The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor

Silvia M. Mueller[1], Christian Jacobi[1], Hwa-Joon Oh[2], Kevin D. Tran[2], Scott R. Cottier[2], Brad W. Michael[2], Hiroo Nishikawa[3], Yonetaro Totsuka[4], Tatsuya Namatame[5], Naoka Yano[5], Takashi Machida[5], Sang H. Dhong[2]

[1] *IBM Boeblingen, Germany.* [2] *IBM Austin, TX.* [3] *IBM Japan Industrial Solution Co.*

[4] *Sony Computer Entertainment of America, Austin, TX.* [5] *Toshiba Corporation, Japan.*

{*smm, cj*}*@de.ibm.com,* {*hjoh, tranke, cottier, bradmich, dhong*}*@us.ibm.com, hiroon@jp.ibm.com, totsuka@ausafs.ihost.com* {*tatsuya.namatame, naoka.yano, takashi.machida*}*@toshiba.co.jp*

Abstract

The floating-point unit in the Synergistic Processor Element of the 1st generation multi-core CELL Processor is described. The FPU supports 4-way SIMD single precision and integer operations and 2-way SIMD double precision operations. The design required a high-frequency, low latency, power and area efficiency with primary application to the multimedia streaming workloads, such as 3D graphics. The FPU has 3 different latencies, optimizing the performance critical single precision FMA operations, which are executed with a 6-cycle latency at an 11FO4 cycle time. The latency includes the global forwarding of the result.

These challenging performance, power, and area goals were achieved through the co-design of architecture and implementation with optimizations at all levels of the design. This paper focuses on the logical and algorithmic aspects of the FPU we developed, to achieve these goals.

1 Introduction

The Synergistic Processor Element (SPE) of a CELL Processor [4] is the first implementation of a new processor architecture designed to accelerate media and data streaming workloads. The SPE is a 32b, 4-way SIMD, high-frequency design with an 11FO4 cycle time. Area and power efficiency are key enablers for the multi-core design of a CELL Processor that takes advantage of the parallelism in the target workloads.

Real-time 3D graphics applications demand a single precision (SP) performance significantly exceeding that of conventional processors and a competitive double precision (DP) performance. The FPU of the SPE is therefore optimized for SP performance; SP multiply-add operations are executed at maximum speed. Double precision operations and converts between integer and floating-point are less performance critical; they can tolerate extra execution cycles. The SPE therefore has a separate 4-way single precision FPU (SPfpu) and a double precision FPU (DPfpu), rather than to support SP inside a DP FPU. The SPfpu also supports 4-way SIMD integer multiply-shift instructions.

Early performance studies showed that real time graphics applications achieved optimal performance with a 6-cycle pipelined SPfpu when the cycle time is approximately 11fo4 [12]. This includes the 6FO4 latency for distributing the result to the register file and all functional units of the SPE. In an 11FO4 design that only leaves 60FO4 for logic and latches. Conventional SP FPUs have a latency of about 100FO4 [17]. The key challenge of the SPfpu is to save 40FO4 and still be power and area efficient. That requires optimizations at all levels of the design; system architecture, micro architecture, logic, circuits, layout and floorplan have to be carefully optimized and co-designed. The SPfpu architecture and implementation are customized to the needs of the target application, also trading infrequently used features for overall performance. For example, with respect to the SP operations, the target applications virtually only use truncation rounding.

While the SPfpu is optimized for very high performance, the DPfpu has different constraints. The major challenge of the DPfpu is a state-of-art design with a very tight area budget. In addition, its interface to the SPE has to be such that it does not penalize the SPfpu performance. Both FPUs are based on a conventional FMA pipeline.

This paper mainly describes architecture and logic optimizations. Physical design optimizations are addressed as far as they have an impact on the architecture. After discussing physical design considerations, we give an

Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)
1063-6889/05 $20.00 © 2005 **IEEE**

overview of the SPE FPU. We then discuss the major optimizations used in the SPfpu and DPfpu and address the power saving concept.

2 Physical Design Considerations

The high performance, power and area efficient design of the SPE FPU strongly hinges on optimizations of the circuit and physical design [15], some of which also have an impact on the FPU architecture and logic design. The two major aspects are described here:

Interface to the SPE To achieve the SPE's 6FO4 global result forwarding, physical considerations, such as wiring and floorplan, have to be forced to the forefront. Register file, forwarding network and all functional units are placed in a bit stack, except for the DPfpu which is to the side of the SPfpu due to area constraints. There are not enough wiring resources to drive ten 128b operand and result busses sidewise to the DPfpu or to allow for a global DPfpu result bus. The DPfpu is therefore connected to the SPfpu; they share a set of operand latches and a result bus.

Latch Types A latch insertion delay of 2 to 3FO4 occupies 20 to 30% of the 11FO4 cycle. In order to minimize this overhead, the CELL Processor uses a special set of latches [16]. Mux latch and pulsed latch are the two major types used in the FPU.

The mux latches integrate a 4, 5 or 6-port mux with the latch function, hiding the mux delay to a large extend. Most functional FPU blocks, like aligner, adder and normalizer, require some mux function. The FPU designs are therefore optimized to end at least every other cycle in a mux latch. The use of mux latches in the alignment and normalization shifter comes natural. However, the logic in the adder, LZA and exponent path need to be re-organized to make good use of wide mux latches (Section 4).

The pulsed latches integrate an AND function and allow to delay the latch point up to 1FO4. By carefully choosing the latch points and adjusting them somewhat through pulsed latches, the FPU achieves a maximum path delay difference of only 3% between pipeline stages. Through the careful use of these two latch types, we were able to improve the latency of the SPfpu by about 10 to 12 fo4.

3 Overview of the SPE FPU

The FPU (Figure 1) consists of four 32b SPfpu cores, a 64b DPfpu core, and a 128-bit wide frontend which provides the operands. The frontend has two sets of operand latches. Each operand latch is a 5- or 6-port MUX latch which selects the most recent copy of the operand among

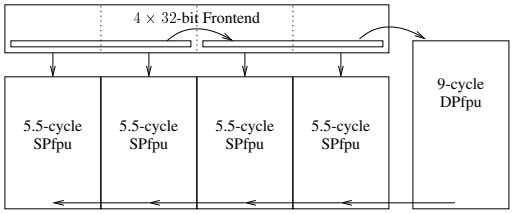

Figure 1. Overview of the SPE FPU

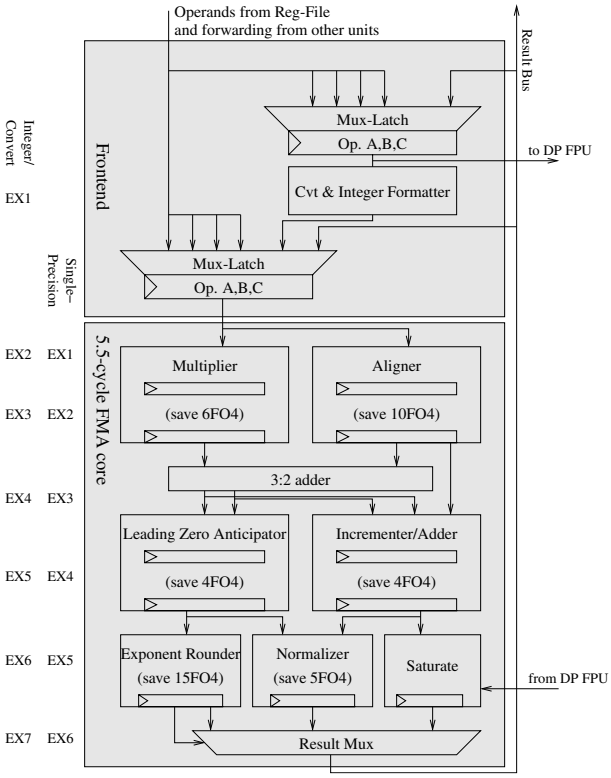

Figure 2. Single Precision FPU

the forwarded results and the data provided by the register file and the forwarding network.

The primary set of operand latches feed the SPfpu cores. The SPfpu is a fully pipelined, fused multiply-add design with 2 pipeline stages for aligner and multiplier, 2 for adder and LZA, and 2 for normalizer, rounder and result forwarding (Figure 2).

The second set of operand latches feeds the DPfpu and a SP formatter stage. The formatter is used for integer multiply instructions, for converts and for the interpolate instruction. It pre-processes the operands such that the SPfpu core can execute these 7-cycle instructions as special multiply-add operations without penalizing the performance critical SP multiply and add type instructions (6-cycle). In parallel to the normalizer, the SAT macro performs the post-processing for the integer multiply instructions and the saturation for the converts. A port of the SPfpu result mux is

used to merge in the integer result.

The DP instructions are 2-way SIMD. Due to tight area constraints, only one 64-bit DPfpu core is instantiated. The FPU breaks the DP instruction in 2 64-bit operations, which the 9-cycle DPfpu processes in a pipelined fashion, i.e., the DPfpu is half-pumped. Since the DPfpu is located to the side of the SPfpu, an extra cycle is required to transfer the operands to the DPfpu and to transfer the result back to the SPfpu. For the sake of a fast SPfpu result mux, DP and integer results share a port; both results get merged in the cycle prior to the result mux. This interface adds a 4-cycle overhead to the latency of DP instructions.

3.1 Single Precision FPU Architecture

The main instruction class of the SPfpu are fused-multiply-add instructions $A * B + C$ and derivatives like $A + B$ and $A \times B$, which have a 6-cycle latency. The SPfpu also implements support-instructions for computing $1/A$ and $1/\sqrt{|A|}$ explained below, conversions between integers and floating point numbers, and various integer multiply-add instructions. Floating point compares are executed in the fixed point unit.

To achieve the high performance needed for the target application and to meet their special architecture requirements, the SPfpu deviates from the IEEE standard [8] in some points: First, denormal operands and results are forced to zero. This speeds up the FPU pipe, as will be explained later. The same kind of saving can for example be obtained by trapping on denormal numbers, as several FPU designs do.

Second, numbers with exponent $e = 255$ are treated as normal numbers in the binade of $2^{255-bias}$; in the IEEE standard these values represent either infinity or Not-a-number (NaN). In media applications, infinity and NaN usually have no real meaning; on the other hand, the extra binade of normal numbers is very useful. Note that this modification has no impact on the latency of the SPfpu. Operations which produce an overflow are saturated to the maximum representable number instead of infinity. The SPfpu sets a special exception flag, when forcing a denomal number to zero or encountering a number in the extended range.

Third, the SPfpu supports only the round-towards-zero rounding mode. This speeds-up the fraction datapath since sticky-bit computations are simpler and no fraction rounding is needed. However, it puts more pressure on the exponent logic which now becomes timing critical, as will be explained in Section 4. In addition, the SPfpu does not support trapping on exceptions, to allow for a simpler and faster SPE control.

The fused-multiply-add instruction in the SPE is defined as $A * B + C$, while the PowerPC architecture [1] defines it as $A * C + B$. This subtle difference can be exploited to reduce the logic depth (Section 4).

The VMX architecture [2] supports estimate instructions for divide and square-root. These instructions are usually composed of a lookup- and interpolate-step. When implemented with reasonable overhead, they tend to have a longer latency than the standard FMA instruction. In the SPE architecture these two steps are defined as separate instructions: there are two estimate instructions for $1/A$ and $1/\sqrt{|A|}$; these return a $base$ and a $slope$ value which are stored together in the fraction field of the result. This result can then be fed into an interpolate instruction which increases the precision of the estimate by linear approximation. The result of the interpolate step can further be refined by means of a Newton-Raphson step (exploiting the FMA instruction). Note that splitting the estimate and interpolate step into two instructions has twofold benefit for the application programmer: (i) the hardware can be better exploited through software pipelining, and (ii) the result of the estimate instruction can be used as a fast approximation if the low-precision is sufficient. This is also the reason why the SPE does not support atomic divide instructions, but leaves this to software. The two estimate instructions are executed in the FXU; the interpolate instruction is executed in the SPfpu.

The last instruction class supported by the SPfpu are signed and unsigned integer-multiply-adds. Various 16×16-bit multiplications are supported, for example $A_{lo} \times B_{lo} + C$, $A_{hi} \times B_{lo}$, or $A_{hi} \times B_{lo} + C$. Software can combine the 16×16-bit multiplications to perform 32×32-bit multiplications. The hardware restriction to 16×16-bit multiplications allows the reuse of the single-precision multiplier without penalizing the floating-point performance.

3.2 Double-Precision FPU Architecture

The DPfpu is IEEE-compliant except for the following aspects: (i) denormal operands are treated as zero (denormal results are computed compliant to the standard), and (ii) NaN operands are not propagated to the result, instead a generic NaN is computed whenever a NaN result occurs. The DPfpu supports all four IEEE rounding modes for the standard set of fused-multiply-add instructions, add/subtract, multiplication, and conversions between single and double precision. Like the SPfpu, the DPfpu also goes for the option in the IEEE standard to only support non-trapping exception handling.

For the rare cases, that an application requires single precision add/subtract or multiplication with a non-truncation rounding mode, the conversions in the DPfpu allow for an efficient emulation using a code sequence of extend-to-double, double precision operation, and round-to-single.

4 Design of the Single Precision FPU

The SPfpu is optimized for SP multiply-add operations with truncation rounding and trap disabled execution. Denormal operands and results are forced to zero. In order to fit the SP multiply-add function in 5.5 11FO4 cycles, each 2-cycle functional block has to be improved by at least 5FO4, some even by 15FO4 (Figure 2).

The latency reductions in multiplier, LZA and normalizer are mainly due to highly customized circuits and physical design. Considerable logic optimizations are applied to frontend, aligner, adder, and exponent rounder, described below.

4.1 Optimizing the Operand Formatter

The formatter pre-processes the operands so that the later pipeline stages can process all instructions in a uniform way as special multiply-add.

The single precision multiply and add type instructions require little formatting: the unpacking of packed floating point data and the selection of the operands in order to support A*B+C as well as A*B and A+B. With the optimizations described below, this formatting can be combined with the aligner logic hiding its delay.

The integer multiply, convert and interpolate instructions require a more extensive operand formatting. This is done in an extra formatting stage, such that these instructions have a 7-cycle latency.

4.1.1 Fast Unpacking for 6-cycle Instructions

The FPU receives the operands in the packed format. The operands get unpacked into a sign bit, an exponent, and a mantissa.

In the SPfpu, denormal inputs and results are forced to zero. That simplifies the unpacking. The SPfpu assumes that the input is a normalized number, setting the integer bit of the mantissa to 1 without inspecting the exponent. In case of a Zero or denormal operand, a late correction is performed forcing special values into the aligner output, using the timing uncritical bypass path:

- For a Zero addend the aligner output is forced to zero or all ones depending on the effective operation.
- For a Zero product the addend is forced into the most significant 25 bits of the aligner output; these bits do not overlap with the product. Adder and normalizer are set to only use the most significant 25b of the intermediate fractions.
- If addend and product are zero, the rounder forces a true zero based on a special select signal.

Table 1. Operand assignment for the single precision multiply and add type instructions in SPE and PowerPC format.

instruction class	SPE	PowerPC
add, subtract	$A*1+B$	$A*1+B$
multiply	$A*B+0$	$A*B+0$
multiply-add	$A*B+C$	$A*C+B$

The exponent check is off the critical path, since it is done in parallel to the aligner.

4.1.2 Operand Order for 6-cycle Instructions

There are two ways to express a multiply-add either as A*B+C or as A*C+B. Unlike the PowerPC, the SPE uses the first format, because it allows to hide the formatting latency for all 6-cycle instructions. Table 1 depicts the operand assignment for both formats. Forcing a zero addend is covered by the late zero correction. For power saving, the multiplier is bypassed on add and subtract. Thus, only multiplies and aligns with non-constant operand have to be considered.

Three paths are impacted by the operand selection: the multiplier inputs, the aligner shift-amount computation, and the aligner fraction path. The first two are equally timing critical; the timing of the third is somewhat relaxed.

With the PowerPC format, the multiplier either computes A*B or A*C and therefore needs a multiplexer on one of its operands. Even for Booth multipliers, both inputs are equally time critical; while one operand gets re-coded, the other operand gets amplified and distributed to all the partial product generation macros. Thus, the mux adds to the overall delay of the multiplier. The aligner receives the fraction of B, and its shift amount equals

$$sha \;=\; ea + ec - eb + K,$$

where K is a design specific constant (a pre-shift correction minus bias); ec is forced to zero in biased format for fadd and fsub. Thus, the aligner needs no operand muxing. The PowerPC format penalizes the multiplier over the aligner.

With the SPE format, the multiplier always computes A*B; no muxing on the multiplier inputs is needed, allowing for a faster multiplier path. The aligner gets either the fraction of C or B. Since this is not the critical path of the aligner, the muxing of the fraction causes no extra delay.

The alignment shift amount now equals

$$sha \;=\; \begin{cases} ea + \text{'0'} - eb + K & \text{for add type} \\ ea + eb - ec + K & \text{for multiply-add type} \end{cases}$$

The shift amount for add operations can also be expressed as $ea + eb - 2eb + K$. The exponent muxing is done in parallel

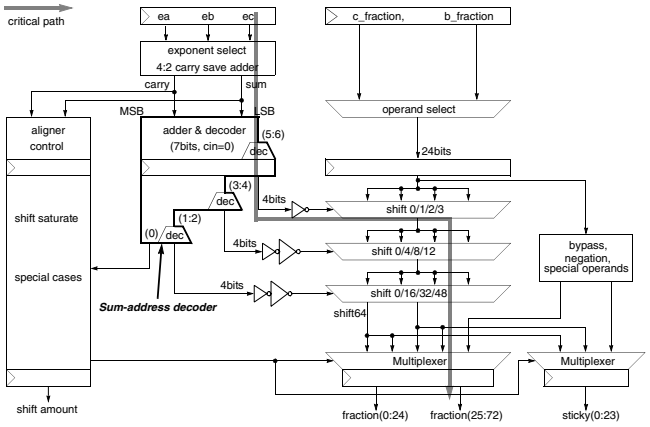

Figure 3. SPfpu Aligner. The timing critical path is marked in grey.

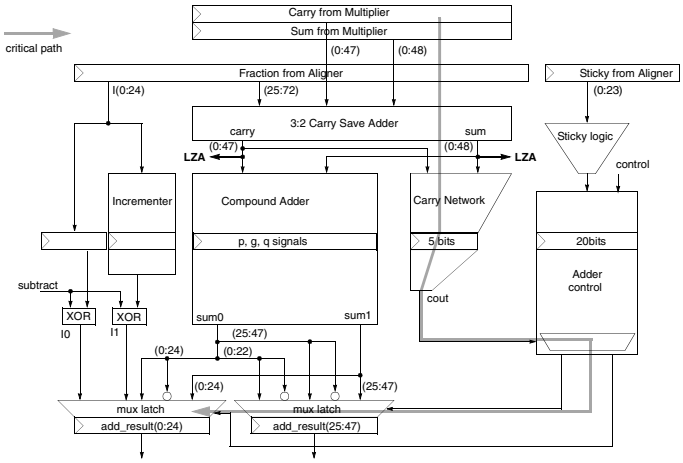

Figure 4. SPfpu Fraction Adder. The timing critical path is marked in grey.

to the 3:2 compression of ea, eb, K, hiding the mux delay. Thus, the SPE operand order allows for a faster multiplier without penalizing the aligner latency.

4.2 Alignment Shifter

The alignment shifter (Figure 3) consists of the exponent selection described before, an adder-decoder block which computes the decoded shift amount, the actual shifter which is partitioned into 4 mux stages, and the bypass logic and control which handles special operands and shift saturation. The last mux stage is integrated in a 6-port mux latch; it performs the wide shift, recomplements the addend in case of an effective subtraction, and merges in the result of the bypass logic.

The shift amount computation is on the critical path of most aligner designs. In a conventional design, exponents ea, eb, ec and constant K get compressed into a carry-save representation (s, t) of the shift amount. A 7-bit adder produces the binary representation sha which gets decoded into hot-1 mux select signals. Our implementation uses a special sum-addressed shifter which removes the 7b adder from the critical path:

1) Vectors s and t are partitioned into 2-bit segments. For each segment s', t', we compute the unary decode sel of $s' + t'$ ignoring the carry-in from previous segments. Each bit position adds up to a value v in $\{0, 1, 2\}$ represented by standard kill, propagate, generate signals. Signals sel are obtained from v by simple AOI functions.

2) In parallel to step 1, a carry network computes the group carries to be added to each of the segments.

3) Finally the group carries are used to correct the select signals of step 1. Since the signals sel are in a decoded form, adding a 1 corresponds to rotating sel left by one bit position using a 2-port mux.

With this optimization and with highly tuned circuits and the fast mux latch, we were able to fit the aligner in 21FO4.

4.3 Fraction Adder

The fraction adder (Figure 4) computes the sum or absolute difference of its inputs using the end-around-carry concept. Let x and y be two numbers, and let eac be the carry-out of $x+!y$, where !y indicates the one's complement of y. The sum or absolute difference r of x, y can then be expressed as

$$r = \begin{cases} x + y & \text{for add} \\ x+!y + 1 & \text{for sub with eac=1} \\ !(x+!y) & \text{for sub with eac=0} \end{cases}$$

A 3:2 adder compresses the main part of the aligned addend and the two partial products. The intermediate results are passed to a carry-look-ahead compound adder which computes sum (sum0) and sum+1 (sum1). The alignment, the operation, the sticky bit, and the carry-out of the adder determine whether sum0 or sum1 is chosen, and whether a recomplement of the result is needed.

The msb bits I of the addend are passed to an incrementer which computes I and I+1, and recomplements both results on an effective subtraction. The carry-out of the adder selects between these two results I0 and I1. The msb bits I only matter when the exponent of the addend is larger than the exponent of the product.

The selection of sum0, sum1,!sum0 and of I0, I1 gets combined with the first stage of the normalizer, which performs a 25b left shift if I0 is all zero. By using 5-port mux latches, the delay of this muxing is hidden by the latch insertion delay.

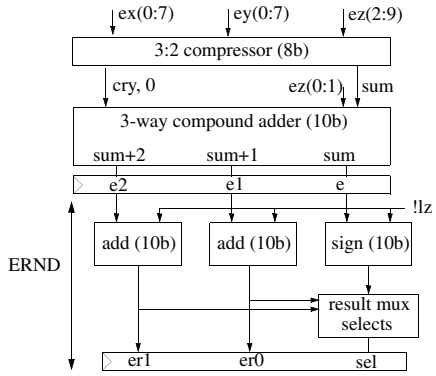

Figure 5. SPfpu exponent rounder ERND

The computation of the select signals based on the carry-out signal $cout$ is the critical path of any fraction adder. An extra carry-only network is used to speed up signal $cout$. The alignment information, operation type and sticky bit are still available before signal $cout$. The adder control logic therefore pre-computes 2 sets of select signals for $cout$ being 0 and 1 and selects between them before controlling the adder result mux.

4.4 SPfpu Backend

To meet the performance target, the normalizer and rounder stages have to be improved by 15 to 20FO4. The key enabler for this big saving is SPfpu's special floating point architecture: Denormal results are forced to zero, the binade for e=255 is used as normalized numbers, only round towards zero and non-trap exception handling are supported. The latter means that no exponent wrapping is needed.

Consequently, the fraction is just truncated saving a 24b incrementer on the fraction path. The correction of the LZA error is integrated in the result mux, which also merges in the special results Zero and Xmax and the integer or double precision result. Thus, the fraction rounding is reduced to a single 4-port mux.

The exponent rounder ERND (Figure 5) also profits from the simplified rounding but much less than the fraction path, so that ERND becomes timing critical. ERND adjusts the exponent by the number of leading zeros lz predicted by the LZA and generates the select signals for the result mux; this involves the checking for overflow, underflow and zero result:

$$\begin{aligned} e_r &= \begin{cases} e + 1 + !lz & \text{for lzaerr=0} \\ e + 2 + !lz & \text{for lzaerr=1} \end{cases} \\ OVF: &\quad e_r > emax = 255 \\ UNF: &\quad e_r < emin = 1 \\ Zero &= \text{UNF or fracZero or specialZero;} \end{aligned}$$

e is the exponent corresponding to the adder result.

To match the fast fraction rounder, the exponent rounding is improved by using the following logic optimizations in addition to highly tuned circuits and a dynamic result mux which speeds up the select path:

(1) The previous pipeline stages pre-compute multiple exponent values $e, e_1 = e + 1, e_2 = e + 2$ by using a 3-way compound adder. That saves ERND to increment the exponent, so that none of its adders and checkers requires a carry-in.

(2) The LZA error lzaerr is detected late in the normalization cycle. ERND therefore computes result exponents for lzaerr being 0 and 1:

$$e_{r0} = e_1 + !lz \qquad e_{r1} = e_2 + !lz.$$

The result mux then selects the proper exponent.

(3) ERND also computes two sets of select signals for e_{r0} and e_{r1} and selects the proper set based on lzaerr before latching it. The exponents are 10b two's complement numbers with a 127 bias; this format simplifies the OVF and UNF check. OVF is detected by checking the 2 msb of e_{r0} and e_{r1} for '01'. UNF is detected by checking the sign of the decremented exponent, i.e.:

$$\begin{aligned} e_{r0} < 1 &\quad \leftrightarrow \quad sign(e + !lz) \\ e_{r1} < 1 &\quad \leftrightarrow \quad sign(e_1 + !lz) = sign(e_{r0}). \end{aligned}$$

5 Design of the Double Precision FPU

The DPfpu core has a 9-cycle computation pipeline. Cycles 1–3 comprise a radix-4 Booth-multiplier and the alignment shifter. The alignment shift is done in a sum-addressed fashion similar to the SPfpu. Cycles 4 and 5 comprise the incrementer, end-around-carry adder, and the LZA. The result is fed into the 4-cycle combined normalization shifter and rounder.

The main challenge in the design of the DPfpu was the tight height and width constraints. The floorplan of the SPE allocates only 60 bits to the fraction datapath, which is not enough to accommodate the full 160 bit intermediate results. Most of the intermediate fractions are therefore folded into 2 to 3 rows. The multiplier outputs its 106-bit product in a carry-save format (S, T) folded into two rows. The aligner outputs its 160-bit aligned addend ALN folded into three rows; the most-significant row ALN_{hi} is sent to the incrementer, the lower two rows ALN_{mid}, ALN_{lo} are added to the product (S, T) in the end-around-carry adder.

Folding the aligner and the multiplier has benefits and drawbacks at the same time. It increases the height of the macros and also causes some long horizontal wires, e.g., where a signal in the multiplier 3:2-reduction-tree has to cross the boundary between low-order and high-order bits.

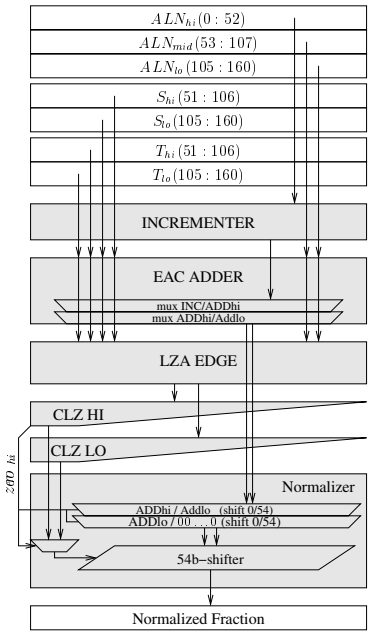

Figure 6. DPfpu normalization scheme

On the other hand, some other wires become shorter, e.g., where a select line in the shifter now has to cross only half the width. In the adder, the biggest problem due to folding are the propagate/generate lines which cross the boundaries of the two rows. The carry-lookahead structure was slightly rebalanced to account for the different wire delays. The LZA edge vector E is computed as described in [6]. As the inputs to the computation are folded, the edge vector E is also folded. Every bit of E is determined from 3 consecutive bits of $ALN_{mid,lo}, S, T$. In order to avoid long horizontal wires at the folding-boundaries, the three vectors have the 2 bits at the folding boundary replicated at both sides.

The incrementer and adder outputs are put onto a normalization shifter which is usually build as a standard barrel-shifter. Such a barrel shifter is usually addressed by a binary shift amount and hence shifts by powers of 2 in each shift stage. This approach without adjustment to the folded design would create many long horizontal wires. Instead we exploit the folding for the normalization shifter, which decreases wire length and thereby improves timing. The following description refers to Figure 6.

1. The first shift-stage selects between "INC,ADDhi" or "ADDhi,Addlo" vectors, as in the SPfpu. This first shift can be done without any horizontal wires, since the three parts are aligned horizontally due to the folding. The selection for this first shift depends only on the exponent difference which is known early. The multiplexer for this first shift stage is combined with the end-around-carry mux of the adder using a 6-port mux-latch; that saves a whole latch row.

2. The second shift stage is implemented as a shift by 54, which again can be done without any horizontal wiring due to the folding. The select signal for this shift stage is obtained by checking whether the high-part of the folded edge vector is all '0'. This would mean that the LZA anticipates that there are at least 54 zeros in front of the intermediate result, so the 54-bit shift must be performed. Otherwise the leading one appears to be in the first 54 bits, so this shift-stage does not shift. We will explain below how the possible error in the anticipation can be corrected.

3. To finish the normalization after the first two shift stages, we use a standard binary barrel shifter. This shifter has a maximum shift distance of 55 (i.e., 54 plus one position for a possible LZA error), and hence has considerably less wires than a full width shifter. The shift amount for this shifter is obtained from separate leading zero counters (LZC) for each of the two edge rows. Depending on the selection in the second shift stage, either the high- or the low-LZC output is chosen.

So, in order to exploit the folding for the normalization shifter, we do not use a single 106 bit wide LZC, but use two separate smaller counters. The high-order LZC also provides the zero indication for the high part. However, care must be taken because this zero-high indication can be wrong due to an LZA error, i.e., the edge vector may overestimate the number of leading zeros by one. Hence it may happen that the leading one is actually in the LSB of the high-order row of the sum, but the edge vector anticipates the leading one wrongly to be at the MSB of the low-order row. In this case the zero-high indication spuriously causes the 54-bit shift in the second stage. To solve this problem, the input to the third stage is simply prefixed with either '0' or the LSB of the sum-high row in order to prevent losing the leading one at that position.

If the addend is larger than the product, the normalization shift amount is usually determined from the exponent logic. It is simple to adjust the exponent logic to produce the correct shift-amount signals for the described normalization method.

6 Power Saving

The CELL processor is a multiprocessor chip with multiple SPEs. Since the FPUs are replicated for each SPE, their power consumption considerably contributes to the overall chip power consumption. Therefore a lot of effort was put into power-reduction mechanisms, such as clock-gating. Clock-gating turns off latches when they are not used; this decreases power consumption in two ways: first, the switching of the latches and the connected clock-wires is

prevented; second, since the latch content does not change if clocking is gated, the combinational logic in the fanout cone of gated latches does not switch either.

The FPUs described in this paper use three levels of clock gating: first, the complete FPU can be shut down by means of a global clock gating signal; this is used to shut-down most of the SPE when no active process is running. Second, during pipelined execution of instructions, only those pipeline stages are activated which contain valid instructions. Third, and most complex, we have implemented opcode- and data-dependent clock-gating. Only those parts within a pipeline stage are activated which are actually used for the operation. For example, the multiplier is disabled for add-instructions; incrementer and compound adder are disabled based on the alignment of addend and product.

7 Related Work

Multimedia extensions to microprocessors are available in most architectures nowadays. For example, IBM calls their extension to the PowerPC architecture VMX [17], which is also known as Freescale's AltiVec [2]. Intel extended their x86 architecture with SSE, and SSE-2 [20] and now SSE-3. AMD developed their 3DNow! multimedia extensions [14]. The floating-point architecture of those extensions usually closely follows the IEEE standard [8]. For example, VMX supports denormal numbers, but the architecture allows that these are executed slower than other operations. Only the round-to-nearest mode is supported in VMX. Intel's SSE supports denormal numbers and all 4 rounding-modes. In contrast, our FPU architecture is tuned to the target application in order to improve performance as much as possible. It only supports truncation rounding and does not handle denorms, NaNs and infinity.

The FPU of the Emotion Engine of the PlayStation 2 [7, 10], only supports truncation rounding and normalized numbers. It supports a fused-multiply-accumulate instruction with 2 operands for the multiply and a fixed accumulate register which is used as addend and as result. In contrast, our FPU supports 3-operand FMA operations that return their results into an arbitrary register in the register file. The Emotion Engine does not support double precision.

Several other publications describe designs of fused-multiply-add FPUs. The first FMA FPU was described in [13]; successors of this FPU are described in [5, 9]. Schwarz *et al.*[18] discuss FMA FPUs supporting denormal numbers in hardware. In this paper we have described some optimizations to the common FMA implementations; some of these optimizations are only possible in our application-tailored architecture, while others are applicable to any FMA FPU design.

Derivations from the common FMA design are proposed by Lang and Bruguera in [11] and Seidel in [19], reducing the latency by merging the addition and rounding of the fraction. Both designs focus on the fraction data path of a double precision multiply-add with normalized operands.

As shown in [19], the designs proposed by Seidel require fewer logic levels than that of Lang and Bruguera. In the following, we therefore compare our 6-cycle SPfpu core with a single-precision version of the Seidel's designs. The optimizations in the SPfpu frontend mainly target the efficient support of non-FMA operations such as estimates, converts and integer multiply operations. Those optimizations are, of course, also applicable to the Seidel's designs when extending the FMA pipe to a full FPU.

Seidel's designs are an adaptation of Farmwald's dual path algorithm for addition [3] to fused-multiply-add FPUs. Depending on the alignment of addend and product, he distinguishes five cases which can be implemented with different latencies. In the "far-out" path, the addend is so much larger than the product that the fractions of addend and product do not overlap. This is the fastest case which only requires a third of the latency of the slow near-path. In the near-path, addend and product have roughly the same exponent; an effective subtraction can cause massive cancellation.

This design is suited for processors which can benefit from a variable-latency FPU, e.g. out-of-order processors. Seidel also proposes a design with a fixed latency, which is better suited for in-order designs such as the SPE. There are two parallel data paths: one implements the four faster cases and the other one implements the slow near-path.

The delay of the fraction data path for the near case is the sum of the delays of the following components: a full-size 24×24 multiplier with an extra term for the aligned addend, a 50-bit adder with integrated rounding function, a re-complement stage, a 50-bit normalization shifter, and the result mux with select logic which performs the post-normalization and merges in special results and the result from the second data path.

In the SPfpu it turned out that the exponent rounding dominates the delay of the normalization-round stage of a single-precision FPU, when removing the fraction rounding. A similar effect is to be expected in a single-precision dual-path design which merges the fraction rounding with the adder. When applying the SPfpu optimizations for speeding-up the exponent rounding and result selection, the normalization-round stage of the SPfpu and of the dual-path design are equally fast. Any latency difference therefore has to result from the multiplier/aligner and adder stage.

Like in the dual-path design, the SPfpu has a full size 24×24 multiplier. The 96-bit aligner is in parallel to the multiplier. Due to several optimizations the aligner keeps up with the multiplier latency.

The timing critical path of the SPfpu adder stage goes

through a full-adder, a 48-bit carry-tree, a 2-port mux, and then selects the adder result. The adder result-mux integrates the re-complement and the 1st stage of the normalization shift, saving two mux stages. That partially compensates for the FA delay. The SPfpu adder is totally in parallel to the LZA. The dual-path design uses an adder with integrated rounder, which requires extra hardware and latency for the leading-zero prediction. Thus, the adder stages of the two designs have roughly the same latency.

When looking at the logic levels, the two FPU cores have about the same delay. The dual-path FPU requires two full-size multipliers and adders. That increases the area of a single-precision FPU by about 50%; for a double precision design, this increase is even larger. Such a large area overhead increases the power of the FPU, results in a non-trivial placement problem, and adds extra wire and transfer delay. The operands have to be distributed to the two data paths, and their results have to be collected; that easily accounts for 5fo4.

The SPfpu design has a significantly lower area and power at the same (or better) latency. It is optimized for its target applications, which mainly demand single-precision operations with truncation rounding. Single-precision operations with the other rounding modes are emulated in the DPfpu at a lower performance. The dual-path FPU supports all four rounding modes at the same latency. This additional functionality comes at a 50% larger area. Such a severe area penalty is not acceptable for an area and power efficient SPE design, especially since SPE is a building block of a multi-core CELL processor.

8 Summary

The vector FPU presented here occupies $2mm^2$, fabricated in the IBM 90nm SOI-low-k process. Correct operation has been observed up to 5.6GHz at 1.41V of supply voltage and $51°C$, delivering a single precision peak performance of 44.8Gflops.

The key enablers for this high-frequency low-latency power and area efficient FPU design are threefold: (1) Architecture and implementation are optimized for the target applications, such as real-time 3D graphics, trading uncritical function for performance. (2) Architecture, logic, circuits, and floorplan have been co-designed. (3) The pipeline stages are carefully balanced, achieving a maximal path delay difference of only 3%.

References

[1] *The IBM PowerPC Architecture: A New Family of RISC Processors*. Morgan Kaufmann Publishers, 1993.

[2] K. Diefendorff, P. Dubey, R. Chochsprung, et al. AltiVec(tm) Technology: Accelerating Media Processing Across the Spectrum. In *HOTCHIPS 10*, 1998.

[3] P. M. Farmwald. *On the Design of High Performance Digital Arithmetic Units*. PhD thesis, Stanford University, 1981.

[4] B. Flachs, S. Asano, S. H. Dhong, et al. The Microachitecture of the Streaming Processor for a CELL Processor. 2005. IEEE International Solid-State Circuits Conference (ISSCC), paper 7.4.

[5] T. N. Hicks, R. E. Fry, and P. E. Harvey. POWER2 floating-point unit: Architecture and implementation. *IBM Journal of Research and Development*, 38(5):525–536, Sept. 1994.

[6] E. Hokenek and R. K. Montoye. Leading-zero anticipator (lza) in the ibm risc system/6000 floating point execution unit. *IBM Journal of Researh and Development*, 34(1), 1990.

[7] N. Ide, M. Hirano, Y. Endo, et al. 2.44-GFLOPS 300-MHz Floating-Point Vector-Processing Unit for High-Performance 3-D Graphics Computing. *IEEE Journal of Solid-State Circuits*, 35(7), 2000.

[8] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[9] R. M. Jessani and C. H. Olson. The floating-point unit of the PowerPC 603e microprocessor. *IBM Journal of Research & Development*, 40(5), 1996.

[10] A. Kunimatsu, N. Ide, T. Sato, et al. Vector Unit Architecture for Emotion Synthesis. *IEEE Micro*, 20(2), 2000.

[11] T. Lang and J. Bruguera. Floating-point fused multiply-add wth reduced latency. In *IEEE Conference on Computer Design(ICCD)*, 2002.

[12] B. Minor. Personal communication.

[13] R. K. Montoye, E. Hokenek, and S. L. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research & Development*, 34(1), 1990.

[14] S. Oberman, G. Favor, and F. Weber. Amd 3dnow! technology: Architecture and implementation. *IEEE Micro*, 19(2), 1999.

[15] H.-J. Oh, S. M. Mueller, C. Jacobi, et al. A Fully-Pipelined Single-Precision Floating Point Unit in the Synergistic Processor Element of a CELL Processor. 2005. IEEE VLSI Conference, paper 2.4.

[16] D. Pham, S. Asano, M. Bolliger, M. N. Day, et al. The Design and Implementation of a First-Generation CELL Processor. 2005. IEEE International Solid-State Circuits Conference (ISSCC), paper 10.2.

[17] N. J. Rohrer, M. Canada, E. Cohen, et al. PowerPC 970 in 130nm and 90nm Technologies. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2004.

[18] E. M. Schwarz, M. S. Schmookler, and S. D. Trong. Hardware implementations of denormalized numbers. In *IEEE Symposium on Computer Arithmetic (Arith 16)*, 2003.

[19] P.-M. Seidel. Multiple path ieee floating-point multiply accumulate. In *46th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2003.

[20] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, Q2, 1999.