Automatic Formal Verification of Fused-Multiply-Add FPUs

Christian Jacobi[1]   Kai Weber[1]   Viresh Paruthi[2]   Jason Baumgartner[2]

[1] IBM Deutschland Entwicklung GmbH, Boeblingen, Germany

[2] IBM Systems Group, Austin, TX

Abstract

In this paper we describe a fully-automated methodology for formal verification of fused-multiply-add floating point units (FPUs). Our methodology verifies an implementation FPU against a simple reference model derived from the processor's architectural specification, which may include all aspects of the IEEE specification including denormal operands and exceptions. Our strategy uses a combination of BDD- and SAT-based symbolic simulation. To make this verification task tractable, we use a combination of case-splitting, multiplier isolation, and automatic model reduction techniques. The case-splitting is defined only in terms of the reference model, which makes this approach easily portable to new designs. The methodology is directly applicable to multi-GHz industrial implementation models (e.g., HDL or gate-level circuit representations) that contain all details of the high-performance transistor-level model, such as aggressive pipelining, clocking, etc. Experimental results are provided to demonstrate the computational efficiency of this approach.

1 Introduction

Traditionally, industrial floating point units (FPUs) are validated by simulation, often using targeted techniques such as specialized testcase generators [1]. While such approaches are efficient at exposing many bugs, they are based on *incomplete* methods which cannot achieve full coverage, e.g. evaluation of all operand combinations over all rounding modes and exception states. To compound the coverage problem, designs face shorter time-to-market (hence lesser verification time) from generation to generation, require higher clock speeds thus a larger degree of pipelining, and acquire additional features such as clock gating for low-power. Formal and semi-formal verification techniques constitute an increasingly-prevalent mechanism by which to attempt to close the coverage gap imposed by simulation. For example, numerous industrial approaches have proposed the use of a combination of automatic methods and manual theorem-proving techniques to yield complete proofs of correctness of FPUs [2, 3, 4].

In this paper, we present an efficient, fully-automated methodology for the verification of fused-multiply-add (FMA) FPUs. This methodology targets exhaustive verification of the complex dataflow of such FPUs, focusing on the arithmetic correctness of a single arbitrary instruction. The presented approach compares the FPU implementation against a simple reference model derived from the processor's architectural specification, which may include all aspects of the IEEE specification such as denormal operands and exceptions. Our methodology is portable to simulation, emulation, semi-formal, and formal verification frameworks; no customized toolset is necessary. To enable formal proofs, we use a combination of case-splitting and automatic model reduction tech-

niques, and isolate the multiplier for dedicated verification. All case-splitting is defined in terms of the reference model, ensuring that this overall methodology, as well as the reference model itself, is easily portable to various implementations. The presented methodology is directly applicable to an HDL or gate-level implementation model without a need for manual abstraction. Experimental results demonstrate applicability to HDL implementations of high-performance industrial FPUs which are combinationally-equivalent to – i.e., share the same state encoding as – the (primarily custom-designed) transistor-level circuit, as is a common industrial design style [1]. Coupled with the use of a Boolean equivalence checker to correlate the HDL implementation against the fabricated schematics [14], this overall approach enables a seamless proof of datapath correctness from the transistor schematics all the way up to the architecture-level specification.

The novel contributions of this paper are the following. First, our paper is the first to address formal datapath verification of FMA FPUs using only common automatic verification tools. Second, ours is the first to address denormal operands and results for FMA instructions. Third, our methodology is easily portable to new implementations because the portable reference model encompasses the case splitting strategy. Fourth, our reference model is also reusable in platforms such as simulation and emulation.

The rest of the paper is organized as follows. Section 2 provides an overview of our verification framework. Section 3 details the reference model and Section 4 describes the case-splitting strategy used to reduce formal complexity. Section 5 provides experimental results and Section 6 discusses the portability of this method, in particular to fully IEEE-compliant FPUs. Section 7 discusses related work. In Section 8, we summarize and conclude the paper.

2 Verification Overview

The FPU under verification supports the double-precision FMA instruction and its derivatives as defined in the PowerPC[1] architecture [5]. FMA computes $A \times B + C$ on operands A, B, and C. Instructions such as $A \times B$ and $A + B$ are computed as $A \times B + 0$ and $A \times 1 + B$, respectively. All four IEEE rounding modes are supported. For implementation details on FMA FPUs refer to [6].

For clarity, the discussion in this paper primarily focuses on the verification of an FPU which produces IEEE-compliant denormal results, but maps denormal operands to zero. In Section 6 we describe the extension of our methodology to FPUs which support denormal operands. In this paper we do not discuss the details of the special values NaN and infinity; the cases involving such operands are discharged trivially by the formal algorithms.

Our verification methodology compares an implementation FPU against a simple reference model. A *driver* generates the instructions (opcodes) and the operands, and dispatches them into both

---

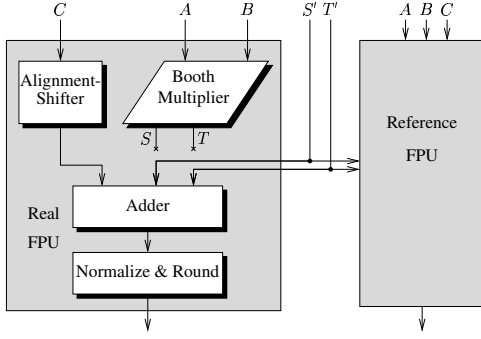[1] PowerPC is a registered trademark of IBM Corporation.

Figure 1: Removal of the multiplier from the cone-of-influence

FPUs. After all computations are finished, the results are compared and an error is flagged if the results differ. The two FPUs are evaluated independently; there is therefore no need to establish corresponding pipeline stages between these FPU models. Because the number of steps needed to complete a single floating point computation is bounded, we may cast our verification problem as a bounded check using SAT- and BDD-based techniques instead of requiring more expensive unbounded algorithms.

Our goal is to exhaustively verify the *datapath functionality*. We aim at bugs which cause numerically incorrect results (e.g., incorrect rounding) instead of bugs which, for example, result from unintended interaction between multiple instructions. Hence we restrict our verification to a single arbitrary instruction issued into an empty FPU. Note that bugs in the inter-instruction control may often cause data errors. However, these are often considered a separate class of bugs which may be verified separately from the numerical computation, as per [7, 8]. Though we do not cover inter-instruction aspects, we do exhaustively verify the control logic specific to the execution of a single instruction, including opcode decoding and aspects of the clock gating control (which is particularly nontrivial for modern processors).

There are three building blocks in the FPU that are major hurdles for formal algorithms: namely, the multiplier, the alignment shifter that aligns the addend to the product, and the normalization shifter that eliminates leading zeros in the intermediate result before rounding. Each of these building blocks leads to run-time explosion of SAT, and memory-explosion of BDDs.

To circumvent the difficulties posed by the shifters, we split the overall verification problem into sub-problems as discussed in Section 4. Each sub-problem restricts the shift-amount for both shifters, causing them to "collapse" into simple wires, similar to the case-splitting strategy employed in [9] for verifying floating-point addition. We extend this approach to FMA instructions, and to handle denormal results and operands (refer to Section 6).

To circumvent the difficulties posed by the multiplier, we remove it from the cone-of-influence of the FPU outputs, as depicted in Figure 1. Note that the inputs to the multiplier are the significands of the operands A and B, and its outputs are S and T such that the sum of S and T equals the product of the input significands. Additionally, S and T contain some leading bits known as "hot-ones" which are an artifact of Booth-encoding. To eliminate the multiplier, we override S and T with new signals S' and T', causing multiplier logic to become sinkless and consequently to be removed from the cone-of-influence. Note that the multiplier in both the real and reference FPU is overridden by S' and T' for consistency.

The FPU relies on two properties that the real multiplier satisfies. We therefore define S' and T' such that they may take arbitrary

values which adhere to these two properties: (i) the sum of S' and T' lies in the range $[1, 4)$ for normal input significands, and (ii) the most significant bits of S' and T' can take only specific values, depending on the exact implementation of the multiplier array (hot-one bits). To ensure soundness, we formally verify that the original multiplier also satisfies the two constraints, i.e. S' and T' *over-approximate* the behavior of S and T. Hence, all possible outputs of the real multiplier are covered in our verification. Practically, this is a simple proof obligation for SAT, since it requires only a fraction of the multiplier logic in the cone-of-influence.

Our formal methodology thus verifies the entire numerical computation except for the actual significand multiplication. We thus cover alignment shifts, end-around-carry addition and leading zero anticipation, normalization with sticky-bit computations, mis-anticipation correction, rounding, overflow and underflow detection, etc. Such implementations are notoriously customized and error-prone, while the multiplier itself is relatively straightforward in comparison. Since our methodology is portable to alternate frameworks, we also validate the design without the multiplier overrides or case-splits using simulation and semi-formal methods. The multiplier may also be formally verified using existing techniques [10, 11] to ensure completeness of the overall process.

3 Reference FPU

In this section we describe the reference FPU, against which we compare the real FPU. The reference FPU is written in VHDL[2] and, as with the real FPU, is transformed into a netlist using a standard VHDL compiler. The primary goal of the reference FPU is simplicity: it must be a concise specification, not prone to the introduction of bugs via the high-performance design and micro-architecture tricks that complicate the real FPU. In industrial designs, blocks such as adders, shifters, and leading-zero counters are often designed at the gate-level in order to match the high-performance circuit structure and facilitate combinational equivalence checking between the two representations [1, 14]. In the multi-GHz design domain, this means that such blocks must be pipelined and hence cannot be specified with high-level VHDL operators such as "$+$" and "sll". In contrast, we do use such VHDL operators in our reference model since we need not match circuit characteristics or latch points. This simplicity comes at the cost of increased gate count and greater structural dissimilarity with the real FPU which precludes redundancy removal techniques from significantly simplifying the verification problem. This is inevitable with a portable reference model – and addressed by our overall methodology.

Altogether, the core of the reference FPU is only 300 lines of VHDL; the handling of special cases such as operations on NaN and infinity requires another 150 lines of trivial *if-then* constructs. The reference FPU is thus approximately 450 lines of VHDL, versus approximately 15,000 for the real FPU.

Algorithm. We now describe the reference FPU algorithm and give some details of its implementation. The reference FPU is required to compute $A \times B + C$ for three operands A, B, and C. Other operations like addition or multiplication can be derived from FMA in an obvious way. Let s_a denote the sign, e_a the unbiased exponent, and f_a the significand including the implicit bit of the operand A (similarly for B and C). We define $s_p = s_a$ xor s_b, $e_p = e_a + e_b$,

---

[2]Others have formalized the IEEE standard in a theorem prover as a mathematical specification [12, 13]; we use an HDL-based reference model for portability to simulation, emulation, semi-formal, and formal verification frameworks.
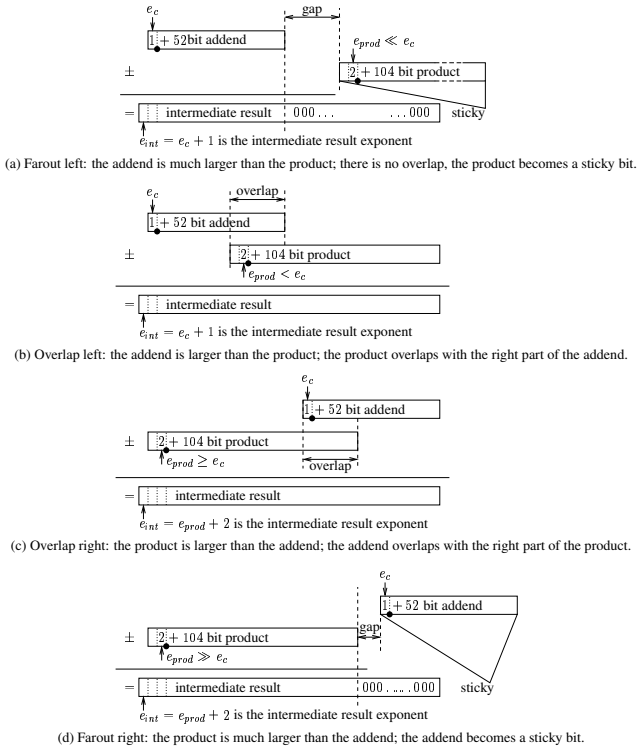
(a) Farout left: the addend is much larger than the product; there is no overlap, the product becomes a sticky bit.

(b) Overlap left: the addend is larger than the product; the product overlaps with the right part of the addend.

(c) Overlap right: the product is larger than the addend; the addend overlaps with the right part of the product.

(d) Farout right: the product is much larger than the addend; the addend becomes a sticky bit.

Figure 2: The four cases of the reference FPU's algorithm

and $f_p = f_a \times f_b$. The FMA operation can be rewritten as

$$
\begin{aligned}
& A \times B + C \\
& = [(-1)^{s_a} \times 2^{e_a} \times f_a] \times [(-1)^{s_b} \times 2^{e_b} \times f_b] \\
& \quad + [(-1)^{s_c} \times 2^{e_c} \times f_c] \\
& = ([(-1)^{s_p} \times 2^{e_p} \times f_p] + [(-1)^{s_c} \times 2^{e_c} \times f_c]).
\end{aligned}
$$

Because the operand significands have 1 bit before and 52 bits behind the binary point, f_p has 2 bits before and 104 bits behind the binary point, thus, a total of 106 bits. Let $\delta := e_p - e_c$ denote the difference of the product exponent and the addend exponent. For simplicity, the reference FPU is implemented as a case-statement distinguishing the following four cases. In a real FPU, one would reuse as much logic as possible between these cases, decreasing circuit size but increasing implementation complexity.

Far-out left: $\delta \leq -55$. In this case, the addend is much larger than the product, hence lies completely to its left as in Figure 2(a). The addend is thus used as an intermediate result and the product is reduced to a single sticky bit used for rounding.[3]

Overlap left: If $\delta \in \{-54, \ldots, -1\}$, the addend is larger than the product but the product vector overlaps with the right end of the addend vector as in Figure 2(b). In this case the intermediate result is computed by adding/subtracting the properly aligned product to the addend, depending on the signs and opcode. Aligning means shifting by an amount depending upon δ.

Overlap right: $\delta \in \{0, \ldots, 105\}$. Similar to the previous case, now the addend overlaps with the right side of the product

---

[3]The -55 boundary is derived as follows (cf. Figure 2(a)): the addend has 52 bits behind the binary point, while δ reflects the distance between the most-significant addend bit and the first bit left of the binary point of the product. Hence, if this distance is 52, i.e. $\delta = -52$, the two bits left of the binary point of the product lie below the two LSBs of the addend. At a distance of 54, the product lies directly behind the addend, but then the MSB of the product would be the guard bit for rounding. At a distance of greater or equal to 55 ($\delta \leq -55$), the product is completely reduced to a sticky-bit for rounding. The other boundaries can be derived similarly.
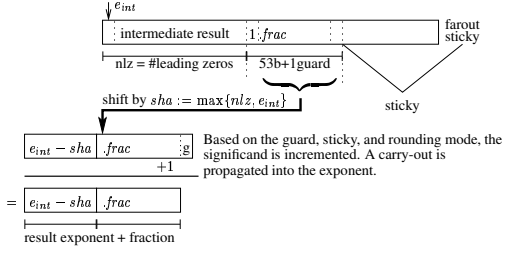

Figure 3: Block level diagram of the reference FPU's rounder

as in Figure 2(c). Here the intermediate result is computed by adding/subtracting the properly aligned addend to the product.

Far-out right: $\delta \geq 106$. This is the case where the addend is much smaller than the product, hence lies completely to its right as in Figure 2(d). In this case, the product becomes the intermediate result, and the addend is reduced to a sticky bit.

The maximum width for the intermediate result in these cases is 161 bits – this accounts for 1 carry-out bit, 53 bits of the addend, 106 bits of product, plus 1 guard bit. In all cases an intermediate result of this width is computed; if the overlap is small (or a far-out case happens), the intermediate result is padded with 0's. The intermediate exponent e_{int} is the weight of the most-significant bit.

The intermediate result is next passed to the rounder, depicted in Figure 3. First, the rounder counts the number of leading zeros nlz of the intermediate result. This is necessary since, in the overlap cases, the addend and the product may cancel-out some positions. Next, the intermediate result is shifted to the left by nlz places, and the intermediate exponent is adjusted by subtracting nlz from it. However, the shift-amount is bounded if necessary to prevent the exponent from becoming negative; note that a denormal result may be generated here due to such partial normalization. Finally, the normalized intermediate result is rounded according to the rounding mode and the bits behind the significand, as well as the sticky bits generated in the two far-out cases. The rounder also produces flags such as *overflow*, *underflow*, and *inexact*, which are readily computed from the exponent and the rounding decision.

4 Case-Splitting

The FPU's alignment and normalization shifters are inherently difficult for both BDD- and SAT-based algorithms, due to shifts of variable values by variable amounts. In order to make the verification task feasible, the overall problem must be divided into sub-cases. The general idea of case-splitting is to fix the shift amounts of both shifters to a constant in each case, rendering the shifters amenable to BDD- and SAT-based analysis within each case. To ensure complete coverage, all possible combinations of shift amounts must be included in at least one case. This general scheme of case-splitting was also used in [9] for the verification of floating-point addition. We now describe the case-splits needed for the verification of FMA, and contrast to those of [9] below. First, we distinguish between overlap and far-out cases:

i) **Far-out:** in the far-out cases, the intermediate result is the product and the addend is reduced to a single sticky bit, or vice versa. This case does not need to be sub-divided further.

ii) **Overlap:** this is the case where the addend and the product overlap, i.e., $\delta \in \{-54, \ldots, 105\}$. The alignment-shift amount is determined by δ. This case is divided into a sub-case for each of the 160 different δ-values to trivialize the

alignment shifter. These 160 cases belong to two classes:

(a) No cancellation: if $\delta \notin \{-2, -1, 0, 1\}$, the most-significant bits of the addend and the product are at least two bit-positions apart. In this case, no massive cancellation can occur. The small normalization shift amounts between 0 and 2 due to carry-outs or borrow-outs during the addition can be handled by the formal algorithms without further splitting.

(b) Cancellation: if $\delta \in \{-2, -1, 0, 1\}$, the product and the addend may cancel out leading bits when doing effective subtraction. In this case the normalization shift amount is determined by the leading-zero counter in the reference FPU and by a leading-zero-anticipator [6] in the real FPU. Both methods normalize at most to the extent that the exponent does not drop below 0 as per Figure 3. Since in these cases the normalization shifter can perform shifts by arbitrary amounts, we sub-divide these four δ-values into sub-cases for every normalization-shift-amount, i.e., into 107 sub-cases.[4]

We thus have one far-out case, 156 non-cancellation overlap cases, and 4×107 cancellation overlap cases, totaling 585 cases which are independently verified. Note that these cases naturally reflect the way in which *every* FPU must generate results, from simple reference models to multi-GHz implementations using any variety of design and micro-architectural techniques. We thus feel that these case-splits will directly apply to *any* design, which is substantiated by our having successfully applied them to four industrial FPUs.

There is an interesting case hidden in the above case-splits: the product of two *normal* numbers may be *denormal*. If such a product is added to a zero addend, the result is the appropriately denormalized product. For example, $e_{prod} = -15$ and $e_c = 0$ would cause a denormalization of the significand product by 15 places to the right. In the reference FPU, this is handled as an overlap-left case, because $\delta = -15$. This provides the correct number of 0's in front of the product. Since the intermediate exponent $e_{int} = e_c + 1 = 1$ in this case, the normalization shift amount is limited to 1. This is also a fixed shift amount, hence this can be handled simultaneously with the standard $\delta = -15$ non-cancellation case-split as described above. Note that this case also applies to the multiplication-instruction, where the addend is always zero.

Constraining. Case-splitting is achieved by "constraining" certain signals in the reference FPU corresponding to the sub-case. Formal algorithms use the constraints to define a care-set, and may simplify their processing with respect to the defined care-set.

The distinction between far-out and overlap, and between the different δ-values, is defined by a constraint on the operand exponents. Specifically, we define a constraint $C_\delta := (e_a + e_b = e_c + \delta)$ for every δ-case. The constraint for the far-out cases is the respective inequality over the operand exponents.

As regards the additional constraints for the cancellation cases, the normalization shift amount depends on the number of leading zeros of the intermediate result and the intermediate exponent e_{int}, cf. Figure 3. Hence, it is difficult to define these constraints directly upon the operands while still ensuring that the case-split is complete and still sufficient to trivialize the normalization shifter. For this reason, the normalization shift constraint is defined directly on the shift-amount signal sha of the reference FPU. We define a constraint $C_{sha} := (sha = X)$ for all 107 possible shift-amounts.[4]

---

[4]This includes all 106 possible shift amounts, plus one additional case $C_{sha/rest} := (sha > 106)$ to cover the remaining values. The $C_{sha/rest}$ case defines an empty care-set, hence this case is trivially discharged; it is checked only for completeness.

The disjunction of all the cases is easily provable as a tautology, guaranteeing completeness of our methodology.

The definition of the C_{sha} constraints is simple, since the sha signal may be directly referenced from the reference FPU. However, the logic driving this signal has considerable complexity. The number of leading zeros in the intermediate result is obtained from a 161-bit addition of the product and the addend. The addition itself is based on the (constrained) alignment shifts of the product and the addend. Nevertheless, the C_{sha} constraint alone suffices to bound BDD size both for the reference and real FPU computations without any explicit constraint on the real FPU. This is a significant observation and demonstrates the benefit of using constraints as a mechanism for case-splitting: the sha signal is a function of the operand and opcode values represented as a BDD. A constraint on sha is therefore effectively a constraint on the operand and opcode; although the BDD-minimization algorithms are heuristic, they are powerful enough to automatically carry over this constraint from the reference to the real FPU, effectively constraining the real FPU's shift amount as well. This is non-trivial, considering that the real FPU's shift-amount is obtained from a completely dissimilar piece of logic, namely a limited leading-zero anticipator (LZA) working in parallel with the adder (such as the one described in [6]). The shift-amount signal in the real FPU may even differ in value from sha in the reference FPU, e.g., offset by a constant preshift, or simply offset by one due to the possible shift-amount anticipation error inherent in the LZA.

Discussion. The distinction between far-out and overlap with and without cancellation cases as described above was also done in [9]. They do not need to sub-divide the $\delta = -2$ case into different normalization cases, since this case can lead to cancellation only in FMA due to the two bits in front of the binary point of the product. In our approach, the C_δ constraint is defined similar to [9]. The normalization shift case-split, however, is done differently: Chen and Bryant define their constraints directly on the operand significands, while we use the reference FPU's sha signal, which imposes a constraint on the operands implicitly. This is necessary in our methodology because denormal results may be produced (whereas [9] does not verify denormal results), which requires a comparison of the number of leading zeros to the intermediate exponent. This becomes even more complex with denormal operands as discussed in Section 6. Performing such a case splitting strategy for FMA explicitly on the operands would thus be error-prone and amplify the difficulty of ensuring completeness across the cases. The verification task in [9] is further sub-divided into effective addition/subtraction because cancellation can only happen during the latter. In our setting, this additional split is unnecessary.

5 Verification results

Our experiments were run on IBM pSeries computers with POWER4 processors running at 1.7GHz, using the IBM internal verification tool *SixthSense*. All designs are mapped into a netlist representation containing only 2-input AND gates, inverters, and registers, using straight-forward logic synthesis techniques [15]. As described, our real FPU comprises approximately 15,000 lines of VHDL. After compilation and phase abstraction [16], the netlist of the real FPU has approximately 4,800 registers and 55,000 AND gates. We employed automated redundancy removal algorithms [15] to reduce the size of the netlist prior to application of BDD- and SAT-based analysis. The accumulated run-time for ver-

Instr.	Case	Nodes [10^6]		Time [mins]	
		avg	max	avg	max
add	overlap w/ cancellation	0.2	0.4	3	5
add	overlap w/o cancellation	0.3	0.5	3	4
add	far-out	*n/a*	*n/a*		12
mult	n/a	*n/a*	*n/a*		5
FMA	overlap w/ cancellation	6.9	26.0	8	24
FMA	overlap w/o cancellation	2.1	4.7	5	10
FMA	far-out	*n/a*	*n/a*		53

Table 1: BDD nodes and runtimes for the double-precision cases

ifying the add-instruction was approximately 16 hours. Multiplication took only 5 minutes; the accumulated run-time for all FMA jobs was 73 hours. A complete regression of all jobs takes less than a day when running 10 jobs in parallel. Table 1 lists the average and peak number of BDD nodes and run-times for the different types of runs (*n/a* refers to cases where SAT was used).

We use both SAT- and BDD-based symbolic simulation as our verification engines for FMA. The SAT-solver uses an interleaved BDD-sweeping and structural satisfiability checking technique as described in [15], and operates upon an unfolded combinational netlist. The BDD-based symbolic simulator operates directly upon the sequential netlist. Satisfiability checking was used to verify the "far-out" cases and took 53 minutes to complete. The SAT-solver is able to identify that the shifters which align the addend to the product are not needed in this case, and thus automatically removes these unused shifters from the cone-of-influence. In contrast, BDD-based symbolic simulation would build the BDDs for these unneeded shifters anyway because only in later time-steps would it become apparent that they are not needed for this case. The overlap cases were verified using BDD-based symbolic simulation with the described case-splits applied as discussed in Section 4.

The addition instruction was verified with the multiplier in the cone-of-influence since the second operand of the multiplication is 1.0; constant propagation automatically replaces the multiplier by trivial logic. The applied case-splitting is similar to the FMA instruction, with the unnecessary splits removed.

We used satisfiability checking for the verification of the multiply instruction. After the multiplier is removed from the cone-of-influence, the only difficult aspect of the proof is the possible denormalization (cf. Section 4). Verification of this is possible without case-splitting because the SAT solver and redundancy removal techniques are able to identify structural similarities between the denormalization shifters in the real and the reference FPU, and use this fact to simplify the satisfiability check.

For the cases solved by BDD-based methods, we provided an efficient statically-derived variable ordering to the symbolic simulator. Initially, we attempted to use more generically-computed initial orders coupled with dynamic variable reordering. However, those runs consumed considerably more time and memory, even suffering from memory-explosion at times. The superior orders are intuitively derivable: the operand exponents come first, followed by the fractions intertwined with the pseudo-inputs S' and T' for the multiplier override; the fractions and S' and T' are aligned according to the δ of each individual run (similar to those in [9]). These orders are readily portable to other FPU designs since they are defined in terms only of the operands and pseudo-inputs. Using these orders, we disable dynamic variable ordering as it unnecessarily consumes run-time without yielding a superior order.

We also experimented with different BDD minimization algorithms (using the care-sets defined by the constraints). The BDD operation *constrain* [17] was overall the best choice: it is fast when the number of nodes is manageable. More aggressive minimization algorithms yielded greater reductions in the peak number of BDD nodes, but their overall run-time was significantly higher.

6 Portability of this methodology

We have adapted the reference FPU and the verification environment to several different FPUs under development at IBM, some of which are fully IEEE-compliant and thus handle denormal operands and results in hardware, and may also
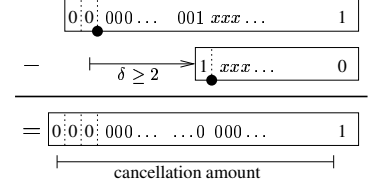

Figure 4: Denormal operands may cause cancellation even for large δ's

perform exponent-wrapping on overflows and underflows. The adaptation of the reference FPU and the verification environment was done within only a few days. The enhancement of the reference FPU to interpret denormal numbers according to the IEEE standard was simple; note that this modification is a one-time effort which does not need to be redone for subsequent FPUs. Also the rules for signals S' and T', overriding the multiplier output, had to be adapted: the properties asserted for S' and T' now also specify the range of the product if one or both operands are denormal. For example, it is asserted that the product lies in the range $[0, 1)$ if both operands are denormal. This is the obvious generalization of the property "the sum of S' and T' lies in the range $[1, 4)$" described in Section 2. It is still simple for SAT to prove that the real multiplier also satisfies these properties and hence that the multiplier isolation is sound.

The rest of the methodology can be applied with little modification. The case-splitting strategy can be reused, due to the facts that (i) the motivation for each individual case-split is architecture-based rather than implementation-specific, and (ii) the definition of the case-splits is done in terms of the reused reference FPU instead of the real FPU. However, more overlap cases can now lead to cancellation in the event of denormal operands. This happens, for example, when a normal and a denormal number are multiplied yielding a product with leading zeros. In this case the (normal) aligned addend might just start at the leading '1' of the product (Figure 4), possibly causing massive cancellation. Hence, all overlap cases with potential cancellation must be sub-divided in order to handle the normalization-shifter. This is a straightforward enhancement to the case-splitting described in Section 4; each case has similar runtime to the cases described in Section 5. Although the number of cases becomes larger (quadratic in the number of δ-cases), the overall task is still tractable for discharging automatically, especially given a larger network of machines to parallelize the task. We discharge the approximately 17,000 cases with an accumulated runtime of 1416 hours, which takes approximately one day given 50 machines. We are currently experimenting with techniques to reduce the total number of cases, to decrease the cumulative run-time; we omit such details for the sake of brevity.

Adaptations of our methodology to subsequent FPU designs required less than one day of effort each. Only the rules for S' and T' had to be adjusted, as these are the only implementation-specific

aspect of our methodology. This clearly demonstrates the portability of this methodology.

7 Related Work

Floating-point verification has been extensively studied both in academia and industry. However, we are not aware of any fully-automated attempts to formally verify FMA datapaths, nor any which cover denormal operands or results for such operations.

Researchers at Intel [3, 18] and AMD [19, 4] have also applied formal methods to the verification of FPUs. At Intel, FPUs are verified using a customized toolset combining STE and theorem-proving, likely requiring implementation-specific manual effort. Recently, a sketch of the application of this approach to the verification of Intel's Itanium fused-multiply-add datapath was provided in [20]. In contrast to our work, their approach does not address details of the case-splits necessary for tractability, and does not cover denormal results nor operands. The verification at AMD uses the theorem prover ACL2, which requires manually-guided proofs. Our approach is fully automatic and portable.

Aagaard and Seger [21] verified a floating-point multiplier using a customized toolset combining STE and theorem proving. Their multiplier does not include a denormalization shifter for multiplication because it traps on denormal results. The multiplier array is verified by sub-dividing the hardware into a Booth-decoder and an addition network. The two parts are combined using theorem proving. A similar approach is taken in [18] for the verification of Intel's Pentium 4 multiplier.

Berg and Jacobi [13] used the PVS theorem proving system to verify an IEEE-compliant FPU against a mathematical formulation of the IEEE standard. The use of a theorem prover is quite labor intensive, especially at low levels of abstraction such as the gate-level. Furthermore, such low level proofs are tailored to the specific implementation, and hence are difficult to adapt to a different FPU.

The approach of Chen and Bryant [9] to verify floating-point addition comes closest to our work. In particular, our case-splitting strategy is similar to theirs, as discussed in Section 4. We extend their technique for applicability to the fused-multiply-add instruction, and to cover denormal results and operands. Additionally, our approach uses standard BDD- and SAT-based analysis (exploiting bit-level redundancy removal), whereas that of [9] requires word-level PHDDs for computational efficiency.

Various research has addressed the automatic verification of integer multipliers, e.g., [10, 11]. One promising future direction is to incorporate such techniques into our methodology to verify the multiplier along with the rest of the FPU, without isolation.

8 Summary

We have presented a fully-automated approach for verifying floating-point addition, multiplication, and fused-multiply-add instructions by comparing their implementation against a simple reference FPU. To reduce formal complexity, we isolate the multiplier from the cone-of-influence by overriding it with conservative variables. We additionally present a case-splitting strategy necessary to make the verification feasible with automatic formal methods. All case-splitting is defined exclusively in terms of the reference FPU, and our specification is synthesizable (hence portable to simulation, emulation, and arbitrary formal frameworks without a need for customized tools), rendering a highly-portable overall methodology.

We have adapted this methodology to partially as well as fully IEEE-compliant designs. Our experiments demonstrate this to be an efficient and scalable approach for the verification of floating point datapaths. The methodology is currently being applied to four FPUs under development at IBM. Dozens of high-quality bugs have been exposed by these efforts, some of which would likely have slipped into first silicon otherwise.

References

[1] J. M. Ludden *et al.*, "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM Journal of Research & Development*, vol. 46, no. 1, 2002.

[2] J. S. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5K86 floating point division," *IEEE Transactions on Computers*, vol. 47, no. 9, 1998.

[3] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating point hardware," *Intel Technology Journal*, vol. 3, no. 1, 1999.

[4] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal of Computation and Mathematics*, vol. 1, 1998.

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

[6] 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*, vol. 34, no. 1, 1990.

[7] C. Jacobi, "Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving," in *CAV*, 2002.

[8] M. Kaufmann and D. Russinoff, "Verification of pipeline circuits," in *ACL2 Workshop*, 2000.

[9] Y.-A. Chen and R. E. Bryant, "Verification of floating point adders," in *CAV*, 1998.

[10] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Design Automation Conference*, 1995.

[11] D. Stoffel and W. Kunz, "Verification of integer multipliers on the arithmetic bit level," in *ICCAD*, 2001.

[12] P. S. Miner, "Defining the IEEE-854 floating-point standard in PVS," Tech. Rep. TM-110167, NASA Langley Research Center, 1995.

[13] C. Berg and C. Jacobi, "Formal verification of the VAMP floating point unit," in *CHARME*, 2001.

[14] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, "Verity - a formal verification program for custom CMOS circuits," *IBM Journal of Research & Development*, vol. 39, 1995.

[15] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on Computer-Aided Design*, vol. 21, no. 12, 2002.

[16] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz, "An abstraction algorithm for the verification of level-sensitive latch-based netlists," *Formal Methods in System Design*, no. 23, 2003.

[17] O. Coudert, C. Berthet, and J. Madre, "Verification of sequential machines using boolean functional vectors," in *Applied Formal Methods for Correct VLSI Design*, 1989.

[18] R. Kaivola and N. Narasimhan, "Formal verification of the Pentium 4 floating-point multiplier," in *DATE*, 2002.

[19] D. M. Russinoff, "A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor," vol. 1954 of *LNCS*, 2000.

[20] A. Slobodova and K. Nagalla, "Formal verification of floating point multiply add on Itanium processors," in *Workshop on Designing Correct Circuits*, Mar. 2004.

[21] M. D. Aagaard and C.-J. H. Seger, "The formal verification of a pipelined double-precision IEEE floating-point multiplier," in *ICCAD*, Nov. 1995.