P6 Binary Floating-Point Unit

Son Dao Trong, Martin Schmookler, Eric. M. Schwarz,
Michael Kroener
IBM Server Division
daotrong@de.ibm.com, martins@us.ibm.com, eschwarz@us.ibm.com,
mkroener@de.ibm.com

Abstract

The floating point unit of the next generation PowerPC is detailed. It has been tested at over 5 GHz. The design supports an extremely aggressive cycle time of 13 FO4 using a technology independent measure. For most dependent instructions, its fused multiply-add dataflow has only 6 effective pipeline stages. This is nearly equivalent to its predecessor, the Power 5, even though its technology independent frequency has increased over 70%. Overall the frequency has improved over 100%. It achieves this high performance through aggressive feedback paths, circuit design and layout. The pipeline has 7 stages but data may be fed back to dependent operations prior to rounding and complete normalization. Division and square root algorithms are also described which take advantage of high-precision linear approximation hardware for obtaining a reciprocal or reciprocal square root approximation.

Keywords: Floating-point unit, denormal result handling, aggressive data forwarding, high-frequency design, data processing without stalls.

1. Architecture and Design goals

The P6 is the next generation PowerPC processor, and is a completely new design from its predecessor, the Power5. Its design is motivated by a need to provide a quantum improvement in performance and reliability for transaction processing type applications. It has been tested at over 5 GHz, which is a much higher frequency than technology scaling alone would provide. Mainframe class robustness is achieved with enhanced recovery capability, parity checking for arrays and data movement, and residue checking for arithmetic operations. A vector execution unit (VMX) based on the Alti Vec architecture [1], and a decimal floating point unit (DFU) based on the new proposed IEEE standard [2] provide new functional capability. The VMX unit has its own architected register file, but the DFU shares the Floating Point Registers (FPRs) with the Binary Floating Point Unit (BFU). However, the Floating-Point Status and Control Register (FPSCR) has been enlarged to allow a separate rounding mode for DFU instructions, and to make room for possible future expansion of other status and control functions. The P6 retains all of the architecture enhancements introduced in the Power5 [3], which had been built upon the design of the Power4. Its most important innovation was to add dual-threading to each core, thus improving the utilization of the various execution units. To reduce power, it also aggressively gated off clocks to various facilities when they were not in use. For the BFU, Power5 added new pipelineable instructions for rounding to a Floating Point integer, corresponding to the Floor, Ceiling, Truncate and Round to nearest functions. It also introduced a new version of non-IEEE mode for divide and square root instructions which reduces their latencies by 6 cycles for applications that do not require IEEE rounding.

To obtain a high frequency on Power6, the cycle time was reduced from 23 FO4 to 13 FO4. This required longer pipelines and simpler instruction scheduling and issueing. Renaming of registers was eliminated for all execution units. Therefore, the FPRs consist only of the 64 registers needed for the two threads, and out-of-order execution within each thread is limited. The BFU execution pipeline is also increased by one cycle to seven, and the complexity of some stages is further increased to prevent needing even more stages. In most cases, however, the effective pipeline length for dependent instructions is still 6 stages, the same as for Power4 and Power5.

The previous designs also allowed the BFU pipeline to stall when any operand was denormal, thus allowing them to first be prenormalized [4]. However, with the shorter cycle time and with wire delays between subunits

approaching a full cycle, this becomes impractical. Several subsequent instructions would be issued before the operands could be inspected and the scheduler signaled and halted. Thus, P6 includes new techniques [5] that allow all multiply-add instructions to be pipelined without stalls. The previous designs also stalled in a late pipeline stage for underflow, overflow, and for some infrequent cases of massive cancellation, but these stalls also are eliminated.

It was also decided that integer multiply and divide instructions, normally executed in the fixed point unit (FXU), should be executed in the BFU to take advantage of its large pipelineable multiplier. The BFU multiplier is modified to accomodate 64-bit integers. The area increase in the BFU is offset by savings in the FXU. For clusters of integer multiplications, it allows these instructions, which are multiple cycle non-pipelined instructions in Power4 and Power5, to be processed every 2-cycles.

There are also some small changes that were made to the custom designed dataflow that allow reuse of large parts of it in a future zSeries processor that includes hexadecimal as well as IEEE binary instructions.

Like the Power4 and Power5, there are two BFUs and each can process two threads simultaneously. Each BFU has its own copy of the FPRs to allow faster operand read. Operands can also be read from the two load store units or forwarded from their load target buffers, thus bypassing the FPR update/read cycles. Within each BFU, results can be bypassed to the operand registers from the normalizer in stage 6 and from the rounder in stage 7.

The FPRs are each 73 bits wide with 64-bit data, 8-bit parity and one for the implied bit. When the implied bit is zero, the exponent Least Significant Bit (LSB) is changed to 1 to provide a consistent bias for calculating the alignment shift. The implied bit is determined for all instruction results and for operands loaded from the data cache.

The ultimate challenge was to achieve a 13 FO4 design while maintaining an effective 6 stage pipeline for most dependent instructions which is similar to Power5 which has a 23 FO4 cycle time. This was accomplished despite additional requirements to provide a much larger multiplier to support the 64-bit integer multiply instructions. Techniques in circuit design, logic design and integration help reduce the cycle time, a few of which are detailed in this paper. Included is a new method for forwarding an unfinished result to dependent instructions from the normalizer. This intermediate result is obtained prior to rounding and also prior to the last normalization step.

In the following the dataflow is first described. Then the interesting specific areas of the design are handled more in details: These are shift amount calculation, multiplier, normalizer and rounder. Section 7 is dedicated to the feedback of unfinished results due to its multiple special cases. Divide and square root implementation is described in section 8. Section 9 gives insight into issue stalling and clock gating and some technology/chip data.

2. Dataflow Overview

The BFU of the Power6 processor has a 7 cycle fused multiply-add pipeline. Dependent instructions can be started after 6 cycles except for a few special forwarding cases where issueing of the dependent instruction is stalled. Figure 1 shows the dataflow with the seven register stages.

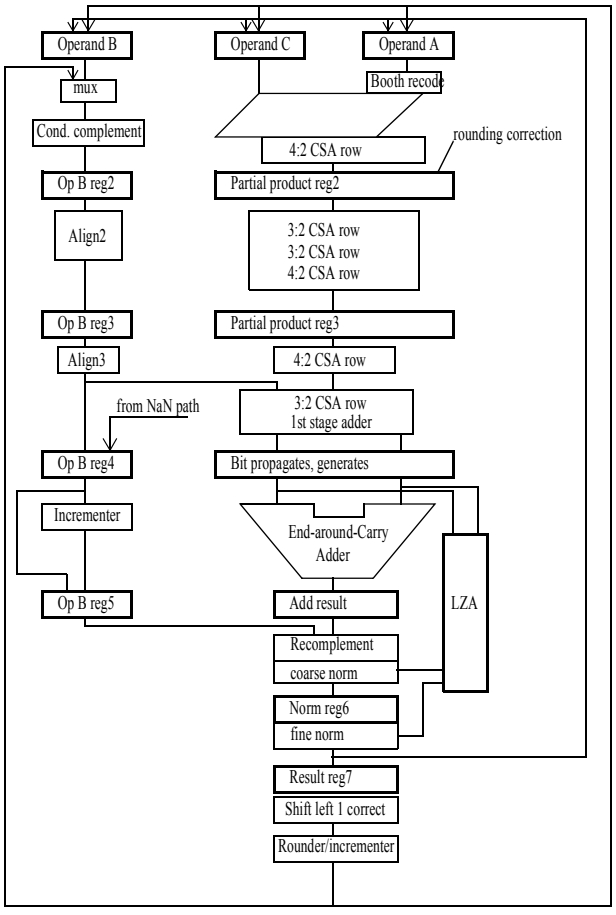

Figure 1: P6 Significand Dataflow

The radix-4 multiplier is designed for 64x64 bit multiplication to support a full fixed-point multiplication as well as IEEE double precision floating-point multiply-

add. Altogether there are 33 partial product terms and one additional rounding correction term that need to be summed up. This rounding term is needed to account for a late increment of the multiplier operands since the result of the foregoing instruction is fed back to the multiplier before rounding. The partial product term reduction is spread out over three cycles, where at the end of the third cycle the aligned addend is merged in the last 3:2 counter row. Alignment shift amount calculation is started in the first cycle without decoding the implied bits, a fine grain shift is done in the second cycle, and the final shift is completed in the third cycle. At the end of the third cycle an out-of-range detection may force the aligner output to all zeros or all ones. In this case the unchanged addend is multiplexed into Breg3. For NaN inputs a special NaN path is pipelined down to be multiplexed into Breg4 when needed. A 120-bit end-around-carry adder [6,7] is used to sum up the partial sums from the multiplier array and the aligned addend. In parallel with the adder is a Leading Zero Anticipator (LZA). Both are spread over latch boundaries, starting from the end of third cycle and finishing in the beginning of the fifth cycle. The output carry from the adder selects either the incremented or non-incremented upper bits of the addend into Breg5. In cycle 5 conditional complementation of the sum is applied to the full width significand and then normalization is started. Normalization is spread over cycles 5 and 6, and rounding is spread across cycles 6 and 7. Excessive wire delays on the feedback paths to the input operand registers make it necessary to forward unfinished results, e.g. unrounded and partially normalized. Data can be forwarded to any of the three inputs. Correction for the multiplier and multiplicand is done with the help of an additional correction term in the multiplier array. For the addend, the exponent is fed back prior to rounding in cycle 6 to the input operand Breg1 and the unrounded significand is discarded, and instead the correctly rounded significand in cycle 7 is multiplexed into Breg2. The exponent doesn't need to be corrected since the dataflow has two vacant bits on the left and a carry out of the rounder is contained in the dataflow of the significand.

In the following section some interesting implementation details are described.

3. Alignment shift amount calculation

Alignment of the basic multiply-add function (f=A*C + B) is calculated by comparing the exponent of the addend B and the exponent of the product A*C to yield a difference which is used as the shift amount of addend B. Only right shift is done during alignment, therefore alignment of addend B is started from left of the product to simulate a left shift of B. Thus shift amount calculation has an offset value to reflect this. When B is much greater than the product AC, it can be placed unchanged logically left of AC with a small gap, usually two bits, between them. In this implementation the gap is five bits for commonality with hexadecimal format. Figure 2 shows the unshifted addend in relation to the product. Note that the product range is 128 bits which is needed for fixed-point multiply which is also executed in the BFU.

Equation (1) shows the shift-amount calculation for binary double-precision (DP):

```
SA0= ExpA + ExpC - ExpB - x'0FFF' + offset    (1)
    SA0: alignment shift amount
    ExpA: exponent of A operand
    ExpC: exponent of C operand
    ExpB: exponent of B operand
    x'0FFF': exponent bias in 13 bit
             internal representation
    offset= 59
```

With the shift amount (SA0) defined as above, the addend B is much greater than the product (BmgtP=1) whenever SA0 is negative (see equation (2)). B can then be placed unchanged and unshifted to the left of the product as shown in Figure 2.

```
    BmgtP = 1 if (SA0 < 0)                    (2)
```

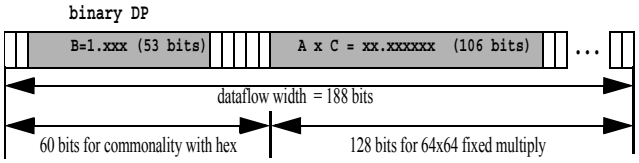

Figure 2: Significand Alignment

Special case requiring additional alignment circuits.

For multiply-add instructions, a difficult case is when the addend is denormal and the underflow trap is enabled. The delivered result must be fully normalized. In past designs, the addend is usually normalized before executing this operation, but that requires stalling the issue of subsequent instructions. There are three variations of this case, differing in how the addend aligns with respect to the product. All can be handled without normalizing any operands and without stalling execution, as recently described [5,6], but they require having a leading zero count for the denormal addend fraction.

One of the cases also requires that the adder output be forced to either zeroes for addition or to ones for subtraction. Another variation ~~also~~ requires the aligner to shift the addend to the left with respect to the placement shown in Figure 2. For this case, the dataflow does not need to be extended on the left as only leading zeroes would be lost. To simplify the decoding of the shift amount for the aligner's multiplexor select signals, the reference point for the shift is moved to the left so that all alignment shifts appear as right shifts, corresponding to positive shift amounts. This change of reference corresponds to adding another offset to the calculation. The equation for the shift amount (SA) then becomes:

```
SA= ExpA + ExpC - ExpB - x'0FFF' + 59 + osl    (3)
  SA: alignment shift amount
  osl: offset for shift left > 52
```

Note that an effective shift of zero corresponds to SA=osl, and a positive value SA<osl corresponds to a left shift of the addend. An effective left shift is restricted to cases where no significant bits of the addend are shifted past the left edge of the dataflow. The condition is determined from the following:

```
BmgtP = 1 if ((SA + lzdb) < osl )              (4)
where lzdb=leading zero count of B
```

When BmgtP=1, the unchanged and unshifted addend is bypassed by the aligner directly to Breg3 and then to Breg4.

Timing consideration:

Given the arbitrary value for osl in equation (3) (it should be greater than 52), it is chosen to be 68 such that the lower 7 bits of the constant value are all zeros. The timing critical lower 7 bits of the shift amount calculation can now be done with a three-port adder. The resulting constant value of -x'0F80' is replaced by its twos complement + x'1080' for addition. Equations (3) and (4) are then:

```
SA= ExpA + ExpC - ExpB + x'1080'               (5)
BmgtP = 1 if ((SA + lzdb) < 68 )               (6)
```

In the first cycle the three-port addition for the least significant two bits is done in random logic. To avoid buffer delays, multiple copies of this logic are used to control the various multiplexors. In the second cycle, three stages of multiplexors allow all shifts up to 63 bit positions to the right, and the total shift amount calculation is completed and decoded. In the third cycle, shifts in multiples of 64 bit positions are done, and all out of range shifts are also handled. The shift amount calculation and the aligner stages are tuned so that alignment of up to 176 bit positions can be achieved in slightly more than two cycles.

4. Multiplier

Traditional Booth encoding is used to determine the partial product terms. For a 64x64 bit multiplication 33 product terms are needed. Because the operands have their implied bit already decoded before getting into the multiplier there is no leading one/zero correction term needed as in other implementations [8, 9]. Instead the FPRs have one extra bit to save the implied bit, and for operands coming from memory the implied bit is decoded in the load/store datapath in time to get latched into the operand register.

One extra term called rounding correction term is added to allow late rounding correction and late shift left one correction (see chapter 8 for more details). At the final stage in the third cycle the aligned addend is reduced with the remaining two partial products to form the two final sum terms input to the carry propagate adder (see Figure 1).

Integration of the multiplier.

Here 4:2 and 3:2 carry-save adders are mixed for a well-balanced pipeline through 3 cycles. Early in high level design the multiplier array was partitioned into macros according to cycle boundaries. The first macro includes the Booth encoding and a first 4:2 reduction stage. However it turned out that the number of wires needed to get out of the first macro is too high and the first macro area is too big. Therefore a new partition was defined. The reduction of the upper 17 and the lower 16 partial products and the rounding correction term are done in two separate macros across the first two cycles. Then a third macro is used for the final summation in the third cycle. Physically, the three macros are placed one below the other, with the third macro below the other two.

5. Normalizer

One of the design goals was to avoid all pipeline stalls, even for infrequent cases of massive cancellation. Therefore, the current design must be able to normalize the full 176-bit intermediate result. The first coarse normalization of 60 bits can be determined before performing the multiply-add. The aligner shift amount and the addend leading zero count (lzdb) are used to determine when the intermediate result is completely contained in the product range or if there are significant bits on the upper addend part, corresponding to B>AC. For this latter case, the

aligner shift amount is added to the lzdb. This is decremented by one for addition but not for subtraction, to allow for loss of a leading zero when the upper bits are incremented. This value is referred to as the high-LZA, and is used for normalizing the intermediate result when the addend is greater than the product.

When the intermediate result is contained in the product range (plus an additional carry out bit), an LZA (Leading Zero Anticipator [10,11]) is used (referred to as the low-LZA). The design selected for P6 is described in [11] and produces a single string for both leading zeroes or leading ones, corresponding to a positive or negative result respectively. A precise LZA as used in [8] or LZA correction as used in [12] or elswhere was rejected due to area and complexity, and for timing reasons for cases where signals from the adder outputs or carries are used. Furthermore, extra circuitry would also be needed for the upper addend part since incrementing may change the number of leading zeroes. Figure 3 shows the structure of the normalizer spread over cycles 5 and 6.

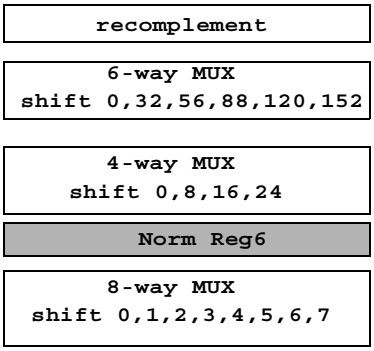

Figure 3: Normalizer structure

Recomplement is done before normalization to give time for the multiplexor select signals to be buffered up. Then coarse normalization of 32 bits is done with integrating selection of either high- or low-LZA. This selection is decoded using the alignment shift amount calculation: when all bits of the addend are shifted into the product range, the high-sum is all zeros and the low-LZA value is used. Since this selection can be determined before performing the multiply-add no timing issue exists. Note that the third normalization shift starts at position 56 (vs. 59) which is 4 bits (vs. 1 bit) to the left of the product. This is for commonality with hexadecimal format instructions where all shifts must be multiples of 4-bit digits.

The normalizer is a very timing critical path. Careful partitioning between cycles, buffering of long wires, routing and placement is needed. To help reduce delay in cycle 6, the one bit left shift needed to correct for the possible error in both LZAs is deferred to the rounder in cycle 7. The result that is bypassed from stage 6 to a dependent instruction is thus unrounded and possibly unnormalized by one bit. However, the corresponding exponent would also be larger by one to compensate, so that the value would be correct.

Generating denormal result.

To generate a denormal result, normalization should be limited to not decrementing the result exponent to a negative value. For the case where B>AC, the exponent value of the intermediate result less one can be directly used for normalization. Compare logic is needed to select between the calculated normalization shift amount normSA (=SA+lzdb for add, SA+lzdb-1 for sub) and the shift amount which would make the resulting exponent equal to one.

For the case where the adder LZA is used, the exponent value of the intermediate result is used to form a bit string with a one in the bit position corresponding to the minimum allowed exponent. This bit string is ORed with the string created from the adder inputs. The resulting bit string is used in the LZA circuit and automatically blocks a shift beyond the limits of normalization for a correct denormal result. No further action is required.

6. Rounder

Figure 4 shows the structure of the rounder. The incrementing of the fraction is started in cycle 6 and completed in cycle 7. The possible left one shift to complete normalization is done simultaneously in the same multiplexor, along with selection of special values (Max number, Infinity) before being sent to the FPR for update.

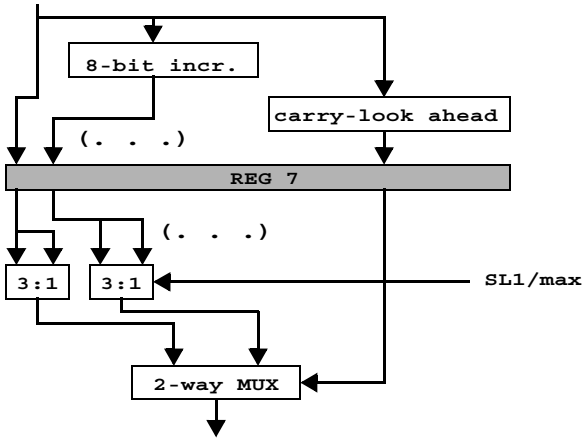

Figure 4: Rounder macro.

The intermediate fraction is first incremented in groups of 8 bits and at the same time a carry-look ahead generates the group carry signals. These signals are first latched. In the next cycle there is a 3-to-1 multiplexor to select data for shift left by one, no shift and the max number. This is done prior to the selection of the incremented result to give logic enough time to decode the rounding condition. There are multiple logic decodes to detect the different possible guard and round bit positions depending on the shift-left by one correction. Similarly, different sticky calculations are done. For the incrementer there are four different positions where an increment may be applied, depending on whether the result is single or double precision and whether a shift-left by one is needed. Especially for the exponent, there are many calculations needed to deliver overflow/underflow rebiased values, the incremented or truncated value, etc... Also exception detection is done for all the possibilities. This is also one of the timing critical corners of the design.

7. Feedback of unfinished result

To achieve the aggressive timing goal and 6 cycles for dependent issueing, an unfinished result (not rounded and not completely normalized) is fed back to the input operand registers for a dependent instruction. Hereto the fraction data bus is extended by one bit to the right. This additional bit is masked if there is no shift left by one correction needed. The result can be fed back to any of the three operand registers. Basically the difference between the unfinished result and final result is the addition of 1 or 0 to the significand, its alignment of shifting right or left by one bit position, and the corresponding exponent. The significand dataflow must also be wider to take care of the eventual carry out of the rounder increment.

Shift amount calculation is done normally as for operands that are sourced from the FPR. No special actions are needed.

Feedback to addend.

Only the exponent of the unfinished result is used for shift amount calculation. The correct fraction (after rounding and complete normalization) from the rounder is then fed back in the next cycle to the second stage of B-register after shift amount calculation is done for the lower alignment bits. With the two extra bits on the left and five bits on the right, the fraction can always be aligned correctly to correspond to the intermediate exponent of the previous cycle. When there is a carry out of the rounder, then the implied bit is placed one bit to the left. When the exponent used had not been adjusted for the final left shift of one, then the fraction from the rounder is placed one bit further to the right in opB reg2.

Feedback to the product.

Feedback of an unrounded result to the multiplier was previously done in the RS/6000 Power 1 and 2 implementations [13], and in the PowerPC Power3. The early bypass was only allowed to one operand, and the function A times (C+1) was performed in the multiplier with another product term to add another copy of A to the product of A and C (this additional term is called the rounding correction term since it corrects the multiplication result to account for the round increment of C which was not applied to the multiplier operand C). This idea is extended in P6 by allowing the early result to be fed back to both A and C thus allowing the square of an unrounded result. Since A equals C in this case, the multiplication performs the function $(A+1)*(A+1)=A^2 + 2A + 1$. Here the correction term is 2A +1 which is A shifted left by one bit with the vacant bit on the right set to 1. This correction term must be set differently for the case where the result fed back is unnormalized by one bit. Different settings are also needed for the cases where either one or both operands are single or double precision. Therefore, there are many ports needed on the multiplexor to create the round correction term, corresponding to the many combinations.

Table 1 shows the different cases of feedback and the corresponding bit setting of the rounding correction term for double precision (DP).

Table 1: Setting of rounding term for DP

Case	rounding term

result is fed back to	`bit 1111....111`
	`1111....666`
	`2345....456`

A=res + 1	`0ccccccccc0`
C=res + 1	`0aaaaaaaaa0`
A=(res+1)/2	`00ccccccccc`
C=(res+1)/2	`00aaaaaaaaa`
A=C=res + 1	`aaaaaaaaa10`
A=C=(res + 1)/2	`0aaaaaaaaa1`

One special case needs to be handled (see table 2). When the result is all ones and is fed back to both multi-

plicand and multiplier and a rounding correction has to be made, the final result exceeds the product range. This single case is easily detected and an extra bit is forced correctly into bit position 59. Also the LZA bit string has to indicate a one in a location in the highest bit position to take care of this occurrence. The multiplier array delivers an all zeros result which is correct.

Table 2: Special case of forwarding and squaring.

```
normal multiplication:
   A=  1.xxxx
   C=  1.xxxx
   P= xx.xxxxxxxx
special case:
   A=  1.1111 +1   = 10.0000
   C=  1.1111 +1   = 10.0000
   P=100.00000000 the '1' is out of the
     multiplier range, needs to be forced.
```

8. Divide and Square root implementation

Reciprocal Estimate Tables.

The PPC architecture provides Reciprocal Estimate and Reciprocal Square Root Estimate instructions. These were originally intended for graphics applications, and only provide approximations having 8 and 5 bits of precision, respectively. For applications having many divide and square root operations, it may be better for performance to replace the hardware instructions with in-line software routines, especially when correct IEEE rounding is not required. With such support now being provided by our compilers, it was decided to further improve these routines for P6 with estimate instructions that provide slightly more than 14 bits of precision. Applications that are compiled specifically for P6 can then take advantage of the higher precision. It was also decided that these 14-bit approximations would also be used for implementing the hardware divide and square root instructions. A precision of 14 bits was chosen because any of several algorithms such as Goldschmidt which double the precision each iteration would obtain a good approximation of IEEE double precision after only two such iterations. Also, by using linear approximations, the estimates can be obtained in only three cycles with very small tables and small area. The tables are similar to those in Alti Vec [1], but provide two additional bits of precision. For the reciprocal, the first 6 bits of the operand, after the implied bit, are used to access a 64 word table, each word containing an offset value for the left edge of the linear segment, and a slope. For the reciprocal square root, two similar tables are used, one for even exponents and one for odd exponents. Outputs from one of the three tables are selected and provided to a small multiply-subtract circuit. For each of the 64 segments of each of the three curves, the slope is that of the line which connects the endpoint of that segment, but the offset is chosen to slide that line halfway down to where it would touch the midpoint of the segment, thus reducing the worst-case error by half and effectively obtaining an additional bit of precision.

Several unique features in the design of this approximation unit keep it small and fast. Since the slope is to be multiplied with the next 11 bits of the operand, the table values do not explicitly provide the slope of the segment, but instead they contains the radix-8 Booth decode of the slope. During the first cycle, while the table is accessed, a small adder obtains the 3-times value of those 11 operand bits. By using radix-8 and avoiding the delay of Booth decoding, the multiplier is kept small and its delay is contained in the second cycle. The third cycle is used to add the sums and carries from the multiplier to the offset value.

If the operand is denormal, it must first be normalized in order to access the tables and obtain the correct linear approximation. When used for the hardware divide and square root instructions, extra cycles can be taken for normalization, since those instructions have data-dependent variable latencies. The estimate instructions however must have fixed latency, and therefore two extra pipelineable cycles are needed to allow for denormal operands. The significant bits of the fraction are obtained using the aligner and the lzdb (leading zero count of B). The complement of the lzdb is used as the shift amount in the aligner, rather than an exponent difference. This results in the MSB of the operand always being in the same bit position of opB reg3 after align2 (see Figure 1), regardless of the number of leading zeros. For the estimate instructions, the appropriate bits are sent from that register to the table. For the floating point divide and square root instructions, that path is only used when the operand is denormal. Otherwise, the data for the table is taken earlier directly from the operand B register.

Floating Point Divide and Square Root.

The algorithms use functional iteration and are equivalent to the Goldschmidt algorithms, but take advantage of the multiply-add operations in the dataflow to avoid extra complementing circuits in the feedback path to the multiplier. Also, the data paths are designed to support the 64-bit integer multiply and the 56-bit hex multiply-add instructions. This allows additional bits to be fed back

during intermediate operations, so that accumulated rounding error from these steps would be much less than a quarter ulp. That eliminates need for the Newton-Raphson step that was used in previous PPC models to compensate for rounding errors prior to the rounding test [14]. As with Power5, a non-IEEE mode bit can be set in the FPSCR to eliminate the rounding test and further reduce latencies by 7 cycles.

Despite the use of higher precision estimates, the latencies are very similar to those of Power 5. Although one iteration is eliminated, additional cycles are needed to obtain the higher precision estimates, and floorplanning placement of the estimate unit requires an additional cycle in each direction between that unit and the pipeline operand registers.

Fixed Point Divide.

The algorithms for the Fixed Point Divide instructions also use functional iteration. There are four such instructions which correspond to either 32-bit or 64-bit operands and either signed or unsigned for each. During the intermediate steps, the full 64-bit unsigned multiplier is used. However, the alignment shifter and normalizer also had to be made wider to accommodate the larger operands. The algorithms are adapted from a software algorithm published by P. Markstein [15]. When implemented in hardware, it allows use of some special operations to reduce latency and special cases can be detected and handled more efficiently. For example, if either operand is zero, that case is detected immediately, and the execution ends early. Overflow cases are also detected and end early. Aside from division by zero, they only occur for signed divides where the numerator is the most negative integer and the divisor is -1.

The operands are first converted to floating point format, and then use the table to obtain an estimate of the reciprocal of the divisor. The conversion makes use of the aligner and lzdb for positive and unsigned operands, which is similar to the handling of denormal operands described above for accessing the tables. A leading ones counter was added so that negative integers could be handled essentially the same way. The intermediate quotient is then obtained in a way that is similar to the floating point divide. For 64-bit precision, a Newton-Raphson step is then used, since the dataflow saves no additional bits of precision. However, for 32-bit divides and also for all cases where the numerator has fewer than 50 significant bits, 55 bit accuracy in the intermediate quotient can be obtained without that step, saving 12 cycles of latency. In either case, once a floating point quotient of sufficient accuracy is obtained, we can add a 1 to an appropriate bit position corresponding to a few bits below the LSB of the integer result, and then truncate to the nearest integer value which corresponds to the correct quotient.

For doubleword divides with larger numerators as described above, the latency is 45 cycles, which does not include the time to move the operands and the result between the BFU and the General Purpose Registers (GPRs) of the fixed point unit. For smaller numerators, which is probably the typical case and for all single word divides, the latency is 33 cycles.

Using vacant cycles during divide or square root.

During execution of a divide or square root instruction, most of the slots in the pipeline are unused as each operation waits for the results from previous operations. In P6, to improve performance, other floating point instructions not dependent on the divide or square root result can be issued during those vacant slots. They must be 1-cycle pipelineable instructions such as multiply-add. The instruction scheduler is notified several cycles before each vacant cycle is available. This is similar to the use of unused cycles for independent multiplication instructions in the AMD-K7 [16]. Although this is a conceptually simple idea, it was not implemented in any of the previous processors. In P6, a second temporary register was needed to hold the divide operands so that the pipeline operand registers wou;ld be available for other instructions. This would have provided greater improvement in a processor such as Power5 which used register renaming and out-of-order execution within the floating point unit. In P6, with only limited out-of-order execution allowed, and with two execution units, it provides only a small benefit when just one thread is active. However, when two threads are active, there is no limit to the use of unused cycles for pipelineable instructions from the other thread.

9. Miscellaneous

Special cases where forwarding is stalled.

There are several cases where bypassing results from stages 6 or 7 to dependent instructions must be prevented. It has been shown that a normal result can be forwarded from stage 6 to the multiplier, and if that result is subsequently rounded up, then the rounding correction multiplexor in the multiplier can be used to compensate for the unrounded operand passed to the multiplier. However, the rounder in stage 7 also produces the special results for

underflow and overflow. For these cases, we must avoid forwarding from stage 6. There are also cases where an invalid operation may occur. For the pipelineable arithmetic instructions, these cases only occur when an operand is infinity or a SNaN. If the Invalid Operation trap is enabled, then writeback to the FPR must be prevented, and forwarding must be prevented from both stage 6 and stage 7. In PowerPC architecture, traps are not required to take place at the instruction that causes the trap when in imprecise exception mode. Subsequent dependent instructions may be executed, but they must use the FPR data as it was before the invalid operation occurred. Thus, we have cases where forwarding must be stalled from stage 6, and other cases where it must be prevented from both stages.

The issue unit which schedules instructions, must be signaled very early when forwarding must be stalled or prevented. To prevent bypassing from stage 6, the signal must go out from stage 2, and the condition must be detected in stage 1. At this time, the result exponent has not been determined. There is only time for the most significant bits exponents to be examined, and if there is any possibility at all that either an underflow, an overflow, or an invalid operation may occur, then a signal goes out to block bypass from stage 6. For the multiplier operands, only the three most significant bits of the A and C exponent are used. If the operation has a B operand (the addend), then the signal also goes out if the implied bit is zero or if the biased exponent has all ones or is close to zero.

Forwarding from stage 6 is also blocked for any single precision instruction. Determining possible underflow or overflow for single precision would require examining more exponent bits and would delay the signal further.

The signal for preventing bypass from stage 7 must go out a cycle later. This allows more time for determination , and also fewer cases must be detected. These are mostly cases where an operand has an exponent of all ones.

Checking - Clock gating.

All FPR registers have parity bits, one for each byte. When read from the FPR into the operand registers, the parity bits of the operand registers are computed and compared to the parity bits read from the FPR. Data coming from the load store units also have parity bits and are checked similarly. All data busses in and out of the BFU are parity checked. Internally the full significand dataflow is checked with residue checking logic for all multiply-add operations. Residue three is used for timing reasons.

Power saving is done in fine grain manner, whereby each stage of the pipeline is powered up and shut down in a pipeline with clock gating. The complete BFU is in sleep mode when not in use. Once a BFU instruction is dispatched, the first input stage of the pipeline is powered up then the next pipeline stage and so on. There is one interface macro which is always clocked to listen to the interface signals. This macro also controls the clock gating.

Technology.

Typically the floating point unit is located toward the outside edge of the central processing core. Due to its physical nature of having a large number of functions which are traversed in series, the BFU always has a lengthy rectangular shape. Data processing however starts from the register file (FPR) and ends up back in the FPR. Some earlier implementations used a wrapped dataflow [17]. However, due to the wide dataflow, there would be too many wires that must go down the pipeline and then go back up to the FPR. To reduce wire lengths and reduce the number of wiring tracks through some macros, an "O" shape floorplan was created. The computation flows down the right stack to the adder and then flows up the left stack back to the FPR. The adder is the optimal cross-over point since only the group generates are timing critical and require fast low-resistance horizontal wires. Latch delay overhead is minimized by the extensive use of scannable pulsed latches [17]. The functional data path of each latch is only through a dynamic L2 latch which is transparent while the clock is active. The L2 clock is only active for a few FO4s to avoid race conditions between latches of different pipeline stages, but it allows some balancing of delays across the cycle boundary during that window. Moreover, the total delay through the latch is half of what the setup time and delay would be for a traditional master-slave latch. For testing, these latches also have a mode which uses a parallel master-slave path that is scannable.

The Power6 chip is fabricated in a 65nm SOI process technology. Tests in the lab show the chip running at close to 6 GHz. Power simulation at 1.1V, 4GHz and 100% utilization shows BFU dataflow consumes 310 mW. When not utilized the whole unit is clock gated to achieve zero active power. Unit area is about 2.5 mm^2.

10. Summary

The Power6 Binary Floating-Point unit has been described which surpasses the design of prior fused mul-

tiply-add dataflows. Predecessor designs required 132 FO4 delays for dependent operations and started independent operations every 23 FO4 cycle. This design has a 7 stage pipeline that is effectively only 6 stages for dependent instructions achieving a delay of only 78 FO4s between dependent instructions and can start independent operations every 13 FO4 cycle. The design, using a technology independent measure, is over 40% faster. And with the latest 65nm SOI technology has been tested at over twice the frequency of Power 5 with the same clocks per dependent BFU pipelined instruction. To achieve this major accomplishment required many logic, circuit, latch, and integration techniques. The key microarchitecture change was to forwarded intermediate results prior to rounding and prior to full normalization. Additionally many cases requiring stalls in prior designs were eliminated. These changes added complexity to many parts of the design as shown in this paper.

The POWER6 BFU incorporates many microarchitecture, logic, circuit, latch, and integration techniques to achieve outstanding performance with an effective 6-cycle 13 FO4 pipeline.

References

[1] M.S. Schmookler, M. Putrino, et al, "A Low-power, High-speed Implementation of a PowerPC Microprocessor Vector Extension,"Proc. of Fourteenth Symp. on Comput. Arith., pp. 12-19, April 1999.

[2] The Institute of Electrical and Electronic Engineers, Inc.,``IEEE standard for floating-point arithmetic, ANSI/IEEE Std 754-Revision," http://754r.ucbtest.org/, Oct. 2006.

[3] B. Sinharoy, "IBM Power5 Processor," Proceedings of Microprocessor Forum, Oct. 2003.

[4] E.M. Schwarz, M. Schmookler, S. Dao Trong, “Hardware Implementations of Denormalized Numbers," Proc. of Sixteenth Symp. on Comput. Arith., pp. 70-78, June 2003.

[5] E.M. Schwarz, M. Schmookler, S. Dao Trong, “FPU Implementations with Denormalized Numbers,” IEEE Transactions on Computers, vol.54, no. 7, pp 825-836, July 2005.

[6] E.M. Schwarz, “Binary Floating-Point Unit Design: the fused multiply-add dataflow”, High-Performance Energy-Efficient Microprocessor Design, Edited by V.G. Oklobdzija and R.K. Krishnamurthy, Springer, Chapter 8, 2006.

[7] X.Y. Yu et. al., “A 5GHz+ 128-bit Binary Floating-Point Adder for the Power6 Processor”, Proc. of ESSCIRC, Sept. 2006.

[8] G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess. "High performance floating-point unit with 116 bit wide divider," Proc. of Sixteenth Symp. on Comput. Arith., pp. 87-94, June 2003.

[9] T. Williams. "Method and apparatus for multiplying denomalised binary floating point numbers without additional delay', Journal="U.S. Patent No. 5,347,481 Sep. 13,1994

[10] E. Hokenek and R.K.Montoye. “Leading-Zero Anticipator (lza) in the IBM RISC system/6000 Floating-Point Execution Unit”, IBM Journal of Research and Development, vol. 34, pp. 59-70, 1990.

[11] M.S. Schmookler, K.J. Nowka. “Leading Zero Anticipation and Detection - A Comparison of Methods", Proc. of Fifteenth Symp. on Comput. Arith., pp. 7-12, June 2001.

[12] S. Mueller et al. "The Vector Floating-Point Unit in a Synergistic Processor Element of a Cell Processor", Proc. of Seventeenth Symp. on Comput. Arith., pp. 59-67, June 2005.

[13] R.K. Montoye, E. Hokenek, S.L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," IBM J. Res. Develop. 34, pp. 59-70, Jan. 1990.

[14] R.C. Agarwal, F.G. Gustavson, M.S. Schmookler, "Series Approximation Methods for Divide and Square Root in the Power3 Processor," Proc. of Fourteenth Symp. on Comput. Arith., pp. 116-123, April 1999.

[15] P. Markstein, “IA-64 and Elementary Functions: Speed and Precision,” pp. 129-131, Prentice-Hall, 2000.

[16] S.F.Oberman, “Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor,” Proc. of Fourteenth Symp. on Comput. Arith., pp. 106-115, April 1999.

[17] B. Curran et al., "4GHz+ Low Latency Fixed-Point and Binary Floating-Point Execution Units for the POWER6 Processor”, Proc. of ISSCC, 24.1, Feb. 2006.