Formally Verifying IEEE Compliance of Floating-Point Hardware

John O'Leary, Xudong Zhao, Rob Gerth, Carl-Johan H. Seger
Strategic CAD Labs, Intel Corporation, Hillsboro OR

Index words: floating-point, IEEE compliance, formal verification, model checking, theorem proving

Abstract

This paper describes the formal specification and verification of floating-point arithmetic hardware at the level of IEEE Standard 754. Floating-point correctness is a crucial problem: the functionality of Intel's floating-point hardware is architecturally visible (it is documented in the programmer's reference manual [1] as well as an IEEE standard [2]) and, once discovered, floating-point bugs are easily reproduced by the consumer. We have formally specified and verified IEEE-compliance of the Pentium® Pro processor's FADD, FSUB, FMUL, FDIV, FSQRT, and FPREM operations, as well as the correctness of various miscellaneous operations including conversion to and from integers. Compliance was verified against the gate-level descriptions from which the actual silicon is derived and on which all traditional pre-silicon dynamic validation is performed. Our results demonstrate that formal functional verification of gate-level floating-point designs against IEEE-level specifications is both feasible and practical. As far as the authors are aware, this is the first such demonstration.

Introduction

The main objectives of this paper are to describe how we specify IEEE compliance of gate-level designs, and how we employ theorem-proving and model-checking tools to formally verify that the designs meet their specifications. A further objective is to relate our experience verifying the gate-level description of the Pentium® Pro processor's floating-point execution unit.

Work on this project began in July 1997, after the discovery of a post-silicon erratum in the Pentium Pro processor's floating-point execution unit (FEU). The discovery of this erratum was especially disturbing to us because the Pentium Pro FEU had been the subject of a previous formal verification project [3]. The work we describe in this paper extends and improves upon the previous effort in the following respects. First, our current specifications cover numeric correctness at the level of the IEEE standard, while the specifications used in the previous verification project are at the level of functional blocks within the FEU. Second, our verification is much more comprehensive: in the current work we verify correct datapath functionality for all floating-point operations implemented in the FEU, including all precisions, rounding modes, and flags. The focus in the earlier verification was core datapath functionality only. Perhaps the most important improvement over our previous work is methodological: whereas in the previous work we had no metric for the completeness of a property set (leaving the door open for errata to escape), we now employ formal proof of compliance to an IEEE-level specification as our completeness criterion.

Formal verification has also been applied to the AMD-K7* floating-point multiplication, division and square root algorithms [9]. We discuss this and other related work after presenting our results.

Formally verifying IEEE compliance of floating-point hardware presents three major challenges. The first challenge is to capture the sense of the IEEE specification in a straightforward and formal way, such that it can be presented and debated in a verification review. For example, the IEEE standard mandates that "when rounding toward $-\infty$ the result shall be the format's value … closest to and no greater than the infinitely precise result" [2]. By introducing appropriate definitions and abstractions we can state this requirement formally in terms of the rounded result R, infinite precision result V, and the smallest representable increment ulp+:

* All other trademarks are the property of their respective owners.

$$round(toNegInf, R, V) = (R \le V) \wedge (V < R + ulp^{+})$$

Note that writing IEEE-level specifications requires us to be able to specify arithmetic operations of unbounded precision, and that specification notations that support only finite-precision "bit vector" arithmetic are therefore inadequate.

The second challenge arises because the precision of the arithmetic operations implemented by the hardware is inherently bounded. A key aspect of our proof is the verification that the finite-precision computations performed by the hardware correctly implement the unbounded precision operations mandated by the specification. The gap between the finite-precision gate-level description and the unbounded-precision high-level specifications is spanned in the first instance by our word-level model checker, using *hybrid decision diagrams* (HDDs) [4]. At higher levels of abstraction, we provide specialized decision procedures that facilitate reasoning about unbounded precision specifications.

The third challenge is the sheer size and complexity of the floating-point algorithms and the hardware. We have implemented a framework that combines advanced model-checking technology, user-guided theorem-proving software, and decision procedures that facilitate arithmetic reasoning. Our approach has been to develop a lightweight theorem prover that is well suited to hardware reasoning and composing/decomposing model checking results. The result is an environment for developing concise and readable high-level arithmetic proofs. Combining theorem proving and model checking extends the capacity of our verification tools far beyond what is feasible by model checking alone.

This paper is organized as follows. We present a brief overview of our formal verification framework. We then explain in detail the specification and verification of the FMUL instruction to illustrate our methods. The remaining instructions are covered in less detail. FADD/FSUB and miscellaneous operations are verified against algorithmic models, which are verified in turn against IEEE-level specifications using the FMUL methodology. FSQRT, FDIV, and FPREM present special challenges because they are iterative algorithms. We explain how our basic methodology is extended to deal with iterative algorithms. The paper closes with a summary of results and a discussion of related work.

Technology Overview

The work we discuss in this paper was made possible by the formal verification system Forte, currently in development at Intel. Forte is an evolution of the Voss verification system, developed at the University of British Columbia [5]. It seamlessly integrates several types of model-checking engines with lightweight theorem proving and extensive debugging capabilities, creating a productive high-capacity formal verification environment.

There are two model-checking engines in Forte relevant to this work: a checker based on symbolic trajectory evaluation and a word-level model checker. Symbolic trajectory evaluation (STE) is a formal verification method that can be viewed as something of a hybrid between a symbolic model checker and a symbolic simulator [6]. It allows the user to automatically check the validity of a formula in a simple temporal logic but it performs the checking by using a symbolic simulation-based approach and thus is significantly more efficient than more traditional symbolic model-checking approaches. Symbolic trajectory evaluation is particularly well suited to handle datapath properties, and it is used in this work to verify gate-level models against more abstract reference models. However, since symbolic trajectory evaluation is based on binary decision diagrams [7], there are circuit structures that cannot be handled. In particular, multipliers are beyond the capability of STE alone.

The word-level model checker is a linear time logic model checker tailored specifically to verifying arithmetic circuits. By using hybrid decision diagrams, the model checker can handle circuits that BDD-based model checkers cannot handle [4]. In particular, it can handle large multipliers efficiently, satisfying one of the requirements for verifying any modern floating-point unit. Although the model-checking algorithms used by the word-level model checker differ from those used by STE, the word-level model checker uses the STE engine to extract a transition relation from the circuit, ensuring that the two model checkers use a consistent view of the circuit's behavior.

Today, neither model checker is capable of checking the high-level IEEE specifications against the gate-level design. As a result, the verification must be broken up into smaller pieces, either following the structure of the circuit or partitioning the input data space. In order to ensure that no mistakes are made in this partitioning process and that no verification conditions are forgotten, a mechanically checked proof is needed. In Forte, a lightweight theorem proving system is used for this task. This theorem prover is seamlessly and tightly integrated with the model checkers. As a result, no translation or re-formulation of the verification results is needed before the theorem proving can be performed. In addition, some special purpose decision procedures are also integrated, making reasoning about arithmetic results significantly easier and more efficient.

Although the theorem prover in the Forte environment can be used to reason about almost any type of object, its design is tailored specifically to combining model-checking results and reasoning about integer expressions. As a result, our high-level specifications are given in terms of integers, rather than rational numbers. As we show in a later section, this is more of an aesthetic than a real limitation.

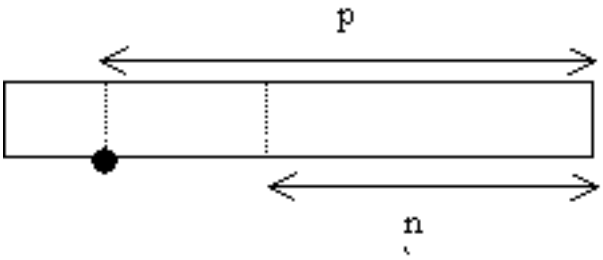

Figure 1: Mantissa representation

Finally, in order to make the verification process practical and efficient, the Forte verification system also includes significant support for efficient debugging. Support ranges from visualization support to provision of extensive counter example generation integrated in all decision procedures.

Specifying IEEE Compliance of Hardware

A floating-point number is made up of a sign $s \in \{-1,1\}$, a mantissa $m \geq 0$, and an exponent $e \geq 0$. Each floating-point number represents the rational number

$$R = \frac{s * m * 2^{e}}{2^{p} * 2^{bias}}$$

where p represents the number of bits in the fractional portion of the mantissa, and is a constant chosen to make e's range non-negative. Figure 1 depicts our representation of the mantissa. For a given R, we define ulp^{+} and ulp^{-} as the distance from R to the next and previous representable values, and ulp as the distance from R to the next largest representable value in magnitude.

$$ulp^{+} = \frac{B^{+}}{2^{p} * 2^{bias}} \quad ulp^{-} = \frac{B^{-}}{2^{p} * 2^{bias}} \quad ulp = \frac{2^{n} * 2^{e}}{2^{p} * 2^{bias}}$$

For most values of R, $B^{+} = B^{-} = 2^{n} * 2^{e}$ where p-n is the number of significant bits in the fractional part of the mantissa (p - n = 23 for single precision, 52 for double precision, and 64 for extended precision). When R's mantissa is 1.0, the adjacent representable values on either side of R have different exponents. In such cases $B^{+} = 2^{n} * 2^{e-1}$ and $B^{-} = 2^{n} * 2^{e}$ when R is negative; and $B^{+} = 2^{n} * 2^{e}$ and $B^{-} = 2^{n} * 2^{e-1}$ when R is positive. The bottom n bits in the mantissa are all 0.

The IEEE standard specifies that the result of a floating-point operation is computed as if the operation has been performed to unbounded precision and then rounded to fit into the destination format. The result of rounding a floating-point number is one of the two representable values closest to the exact value; which one of the closest values is determined based on the rounding mode. The IEEE standard specifies four rounding modes: toward zero, toward negative infinity, toward positive infinity, and to nearest. Since the IEEE standard is written in English, the formalization we present in this paper is necessarily our interpretation of the standard's intent.

We capture the required relationship between a result V calculated to unbounded precision and a result R rounded toward negative infinity

$$\text{round}(\text{toNegInf}, R, V) = (R \leq V) \wedge (V < R + ulp^{+})$$

The relation between a result V calculated to unbounded precision and a rounded result R is specified as follows for each of the three remaining rounding modes:

$$\text{round}(\text{toPosInf}, R, V) = (R - ulp^{-} < V) \wedge (V \leq R)$$

$$\text{round}(\text{toZero}, R, V) = (|R| \leq |V|) \wedge (|V| < |R| + ulp)$$

$$\text{round}(\text{toNearest}, R, V) = (R - \tfrac{1}{2} ulp^{-} \leq V) \wedge (V \leq R + \tfrac{1}{2} ulp^{+}) \wedge ((R - \tfrac{1}{2} ulp^{-} = V) \vee (V = R + \tfrac{1}{2} ulp^{+}) \Rightarrow \text{even}(R))$$

Even(R) means that the least significant bit in R's mantissa is 0, and it depends on the destination precision of the floating-point operation.

Specifying FMUL

To illustrate our methods, we will derive the specification for floating-point multiplication.

With the definitions of the previous section in hand, capturing the IEEE-level specification is easy. The product of two floating-point quantities, to unbounded precision, is

$$V_{FMUL} = \frac{s_1 * m_1 * 2^{e_1}}{2^{p} * 2^{bias}} * \frac{s_2 * m_2 * 2^{e_2}}{2^{p} * 2^{bias}}$$

Let us call the result of the computation

$$R_{FMUL} = \frac{s * m * 2^e}{2^p * 2^{bias}}$$

The units in the last place are as defined in the previous section, and the IEEE-level specification is

$$round(rndMode, R_{FMUL}, V_{FMUL})$$

where $rndMode \in \{toZero, toNegInf, toPosInf, toNearest\}$.

Since our verification environment only deals with specifications written in terms of integer operations, we now need to transform the IEEE-level specification developed above into an equivalent specification involving only operations on integers. Expanding the definitions of V, R, and ulp^+ in the IEEE specification for multiplication rounding to $-\infty$ gives

$$round(toNegInf, R_{FMUL}, V_{FMUL}) =$$
$$\left(\frac{s*m*2^e}{2^p*2^{bias}} \le \frac{s_1*m_1*2^{e_1}}{2^p*2^{bias}} * \frac{s_2*m_2*2^{e_2}}{2^p*2^{bias}}\right) \wedge$$
$$\left(\frac{s_1*m_1*2^{e_1}}{2^p*2^{bias}} * \frac{s_2*m_2*2^{e_2}}{2^p*2^{bias}} < \frac{s*m*2^e}{2^p*2^{bias}} + \frac{B^+}{2^p*2^{bias}}\right)$$

Now, by multiplying both sides of each inequality by $(2^p * 2^{bias})^2$ and rearranging, we obtain an equivalent specification in terms of integer operations only:

$$round(toNegInf, R_{FMUL}, V_{FMUL}) =$$
$$\left((s*m*2^e)*2^{p+bias} \le V'\right) \wedge$$
$$\left(V' < (s*m*2^e + B^+)*2^{p+bias}\right)$$

where $V' = (s_1 * m_1 * 2^{e_1}) * (s_2 * m_2 * 2^{e_2})$.

For the other rounding modes, we follow a similar procedure and obtain the following specifications:

$$round(toPosInf, R_{FMUL}, V_{FMUL}) =$$
$$\left((s*m*2^e - B^-)*2^{p+bias} < V'\right) \wedge$$
$$\left(V' \le (s*m*2^e)*2^{p+bias}\right)$$
$$round(toZero, R_{FMUL}, V_{FMUL}) =$$
$$\left(\left|s*m*2^e\right|*2^{p+bias} \le |V'|\right) \wedge$$
$$\left(|V'| < \left(\left|s*m*2^e\right| + 2^n * 2^e\right)*2^{p+bias}\right)$$
$$round(toNearest, R_{FMUL}, V_{FMUL}) =$$
$$\left((s*m*2^e - \tfrac{1}{2}B^-)*2^{p+bias} \le V'\right) \wedge$$
$$\left(V' \le (s*m*2^e + \tfrac{1}{2}B^+)*2^{p+bias}\right) \wedge$$
$$\left(\begin{array}{l}((s*m*2^e - \frac{1}{2}B^-)*2^{p+bias} = V') \vee \\ \quad (V' = (s*m*2^e + \frac{1}{2}B^+)*2^{p+bias}) \\ \quad \Rightarrow (m \% 2^n = 0)\end{array}\right)$$

These specifications are all in terms of the integer operations addition, subtraction, multiplication, and modulus and are therefore amenable to word-level model checking.

Verifying FMUL

The IEEE-level specifications are much too large and abstract to be directly verified by model checking, so the first step in our verification is to decompose the specification into smaller, more concrete pieces. The decomposition is done manually using engineering judgement about what is feasible to verify with the model checker. The second step is to verify the lower-level specifications by model checking. Typically, several iterations through the decomposition and model-checking steps are required. The third step is to formally compose the lower-level specifications using the theorem prover. The composition should yield back the desired high-level specification, thus verifying the correctness of the decomposition step.

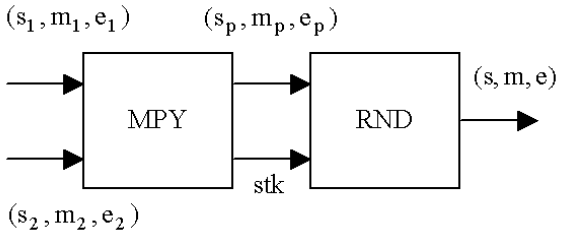

Figure 2: Floating point multiplier

Figure 2 shows a block diagram of the basic floating-point multiplication algorithm. In the MPY stage, inputs (s_1, m_1, e_1) and (s_2, m_2, e_2) are multiplied to compute an approximate result (s_p, m_p, e_p) without regard to rounding. Algorithmically, this means that their mantissas are multiplied, their exponents are added, and their sign bits are XORed. The output of the multiplication stage is a truncated form of the unbounded-precision result. A sticky bit is also generated, which is set if any significance was lost in the truncation. The intermediate result and sticky bit are sent to the rounder, which rounds according to the rounding mode and destination format.

The verification task was broken into multiplication and rounding stages, following the natural structure of the hardware. The multiplication stage was further decomposed into operations on the mantissa, exponent, and sign. Properties at this level were verified by model checking, and the results combined to yield the required high-level result.

First consider the multiplication stage. Using word-

level model checking, we verify the following properties of the mantissa, exponent, and sign computations:

$$m_p \le m_1 * m_2$$
$$m_p + 1 > m_1 * m_2$$
$$e_p = e_1 + e_2 + p + \text{bias}$$
$$s_p = s_1 * s_2$$

We also verify that the sticky bit correctly captures the loss of precision in multiplication.

$$stk = (m_p < m_1 * m_2)$$

We combine the above results using the theorem prover to obtain

$$\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| \le \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|$$
$$\left|s_p * (m_p + 1) * 2^{e_p + p + \text{bias}}\right| > \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|$$
$$stk = \left(\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| < \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|\right)$$

This completes verification of the multiplication part of FMUL.

Verification of the rounding property, like the property of multiplication, required proving some lower-level properties using word-level model checking and manipulating them using the theorem prover. For the rounding calculation, we verified the high-level property

$$\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| \le \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right| \wedge$$
$$\left|s_p * (m_p + 1) * 2^{e_p + p + \text{bias}}\right| > \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right| \wedge$$
$$stk = \left(\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| < \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|\right) \text{ for}$$
$$\Rightarrow$$
$$\text{round}(\text{rndMode}, R, V)$$
$$\text{rndMode} \in \{\text{toZero}, \text{toNegInf}, \text{toPosInf}, \text{toNearest}\}.$$

To complete the verification, we compose the properties of the multiplier and rounder to yield the required high-level property. This composition is performed in the theorem prover, using the logical rule that states "if we have proven both A and $A \Rightarrow B$, , then we can conclude B." In this case A is the property proved of the multiplier (which is also the antecedent of the rounder property) and $B = \text{round}(\text{rndMode}, R, V)$ is the desired property of the multiplier combined with the rounder. It is clear that B follows immediately.

Verifying FADD/FSUB and Miscellaneous Operations

The verification strategy for FMUL was a *structural decomposition* driven by the structure of the gate-level design and by capacity limitations of the model checker. For a large class of operations, which includes floating-point add and subtract, normalize-and-round, conversion to and from integers, and various shift instructions, model checking capacity suffices to perform *black box* verification. The strategy deployed here is to build an abstract *reference model* against which the gate-level design is verified. In a separate step, we can verify the reference model against IEEE-level properties[1]. As an example, Figure 3 shows the reference model for an operation used internally by the Pentium® Pro processor. Note that this is an algorithmic description that takes two Boolean vectors (S1 and S2) and returns a Boolean vector with an appropriately shifted mantissa.

```
let SHR_SPEC S1 S2 =
  let diff =
    exp S1 '<' exp S2 => exp S2 '-' exp S1
    | exp S1 '-' exp S2 in
  let shiftv = diff '< max => diff | max in
  let man = shr (man S2) shiftv in
  let exp = man '= 0 => 0 | exp S2 in
  (sgn S2)@exp@man;
```

Figure 3: Reference model for FPSHR

The IEEE-level specification of FADD/FSUB is similar to that for FMUL. For FADD/FSUB, we used as a reference model a textbook algorithm for addition/subtraction with rounding [8]. The gate-level design was verified against the reference model using symbolic trajectory evaluation. The verification of the gate-level design against the reference model is black box in the sense that no decomposition is needed of the reference model or the design. The reference model (but not the gate-level model) performs addition in five stages: mantissa alignment (so that the exponents are equal), addition/subtraction of the mantissas, normalization, rounding, and renormalization (as rounding may produce an overflow). The proof that the reference model complies to IEEE specifications naturally decomposes into the same stages. For each stage, the appropriate properties were proved (using word-level model checking), and were finally combined using the

[1] Many of the miscellaneous operations are used internally by the processor and lack meaningful IEEE-level specifications. For these operations the second step is unnecessary.

Forte theorem prover.

While the verification of the reference model against the high-level specification required structural decomposition, the decomposition was determined by what was most convenient for theorem proving and not by constraints imposed by the gate-level design or the model checker. The advantage of this approach is that the most time-consuming and complex part of the verification, the use of theorem proving in verifying the high-level specification, is completely isolated from the relatively fast-changing gate-level design. As a case in point, we ported the FADD/FSUB reference models developed for the Pentium Pro processor to two other processors and verified them with only very minor changes that did not impact the high-level proofs. The high-level proofs therefore remain valid for the other processors.

Verifying FSQRT and FDIV

Division, square root, and remainder present special challenges. They are both more difficult to specify and more difficult to verify than FMUL and FADD/FSUB.

The difficulty in specification is minor, and it arises because our verification framework requires that specifications be stated in terms of integer operations. However, FDIV naturally yields a rational result, and FSQRT may well yield an irrational result. Following the method used for FMUL, we define

$$V_{FDIV} = \frac{s_2 * m_2 * 2^{e_2}}{2^p * 2^{bias}} \Big/ \frac{s_1 * m_1 * 2^{e_1}}{2^p * 2^{bias}}$$

$$R_{FDIV} = \frac{s * m * 2^e}{2^p * 2^{bias}}$$

Simple algebra gives

$$V_{FDIV} = \frac{s_2 * m_2 * 2^{e_2}}{s_1 * m_1 * 2^{e_1}}$$

Since $s_1, s_2 \in \{-1,1\}$, it is true that $s_2 / s_1 = s_1 * s_2$. Therefore the above definition can be rewritten as follows (the utility of this step will be explained in a moment):

$$V_{FDIV} = \frac{s_1 * s_2 * m_2 * 2^{e_2}}{m_1 * 2^{e_1}}$$

We can then derive the following specification, in terms of integer operations, for division rounding to negative infinity:

$$\begin{aligned} &\text{round}(\text{toNegInf}, R_{FDIV}, V_{FDIV}) = \\ &\quad \left(s * m * 2^e * m_1 * 2^{e_1} \le s_1 * s_2 * m_2 * 2^{e_2} * 2^{p+bias}\right) \wedge \\ &\quad \left(s_1 * s_2 * m_2 * 2^{e_2} * 2^{p+bias} < \left(s * m * 2^e + B^+\right) * m_1 * 2^{e_1}\right) \end{aligned}$$

Our derivation involved multiplying both sides of each inequality by $m_1 * 2^{e_1}$. If we had multiplied by $s_1 * m_1 * 2^{e_1}$ we would have had to break our final specification into cases according to the sign of s_1. Rearranging V_{FDIV} using the fact $s_2 / s_1 = s_1 * s_2$ allows our final specification to remain simple. Specifications for division in other rounding modes are also easy to derive.

For FSQRT, we define

$$V_{FSQRT} = \sqrt{\frac{s_2 * m_2 * 2^{e_2}}{2^p * 2^{bias}}}$$

$$R_{FSQRT} = \frac{s * m * 2^e}{2^p * 2^{bias}}$$

We derive the specifications for square root by squaring both sides of each inequality and rearranging them. For example, the specification for square root rounding to negative infinity is

$$\begin{aligned} &\text{round}(\text{toNegInf}, R_{FSQRT}, V_{FSQRT}) = \\ &\quad \left(\left(s * m * 2^e\right)^2 \le \left(s_2 * m_2 * 2^{e_2}\right) * 2^{p+bias}\right) \wedge \\ &\quad \left(\left(s_2 * m_2 * 2^{e_2}\right) * 2^{p+bias} < \left(s * m * 2^e + B^+\right)^2\right) \end{aligned}$$

The FPREM (floating-point remainder) instruction is a variant of FDIV and will not be discussed further.

FDIV, FSQRT and FPREM are realized in the Pentium® Pro processor by iterative algorithms, and the complexity of the algorithms makes them difficult to verify. Verification of these operations required verification of a loop invariant plus some special properties of the initial and final iterations. These results were then combined with properties of the rounding algorithm.

First, we need to verify the correctness of the control flow, shown in Figure 4. Suppose the computation takes n iterations. We must prove that the circuit is in its initial state in cycle 0, in its loop state at cycles 1 through n, in the final state of the FDIV or FSQRT computation in cycle n+1, and rounding the result in cycle n+2. These properties are verified by induction over time. The induction base says that at cycle 0, the circuit is in the initial state, the loop counter is n, and

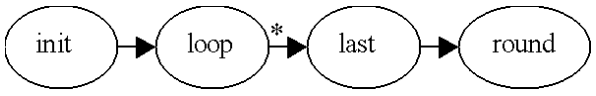

Figure 4: FDIV/FSQRT/FPREM algorithm flow

the next state is the loop state. The induction step says if the circuit is in the loop state and the loop counter is greater than 0, then the next state is the loop state and the counter decreases by 1. Furthermore, if the circuit is in the loop state and the loop counter is 0, then the next state is the final state, and the result will be rounded after the final state. All of the induction base and induction steps are proved by model checking on the circuit. The theorem prover is used to compose these results to verify the correctness of the control flow.

The second step is to prove the data invariants. There are two data invariants for the iterative operations. The first invariant is the range invariant, which relates the values of the partial remainder and divisor (or radicand) in an iteration. The second invariant is the value invariant, which relates the values of the partial remainder, divisor, dividend, and quotient (or partial remainder, radicand, and root) in adjacent iterations. The range and value invariants are proved using induction over time. The base cases and the induction steps are proved by model checking. The theorem prover is used to combine base and induction steps to complete the invariant proofs.

After the second step is done, the results are combined with properties of the rounding stage to reach the required high-level specification. This verification step is very similar to the final verification step for FMUL.

Results on the Pentium® Pro Processor

The preceding sections have given an overview of our verification methodology and its application to various classes of operations. In this section we describe the results of verifying the Pentium® Pro processor's floating-point execution unit (FEU).

This work has its roots in earlier Intel work on arithmetic circuit verification, which focused on specifying and verifying core datapath functionality [3]. Our goals for the current work were much broader in scope. Our aims for the specification and verification effort were as follows:

1. Construct IEEE-level functional specifications for all floating-point operations performed by the FEU.
2. Verify correct datapath functionality for all operations specified.
3. Verify that flags are correctly set and faults are correctly signaled, as required by the microarchitecture specification.
4. Cover all variants of each operation, including those used only by microcode.
5. Cover all precisions and rounding modes.

We accomplished these aims during a five-quarter project involving two engineers full-time, with one additional full-time engineer involved in the fifth quarter. The first two quarters were devoted mostly to tool and methodology development, driven by example operations drawn from the Pentium Pro processor. In this initial period, our word-level model checker, theorem-proving software, and decision procedures were refined, and inference rules were developed to reason about properties proved by model checking. In the final three quarters, our resources were applied to verification in earnest, with work on tools and infrastructure only as needed. In all, we undertook six verification sub-tasks: FADD/FSUB (we consider this one operation), FMUL, FDIV, FSQRT, FPREM, and numerous miscellaneous operations (each is relatively simple, but there are many of them). Each sub-task took approximately one engineer-quarter[2], including specification development, understanding the FEU micro architecture and gate-level design, model checking, and theorem proving.

To limit the scope of the project we chose to focus on the numeric correctness of the FEU. Some important correctness properties are not covered by our verification, including freedom from interference or contention between multiple active operations, the correct response of the FEU to external control events (stalls, for example), and the correctness of the microcode that uses the FEU operations.

A further goal of our Pentium Pro processor verification effort was that our results and methodologies should be reusable within and between major processor generations. To demonstrate the potential for such reusability, we repeated some of our verifications on other Intel® processors. We repeated the model checking of FADD/FSUB and miscellaneous operations on a proliferation of the Pentium Pro processor, and we verified that the proliferation's behavior was identical with respect to our specifications (despite some redesign that had occurred). We observed virtually complete reuse of our properties and verifica-

[1]This figure includes time for staff meetings and other routine activities.

tion scripts. We repeated verification of FADD/FSUB on a new processor generation that has a completely different microarchitecture and, again, obtained extremely high re-use of properties and scripts. The highly reusable nature of the FADD/FSUB verification is a consequence of our capability to perform black box model checking on these operations. We also ported our FMUL verification to the new processor. Since FMUL was verified using a structural decomposition approach, microarchitectural changes between processor generations dictated that many of the low-level properties used in the FMUL verification had to be modified. Our tools and methodologies proved fully reusable, however. One very high-quality erratum was discovered and corrected in the pre-silicon design phase.

Discussion

The success of this project has three principal lessons. The first is that we can formally verify gate-level descriptions of floating-point hardware against IEEE-level specifications using the Forte formal verification framework. Moreover, it is feasible and practical to do so in the timeframe of a major processor design project. The major impact of our work is on the quality of the product: our method will eliminate a class of post-silicon escapes due to incorrect floating-point functionality. A second impact is on validation efficiency: scarce simulation cycles can be redirected to cover functionality that is not covered by formal verification.

The second lesson is that verifying floating-point hardware against IEEE-level specifications requires significant effort and investment. Our verification effort on the Pentium® Pro processor involved two formal verification experts full-time over five quarters, with a third full-time expert added in the final quarter. It is true that part of this time was spent on tool and methodology development, but it is also true that we were able to leverage prior knowledge about the Pentium Pro microarchitecture. We expect that an effort on a future processor will require a similar investment. On the other hand, the payback is potentially very high. Each floating-point erratum has the potential to be a highly visible flaw in the processor, as was the case with the Pentium® processor's FDIV flaw.

A third lesson is that devising and executing floating-point formal verification strategies requires certain specialized expertise. Identifying low-level properties or writing reference models requires solid knowledge of floating-point microarchitecture and algorithms. Relating the low-level properties to IEEE-level specifications requires skill in constructing formal proofs, and mechanizing the verification to the extent described in this paper requires skill and experience in using theorem-proving software. The difficulties encountered in model checking range from moderate (FADD/FSUB and miscellaneous operations) to very challenging (FMUL). While the required expertise can be readily taught and learned, it is not yet a part of the mainstream curriculum in computer science or electrical engineering.

Most formal verification methodologies in use within industry today focus on using model checkers to verify low-level properties of hardware. A notable exception in the field of floating-point verification is the work of Russinoff on verifying the AMD-K7 * algorithms using the ACL2 theorem prover [9]. Russinoff formally verified the behavior of abstract models of the AMD-K7 floating-point algorithms. Russinoff's abstract models were "derived from an executable model that was written in C and used for preliminary testing" and assumed integer addition, multiplication, and bitwise logical operations as primitives [9]. Russinoff does not specify how he validates the correctness of the abstract models with respect to the gate-level design. In contrast, we verified the IEEE-compliance of the gate-level design from which the silicon is derived and which is used for all pre-silicon dynamic validation. Another important difference between Russinoff's work and ours is that the AMD-K7 verification was carried out in a general-purpose theorem prover whereas our verification methodology combines model checking with a theorem prover that is specialized to manipulate model checking results.

A precursor to the current work is that of Aagaard and Seger, who formally verified the IEEE compliance of the gate-level implementation of a floating-point multiplier using a combination of theorem proving and model checking [10]. The verification we describe in the current paper covers many more operations, and we have demonstrated that our work scales to industrial-sized floating-point hardware.

There has been substantial work on formal verification of floating-point algorithms implemented in either microcode or software. The microcode that implements the floating-point divide instruction of the AMD-K5* processor was verified by Moore, Lynch, and Kaufmann [11]. Russinoff verified the microcode for the AMD-K5 floating-point square root instruction [12]. Harrison describes the specification and verification of the exponential function [13]. Cornea-Hasegan describes the verification of programs that compute IEEE-compliant division, remainder, and

* All other trademarks are the property of their respective owners.

square root operations [14]. All these verifications assume, as a starting point, that the underlying hardware correctly implements arithmetic building blocks such as addition, subtraction, and multiplication. Our work is complementary in that it focuses on verifying the correctness of the underlying hardware. The algorithm verifications have often relied on relatively deep mathematics, and therefore are most appropriately formalized in a general-purpose theorem-proving environment.

Conclusion

We have developed tools and methods for formal verification of floating-point functionality and demonstrated them on the Pentium® Pro processor and other processors. No other formal verification method has been shown to be able to fully span the semantic gap between gate-level descriptions and high-level IEEE specifications in this domain.

Acknowledgments

We are grateful to John Harrison for pointing out an error in an earlier version of our specifications. We thank Yirng-An Chen for helping in the development of the word-level model checker and some early verification work, and Mark Aagaard and Donald Syme for helping in development of the theorem prover. Tom Melham and Robert Jones made key contributions to the methodology we used to verify FADD/FSUB and miscellaneous operations. We are grateful to Timothy Kam, Yatin Hoskote, and Pei-Hsin Ho for their contributions to our earlier verification effort.

References

[1] Intel® Architecture Software Developer's Manual. Volume 2: Instruction Set Reference, Intel Corporation, 1997, Order Number 243191.

[2] "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.

[3] Y-A. Chen, E. Clarke, P-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking," in M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*, Volume 1166 of *Lecture Notes in Computer Science,* Springer-Verlag, 1996.

[4] E.M. Clarke, M. Khaira, and X. Zhao, "Word-level symbolic model checking: a new approach for verifying arithmetic circuits," in *Proceedings of the 33rd ACM/IEEE Design Automation Conference,* IEEE Computer Society Press, June 1996.

[5] C. Seger, *Voss—A Formal Hardware Verification System: User's Guide,* Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.

[6] C. Seger and R. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design,* 6(2), April 1994, pp. 147-189.

[7] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers,* 35(8), August 1986, pp. 677-691.

[8] J. Feldman and C. Retter, *Computer Architecture: A Designer's Text Based on a Generic RISC*, McGraw-Hill, 1994.

[9] D. Russinoff, "A mechanically checked proof of IEEE compliance of the floating-point multiplication, division, and square root algorithms of the AMD-K7* processor," *London Mathematical Society Journal of Computation and Mathematics,* 1, December 1998, pp. 148-200.

[10] M. Aagaard and C. Seger, "The formal verification of a pipelined double-precision IEEE floating-point multiplier," in *International Conference on Computer-Aided Design,* IEEE Computer Society Press, November 1995.

[11] J. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5 86* floating-point division program," *IEEE Transactions on Computers,* 47(9), September 1998, pp. 913-926.

[12] D. Russinoff, "A mechanically checked proof of correctness of the AMD-K5* floating point square root microcode," *Formal Methods in System Design,* 14(1), 1999, special issue on arithmetic circuits.

[13] J. Harrison, "Floating point verification in HOL Light: the exponential function," Technical Report 428, University of Cambridge Computer Laboratory, June 1997.

[14] M. Cornea-Hasegan, "Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms," *Intel Technology Journal,* Q2 1998 at http://developer.intel.com/technology/itj/q21-998.htm.

Authors' Biographies

John O'Leary is a senior engineer with Strategic CAD Labs, Intel Corporation, Hillsboro, OR. From 1987 to 1990, he was a member of the scientific staff at Bell-Northern Research, Ottawa, Canada. He joined Intel in 1995 after earning a Ph.D. in electrical engineering from Cornell University. His interests are formal hardware specification and verification. His e-mail is joleary@ichips.intel.com.

Xudong Zhao is a staff engineer with Strategic CAD Labs, Intel Corporation, Hillsboro, OR. He recieved his Ph.D. degree in computer science from Carnegie Mellon University in 1996. His research topics include formal verification for hardware, for arithmetic circuits in particular. His e-mail is xzhao@ichips.intel.com.

Rob Gerth is a staff engineer in the Strategic CAD Laboratories, Intel Corporation, Hillsboro, OR. He received his Ph.D in computer science from Utrecht University, The Netherlands in 1989 and was a lecturer at Eindhoven University of Technology before he joined Intel in 1997. His current interests include multi-processor verification. His e-mail is robgerth@ichips.intel.com.

Carl-Johan H. Seger received his Ph.D. degree in computer science from the University of Waterloo, Canada, in 1988. After two years as Research Associate at Carnegie Mellon University he became an Assistant Professor in the Department of Computer Science at the University of British Columbia, and in 1995 he became Associate Professor. He joined Intel's Strategic CAD Labs in 1995. His research interests are formal hardware verification and asynchronous circuits. He is the author of the Voss hardware verification system and is co-author of *Asynchronous Circuits* (Springer-Verlag, 1995). His e-mail is cseger@ichips.intel.com.