_i} &= (-1)^{X_s} * \ X_f \ * 2^{X_e - bias_{Hi}} \\
bias_{Hi} &= 2^{n-1} = 32768
\end{aligned}$$

This results in no conversion cycles and the dataflow is optimized for both architectures. This requires two different shift amount calculations since the biases differ and the implied radix points differ, but this is a very small amount of hardware.

3. Dataflow Overview

Figure 1 shows the fraction dataflow. At the top of the figure there is the Floating-Point Register file (FPR) with 16 registers of 64 bits each. There are also 5 wrap registers to hold data for loads. Loads are staged through the 5 wrap registers and the dataflow. Loads can be bypassed from any stage in the pipeline to a dependent instruction by using the wrap registers. This eliminates wiring congestion in the FPU dataflow stack and instead localizes it to the register file. When a read of an operand occurs, the data can come from the architected register file, the wrap registers, or a wrap back path from the dataflow, or from memory. In one cycle three registers of 64 bits can be read and one register can be written.

The dataflow is a three operand dataflow, which has a fused multiply and add data structure. One multiplier operand and the addend always come from the FPRs, while the 2nd operand may come from memory. In the starting cycle (labeled E0), the A,B and C registers are loaded with the correct formatting applied, such as zeroing the low order bits of a short precision operand. For binary formats the 'implied one' bit is assumed to be always '1'. If a denormalized number is detected afterwards, this is corrected in the multiplier and/or the aligner logic.

In the first execution cycle (E1), the shift amount for the alignment is calculated (considering potential denormalized operand cases). Also, the multiplication is started with Booth encoding and the first 4 stages of 3:2 counters of the Wallace tree. If there is an effective subtraction, the addend is stored inverted in the C2 register.

In the second execution cycle (E2), the alignment uses the previous calculated shift amount. In the multiplier, the next 4 stages of 3:2 counters reduce the tree to two partial products. These partial products with the aligned addend go through the last 3:2 counter to build the 'sum' and 'carry' of the multiply and add result. To balance the paths for the timing, the propagate and generate logic is performed also in this cycle. The propagate and generate bits are stored in a register instead of the sum and carry bits. A potential high part of the aligner output is stored in the high-sum register (HIS reg).

In the third execution cycle (E3), the main addition takes place. There is a 'True' and a 'Complement' Adder to avoid an extra cycle for recomplementation. Essentially, both $A - B$ and $B - A$ are calculated and the result is selected based on the carry output of the true adder. The number of leading zero bits is calculated using a zero digit count (ZDC) as described in [4]. This algorithm performs a zero digit count on 16 bit block basis of SUM and $SUM + 1$. When the carries are known the result is selected among the digits. The aligner bits which did not participate in the add are called the high-sum and they feed an incrementer in this cycle. At the end of this cycle there is a multiplexor which chooses between high-sum and high-sum plus one and also chooses whether to shift the result by 60 bits. If the high-sum is non-zero, the high-sum and upper 56 bits of the adder output are chosen to be latched. If instead the high-sum is zero, only the bits of the adder output are latched. Also the leading zero count is stored in the LZC register.

In the fourth execution cycle (E4), the normalization is done. The stored leading zero count is used directly to do the normalization. No correction is necessary, since the LZC is precise. For hex formats, only the two low order bits of the leading zero count are not used to get the normalized hex result. Additionally, the sticky bits are built according to the format.

In the fifth execution cycle (E5), the rounding and reformatting is done. For hex operands no rounding is needed, but the operands will pass this cycle anyway. Since there is a feedback path from the normalizer to the A, B, and C registers, this does not cost performance.

4. Denormalized Input

The architecture supported is a CISC type architecture and supports both register and memory operands as input to arithmetic instructions. This makes it very difficult to tag input operands in a timely manner. Since data from memory arrive late there is no time to check whether it is denormalized or normalized. The check requires examining the exponent to see if it is all zeros which would require for short and long operands an 8 way and an 11 way NOR function. Detecting denormalized input is instead calculated in the first cycle of execution.

In an implementation of hexadecimal floating-point the operands are 56 bits wide which requires 29 partial products for a radix-2 Booth encoding. 7 levels of 3:2 counters can only handle a maximum of 28 partial products, so 29 must take 8 levels. Additional correction terms do not add significant delay. Our new FPU assumes that the BFP operands are normalized and have an implied one. It corrects the multiplier Y, prior to creating the most significant partial product. The Booth decode term is calculated for both an implied one and implied zero and then selected once the implied bit is determined. This partial product gates into a delayed partial product in the counter tree. The multiplicand, X is corrected by subtracting a term $lzc1$ from the counter tree [10] as shown below (W_j are the booth scans):

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

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

The addend can be corrected prior to being aligned into the counter tree. The only difficulty is in correcting the alignment for a denormalized input operand. To do this the alignment is calculated for the exponent difference, D, and $D+1$, and $D-1$, and is selected late based on which operands are denormalized. Thus, denormalized input can be handled without stalling the pipeline or trapping to software. There is one rare case that does trap to software which will be discussed in the next section.

5. Alignment Limitations

The dataflow width is limited to an addend of 56 bits plus 4 guard bits and a product field which is aligned with the adder of 112 bits and 4 guard bits for a total of 176 bit wide dataflow. There are certain cases of unnormalized and denormalized numbers which are difficult to handle in this dataflow. To understand the cases better a case by case study is shown detailing whether a case can be handled by the hardware directly or whether some type of intervention is needed.

In regards to the alignment of the addend with the product, the radix point of the product is fixed in the dataflow. The radix point of the addend is right shifted to achieve the proper fraction alignment prior to the addition of the two.

The BFP arithmetic on this dataflow has to consider the larger dataflow width required by HFP. The dataflow is partitioned as follows for BFP data:

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

There are two possible radix points, Radix1 and Radix2, which are used as reference points for the possible right shifts that may be required. Radix1 is the radix point for the addend data. Radix2 is the radix point for the product data. The G bits represent extra guard bits for addend and product fields.

Case 1: Normalized Addend and Normalized Product: This is the most straight forward case. To calculate the right shift amount SA the following equations are used to achieve the correct result S (D is the exponent difference):

$$
\begin{aligned}
S &= [(1.f_A x 2^{E_A - Bias}) x (1.f_B x 2^{E_B - Bias})] + \\
&\quad (1.f_C x 2^{E_C - Bias}) \\
S &= (1.f_A x 1.f_B) x 2^{(E_A + E_B - Bias) - Bias} + \\
&\quad (1.f_C x 2^{E_C - Bias}) \\
D &= (E_A + E_B - Bias) - E_C \\
SA &= E_A + E_B - E_C + K, \\
&\quad where\ \ Constant\ K = 59 + 2 - Bias
\end{aligned}
$$

If the calculation of the shift amount yields a negative result then the addend C, is not shifted at all and any carry out of the adder is not allowed to propagate since the guard bits, G, are zero. A shift amount which is greater than $54 + 106 = 160$ will result in the sticky calculation being an effective OR of the addend C. The information will be used later in the Rounder.

Case 2: Normalized Addend and Denormalized Product: This case doesn't pose any extra difficulty. If A and/or B are denormalized then there is the possibility that the product may be denormalized. As long as C is normalized, then there are enough bits of precision maintained to form a result consistent with the BFP Architecture since the exponent of the denormalized product will be less than the normalized addend.

Case 3a: Denormalized Addend and Normalized Product: In this case, C is denormalized and P is normalized. The exponent of the resulting sum will be equal to that of the product, P, and the dataflow is sufficient.

Case 3b: Denormalized Addend, Denormalized Product and Underflow Trap Disabled: In this case, C is denormalized and P is denormalized. Since the Underflow Trap is disabled, the result will be rounded to a denormalized number or zero in accordance with the rounding mode and the value of the two guard bits.

Case 3c: Denormalized Addend, Denormalized Product and Underflow Trap Enabled: Since the Underflow Trap is enabled, the result will need to be normalized assuming an unbounded exponent range. This will require at least 53 bits of precision. However, it is possible for the values of the addend and product fields to be disjoint since we don't right shift the product field with respect to the addend field. If both the addend and the product are denormalized, then the result is denormalized and the value of Radix1 will be will be 2^{-1022}. If the value of Radix2 is less than $2^{-1022-60-2} = 2^{-1084}$, then the dataflow is disjoint and is not sufficient. This case is implemented in low level software called millicode. A pseudo-exception is taken by hardware and the millicode handler executes a series of instructions to calculate the correct result. Since we have an underflow condition and the underflow trap is enabled, this case will end in an architectural exception anyhow and so there is no real performance degradation. This is the only case implemented in millicode.

6. Divide

There is an extra wide, 116 bit divider dataflow, in which the mantissa of the quotient and the remainder is calculated using an SRT algorithm. There are numerous different divide instructions and formats to be supported. Not only does the divider support the 6 floating-point formats, but it also performs integer divides on operands 32, 64, or 128 bits wide, which may be signed or unsigned.

6.1. SRT-Algorithm

SRT is a frequently used method for implementing divide and square root on modern microprocessors. It is named after Sweeney, Robertson and Tocher, who independently proposed the algorithm [13, 14]. SRT is an iterative algorithm that retires one digit of the quotient in every iteration. After each iteration, the new partial remainder is calculated by multiplying the previous partial remainder with the radix of the algorithm and subtracting a multiple of the divisor.

This operation is formally described as follows:

$$P_{i+1} \quad = \quad r * P_i - q_{i+1} * D$$

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

The value for the actual quotient digit q_{i+1} is estimated and therefore the partial remainder P_{i+1} could be negative. This can be compensated for by allowing negative values for q_{i+1} too. So, errors in the actual partial remainder can be corrected in later iterations. The convergence of the algorithm requires the following condition be met:

$$|P_{i+1}| \quad < \quad (q_{max} * D)/(r-1)$$

The visual representation of the above equation can be shown in a so called P-D plot. The ranges of q_{i+1} can be seen in Figure 3.

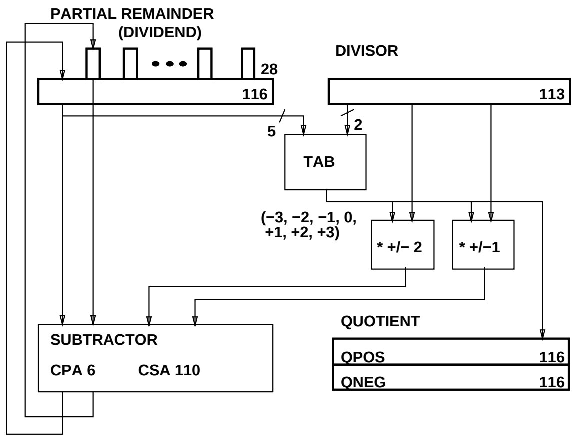


Figure 2. Dataflow structure of divider

Our implementation uses a radix-4 algorithm with a maximally redundant digit set [11, 12]. This reduces the cost of the quotient estimate table lookup at the expense of an increase to the range of quotient digits. Since a full width carry-propagate adder (CPA) would not fit into the required cycle time, a redundant form of the partial remainder is used which allows carry-save adders (CSA) to be used. The increased range of quotient digits has very little effect on a carry-save implementation. The implemented equations are shown by the following:

$$\begin{aligned} P_{s_{i+1}} + P_{c_{i+1}} &= 4(P_{s_i} + P_{c_i}) - q_{i+1} * D \\ q_{i+1} &\in \{-3, -2, -1, 0, +1, +2, +3\} \\ q_{i+1} &= q_{i+1,1} + 2q_{i+2,2} \\ P_{s_{i+1}} + P_{c_{i+1}} &= 4(P_{s_i} + P_{c_i}) - q_{i+1,1} * 1D - q_{i+1,2} * 2D \end{aligned}$$

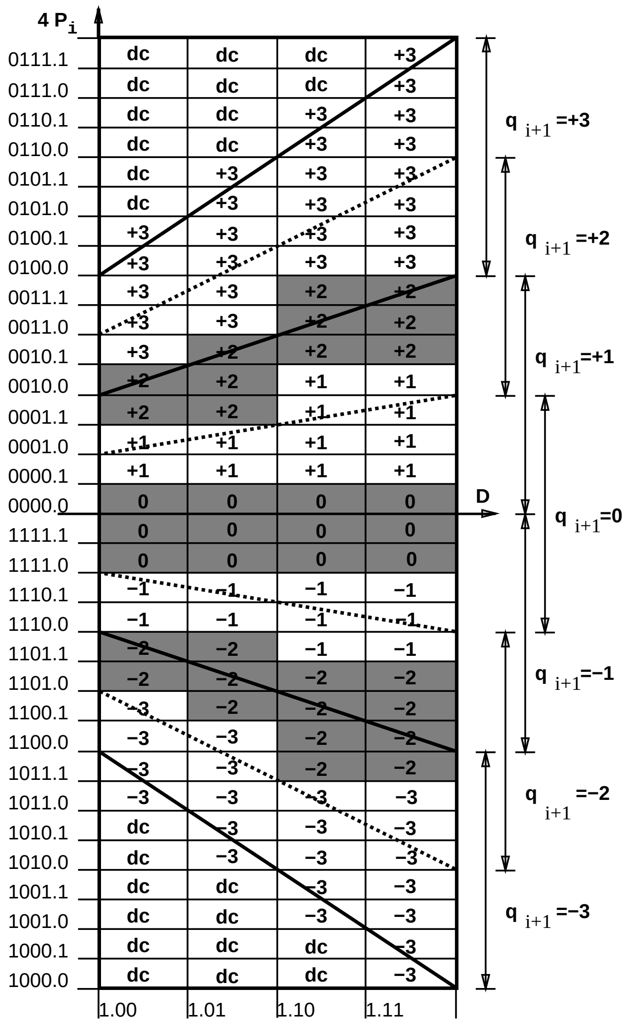
where P_s and P_c are the partial remainder in a sum and carry redundant form, and $q_{i+1,1}$ and $q_{i+1,2}$ are the quotient digit guesses separated into a guess of 1 and a guess of 2 where each can take on the values -1, 0, or +1.

6.2. Structure of Divider Dataflow

Figure 2 shows the 6 main elements of the divide macro, which contains the divide dataflow:

Divisor Register:
This is a simple register with the maximum width of 113 bits for a BFP quad precision, and is where the divisor is stored for the subtractions.

Table Lookup:
This table consists of a relatively small part of combinatorical logic. It needs the five most significant bits of the partial remainder and the two most significant bits of the divisor, after the implied one. Figure 3 is a



$4P_i$	1.00	1.01	1.10	1.11
0111.1	dc	dc	dc	+3
0111.0	dc	dc	dc	+3
0110.1	dc	dc	+3	+3
0110.0	dc	dc	+3	+3
0101.1	dc	+3	+3	+3
0101.0	dc	+3	+3	+3
0100.1	+3	+3	+3	+3
0100.0	+3	+3	+3	+3
0011.1	+3	+3	+2	+2
0011.0	+3	+3	+2	+2
0010.1	+3	+2	+2	+2
0010.0	+2	+2	+1	+1
0001.1	+2	+2	+1	+1
0001.0	+1	+1	+1	+1
0000.1	+1	+1	+1	+1
0000.0	0	0	0	0
1111.1	0	0	0	0
1111.0	0	0	0	0
1110.1	−1	−1	−1	−1
1110.0	−1	−1	−1	−1
1101.1	−2	−2	−1	−1
1101.0	−2	−2	−2	−2
1100.1	−3	−2	−2	−2
1100.0	−3	−3	−2	−2
1011.1	−3	−3	−2	−2
1011.0	−3	−3	−3	−3
1010.1	dc	−3	−3	−3
1010.0	dc	−3	−3	−3
1001.1	dc	dc	−3	−3
1001.0	dc	dc	−3	−3
1000.1	dc	dc	dc	−3
1000.0	dc	dc	dc	−3

Figure 3. Divide Table

combination of a P-D plot and the actual implemented lookup table. It illustrates the shifted partial remainder ranges in which a quotient digit can be selected without violating the bounds on the next partial remainder. It can be seen that the table is asymmetric concerning +/-. This is due to the fact that the partial remainder has a redundant form which causes an additional error. Because of this, the high order bits of the partial remainder can be to small (by one ulp), but never be to large. A symmetrical lookup would also be possible, but then we would need one more bit of the partial remainder or the divisor.

Divisor Multiple Generation:
Before selecting the multiples of one or two, the divisor is inverted depending on the sign bit of the partial remainder. When the partial remainder is negative, we have to add divisor multiples and when it is positive we have to subtract. The $q_{i+1,1}$ term represents divisor multiples of one and and the $q_{i+1,2}$ represents divisor multiples of two. These terms are added together with the partial remainder.

Partial Remainder Register and Subtractor:
The partial remainder register width of 116 bit is defined by the HFP quadword format width (112) plus one hex guard digit. The register consists of a sum part of 116 bits and a carry part of 28 bits. The 6 high order sum bits must be explicit without a corresponding carry because they are used in the table lookup. The most significant carry bit starts at position 6 and only every fourth carry bit is stored. This is possible since the subtractor does not use a full 4:2 reduction, but uses one stage of 3:2 reduction (CSAs) and one stage of CPAs with a width of 4 bits. On the high order side, one CPA with 6 bit width is needed to deliver an explicit value to the table. These 4 bit wide CPAs in the low order range save latches, area and power and do not cost cycle time, since the 6 bit CPA is needed anyhow in the high order range.

Quotient Register:
The quotient register has a width of 116 bits and consists of two parts: the Q_{POS} and the Q_{NEG} registers. When q_{i+1} is positive, it is stored in Q_{POS}, and when q_{i+1} is negative, it is stored in Q_{NEG}. The pointer, which defines which digit of Q is written, is controlled by a counter in the control logic.

6.3. Execution and performance of floating-point divides

After the operands have been loaded into the dividend and divisor registers, the divide iterations can start. When an operand is denormalized, a normalization in the main dataflow is needed in advance. The required number of divide iterations depends on the data format (single, double, quad). After the divider has completed enough iterations, the sum and carry parts of the remainder and the quotient are moved out into the main dataflow before the main adder. There, they are added to get the explicit value of remainder and quotient. Afterwards the quotient is normalized and rounded, using the main dataflow of the floating-point unit.

The following table shows the required cycles for execution of IEEE floating-point divide instructions:

Action \ format	single	double	quad
Load Operands	3	3	15
Divide Loops	14	28	58
Readout Remainder/Quotient	4	4	4
Calculate Quotient	1	1	1
Normalize	1	1	1
Round	1	1	1
Write back	1	1	2
Total Latency	25	39	82
Pipelined Latency	30	34	77

6.4. Execution and performance for integer divides

The integer operands are made positive and normalized in the main dataflow. Afterwards the operands are loaded into the dividend and divisor registers, as for floating-point operands. The difference in handling is the pointer, to which q_i within the quotient registers is written.

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

$$\begin{aligned} n_{Q0} &= n_V - n_D && for\ V_{norm} < D_{norm} \\ n_{Q1} &= n_V - n_D + 1 && for\ V_{norm} \geq D_{norm} \end{aligned}$$

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

$$P_{Start} = 64 - n_{QE}; \quad P_{Stop} = 64$$

The following table shows the required cycles for execution of IEEE floating-point divide instructions:

Action	cycles
Load and concatenate	4
Normalize	5
Divide Loops	1-32
Readout Remainder/Quotient	5
Invert Sign (potential)	5
Write back	5
Total Latency	30 - 61
Pipelined Latency	25 - 56

For integer divides the number of divide iterations depend purely on the effective number of quotient bits. Additionally, there are some base cycles which have a dependency on the operand width too. In classical benchmarks, it often occurs that, the divide result has a small number of effective bits; this improves the performance of integer divides considerably.

7. Physical Implementation

The fraction dataflow has been implemented in a bit stack approach. The A,B and C registers have a width of 56 bits. This is widened during alignment and multiplication. The adder, normalizer, and rounder are 116 bits wide. The output of the rounder is reformatted to a width of 64 (with exponent). The layout has a folded form. On the top of Figure 4 are the architectural floating-point registers with A, B, and C registers below. On the bottom is the normalizer. The exponent dataflow is in a stack on the right of the A, B, and C fraction registers.

The divider is also implemented in a stack approach, whereby the divide-table is combinatorial logic which occupies a very small area on the left hand side of the divider macro. Since the interconnection of the divide engine to the main fraction dataflow is not timing critical, this can be located away from the main dataflow and is shown in the right upper corner of the layout. The fraction dataflow is on the left hand side. On the right are the synthesized control logic macros. For each execution pipeline there is one separate control macro. The macros on the bottom contain some miscellaneous logic, which is not related to the floating-point function.

The divider macro is completely designed in standard inverting CMOS logic. Although it has been implemented as a full custom macro, extensive use of a standard cell library has been made in order to keep layout effort small. As a power saving feature, most parts of the floating-point unit can be turned off completely when not in use. For enhanced testability, each of the master-slave latches is accompanied by an additional scan latch. Adding this extra scan latch to the scan chain configuration results in an increased transition fault coverage. The floating-point unit occupies an area of 3.76 mm^2. The divider macro occupies 0.22 mm^2, which is about 6 % of the FPU. It has been fabricated in IBM's 0.13 micron CMOS SOI technology. At a supply voltage of 1.15V and a temperature of 50^o C it supports a clock frequency significantly greater than 1 GHz.

8. Summary

A new zSeries floating-point unit has been shown which, for the first time, is based on a fused multiply-

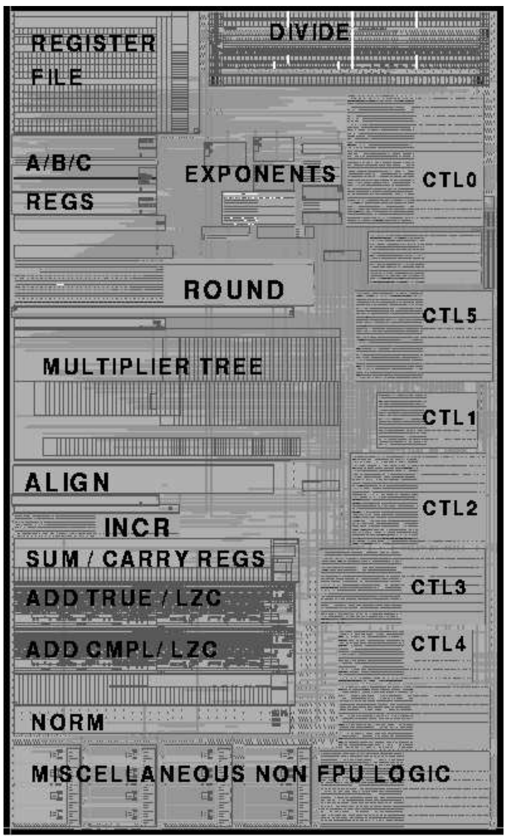


Figure 4. Layout of Floating-Point-Unit

add dataflow capable of supporting two architectures. Both binary and hexadecimal floating-point instructions are supported for a total of 6 formats. The floating-point unit is capable of performing a multiply-add instruction for hexadecimal or binary every cycle with a latency of 5 cycles. This has been accomplished by a unique method of representing the two architectures with two internal formats with their own biases. This has eliminated format conversion cycles and has optimized the width of the dataflow. Though, this method creates complications in the alignment of the addend and product which have been shown in detail. Denormalized numbers are almost exclusively handled in hardware except for one case which is destined for an underflow exception handler anyway.

Also, a fast divide algorithm is used which is capable of supporting a quad precision width and achieves 2 result bits per cycle. The number of divide iterations depends on the effective number of quotient bits, which improves the performance significantly. For the redundant expression of the partial remainder only each fourth carry bit is used, which saves around 80 latches compared to a conventional carry-save approach.

The new zSeries floating-point unit is optimized for both hexadecimal and binary floating-point architecture. It is versatile supporting 6 formats, and it is fast supporting a multiply-add per cycle.

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] "Enterprise Systems Architecture/390 Principles of Operation," Order No. SA22-7201-7, available through IBM branch offices, July 2001.

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

[4] G. Gerwig and M. Kroener. "Floating-Point-Unit in standard cell design with 116 bit wide dataflow," In *Proc. of Fourteenth Symp. on Comput. Arith.*, pages 266–273, Adelaide, Austraila, April 1999.

[5] E. M. Schwarz, L. Sigal, and T. McPherson. "CMOS floating point unit for the S/390 parallel enterpise server G4," *IBM Journal of Research and Development*, 41(4/5):475–488, July/Sept. 1997.

[6] E. M. Schwarz, B. Averill, and L. Sigal. "A radix-8 CMOS S/390 multiplier," In *in Proc. of Thirteenth Symp. on Comput. Arith.*, pages 2–9, Asilomar, CA, July 1997.

[7] 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.

[8] E. Schwarz, R. Smith, and C. Krygowski. "The S/390 G5 floating point unit supporting hex and binary architectures," In *Proc. of Fourteenth Symp. on Comput. Arith.*, pages 258–265, Adelaide, Austraila, April 1999.

[9] E. M. Schwarz, M. A. Check, C. Shum, T. Koehler, S. Swaney, J. MacDougall, and C. A. Krygowski. "The microarchitecture of the IBM eServer z900 processor," *IBM Journal of Research and Development*, 46(4/5):381–396, July/Sept. 2002.

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

[11] M. D. Ercegovac and T. Lang. "Division and Square Root: digit-recurrence algorithms and implementations," *Kluwer*, Boston, 1994.

[12] D. I. Harris, S. F. Obermann, and M. A. Horowitz. "SRT Division Architectures and Implementations," In *Proc. of Thirteenth Symp. on Comput. Arith.*, pages 18–25, Asilomar, California, July 1997.

[13] K. D. Tocher. "Techniques of multiplication and division for automatic binary computers," Quarterly J. Mech. Appl. Math., vol.11, pt.3, pp.364-384, 1958.

[14] J. E. Robertson "A new class of digital division methods," IRE Transactions on Electronic Computers, vol. EC-7, pp.218-222, Sept. 1958.