Formal Verification of the Pentium®4 Floating–Point Multiplier

Roope Kaivola and Naren Narasimhan

Intel Corporation, JF4-451, 2111 NE 25th Avenue, Hillsboro, OR 97124, USA

Abstract

We present the formal verification of the floating-point multiplier in the Intel IA-32 Pentium® 4 microprocessor. The verification is based on a combination of theorem-proving and BDD based model-checking tasks performed in a unified hardware verification environment. The tasks are tightly integrated to accomplish complete verification of the multiplier hardware coupled with the rounder logic. The approach does not rely on specialized representations like Binary Moment Diagrams or its variants.

1 Introduction

We describe a formal verification effort addressing the input-output correctness of the multiplier unit in the Intel IA-32 Pentium®*4* microprocessor with respect to its IEEE specification. The verification is based on a combination of theorem-proving and symbolic trajectory evaluation using binary decision diagrams (BDDs).

Traditional state-based approaches and model-checking techniques based on BDDs do not perform well on multiplier circuits. Bryant in [3] proved that BDDs for multipliers grow exponentially in size regardless of the variable order. Several alternate solutions have been proposed but have met with limited success [4, 5, 6, 15, 20]

Mechanical theorem proving of multiplier circuits provides an alternative to automatic model checking. However, techniques based purely on reasoning have seen limited application due to the effort and user ingenuity involved in verifying even a small to medium-sized design [14, 19]. Verification of industrial strength hardware have so far been limited to abstract RTL descriptions [21].

Complementing a theorem prover with a model checker allows mechanical verification of designs that are difficult or infeasible to verify using either approach on its own. Kurshan and Lamport [16] did some verification of large multipliers constructed by simple recursive procedures using COSPAN and the TLP theorem prover [7]. Aagaard and Seger reported a verification effort on a small-scale multiplier circuit [1] in the Voss [22] verification system. O'Leary et al. [17] verified IEEE compliance of floating-point hardware for the Pentium Pro processor using a combination of word-level model-checking and theorem proving.

Our verification approach handles all flavors of multiplication supported by the Pentium *4* hardware. In this paper we present the verification of the extended precision floating-point operation with IEEE rounding, flags and faults.

The verification was carried out in the Forte verification environment - a combined model-checking and theorem-proving system [10]. The interface language to Forte is FL, a lazy strongly-typed functional language in the ML family [18]. Model checking in Forte is done via symbolic trajectory evaluation (STE) [23]. Theorem proving is done in the ThmTac proof tool.

2 Proof Framework

We use a variant of the traditional pre-postcondition framework for formulating temporal aspects of our specification statements. One main reason for introducing the pre-postcondition framework was to enable reasoning about the flow of computation in a well-structured manner.

The theory of pre-postcondition triples is a standard framework for specification (see e.g. [8, 13]). In this approach, statements about programs are triples $\{P\}S\{Q\}$, where P and Q are logical properties, and S is a program. Such a triple formalizes the statement *precondition P guarantees postcondition Q after running S*, or more accurately *for any possible execution of program S, if the execution starts in a situation satisfying P, then it terminates in a finite time and leads to a situation satisfying Q.*

In STE, trajectory evaluation correctness statements are called *trajectory assertions* and are written as: $\models_{\mathsf{ckt}} [ant \Longrightarrow\!\!\gg cons]$. The *(ant)ecedent* gives an initial state and input stimuli to the circuit ckt, while the *(cons)equent* specifies the desired response of the circuit. Formally, the meaning of $\models_{\mathsf{ckt}} [ant \Longrightarrow\!\!\gg cons]$ is: all sequences that are

// A float f is the triple (f_s, f_e, f_m)
input: float: $S1$, $S2$, $-2^{(N*M)-1} \leq (S1_m, S2_m) < 2^{(N*M)-1}$
output: float: W
multiply ($S1_m, S2_m$)
$W_m := 0$; $i := 0$; $y := 0$;
$bS1_m := \mathbf{booth\_encode}(S1_m)$; /* encoding */
while $y < M$
do /* generate partial products */
$PP_y := S2_m * bS1_m[y]$; $y := y + 1$;
od
while $i < M$
do /* shift partial products and add */
// product mantissa
$W_m := (PP_i * 2^{N*i}) + W_m$; $i := i + 1$;
od

$W_s := S1_s \; xor \; S2_s$; // sign
$W_e := S1_e + S2_e$; // exponent
W_{rnd} := round (W); // rounded result

Figure 1. Multiplier High-level Algorithm

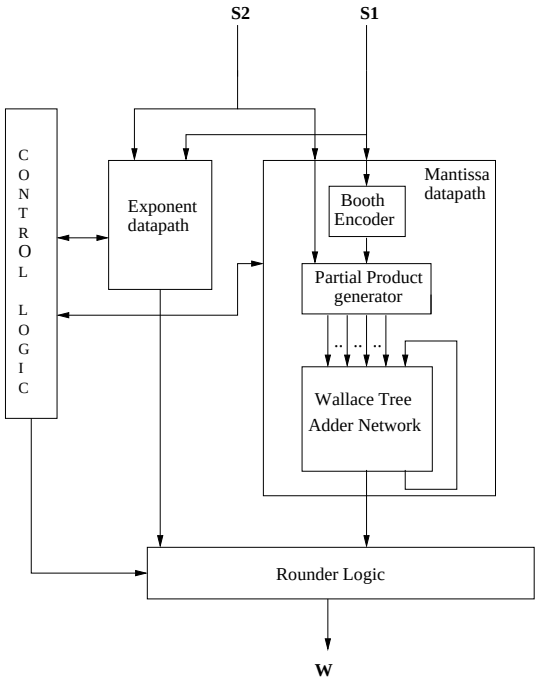

Figure 2. Multiplier Block Diagram

in the language of the circuit and that satisfy the *ant* will also satisfy the *cons*.

Consider a circuit ckt, and assume that a trajectory assertion $tr_{in}(x)$ binds a vector x of Booleans to some input signals of ckt at the times the inputs are read by the circuit, and that a vector y is similarly bound to some output signals by trajectory assertion $tr_{out}(y)$. If a formula $\phi_{in}(x)$ expresses the precondition the input is supposed to meet, and $\phi_{out}(x, y)$ the postcondition the circuit is supposed to produce, the statement *precondition* ϕ_{in} *guarantees postcondition* ϕ_{out} can be expressed by the formula

$$\begin{aligned} \forall in.\phi_{in}(in) \quad &\Rightarrow \quad (\exists out.(\models_{\text{ckt}} [tr_{in}(in) \Longrightarrow\!\!\gg tr_{out}(out)])) \\ &\wedge \quad (\forall out.(\models_{\text{ckt}} [tr_{in}(in) \Longrightarrow\!\!\gg tr_{out}(out)]) \\ &\quad\; \Rightarrow \phi_{out}(in, out)) \end{aligned}$$

we write $\{\phi_{in}\}(tr_{in}, ckt, tr_{out})\{\phi_{out}\}$ as a shorthand for the above formula.

Our proof framework also includes a library of formally verified bit-vector integer arithmetic operations in order to support reasoning of the model-checked results in a mathematical domain.

3 Multiplier Circuit

Figure 1 depicts a high–level view of the multiplication algorithm and Figure 2 a high-level block diagram of the Pentium *4* multiplier. The **multiply** function accepts the mantissas of the two source operands and generates the mantissa of the unrounded product. The sign and exponents of the product are calculated next and the the $round$ function is applied to the complete product. The source mantissas are assumed to be bound by a function of some constant M which is known a priori. The **booth_encode** function is a Radix-2^N modification of the classic Booth encoding scheme [2]. It accepts an n-bit multiplier operand $S1_{n-1}\, S1_{n-2}\, S1_{n-3}\, \ldots\, S1_1\, S1_0$ and generates a suitable encoding *bS1* by viewing *S1* as a combination of i slices and invoking the *Booth* function recursively on every slice.

$\mathbf{booth\_encode}(S1) =$ *Booth(M, S1), Booth (M-1, S1)...*
Booth (i, S1) $\equiv_{df}$ $(-2^{N-1} * S1_{(N*i)-1}) +$
$(\sum_{t=2}^{N} 2^{N-t} * S1_{(N*i)-t}) \;+\; S1_{N*(i-1)-1}$

For floating point multiplication, the mantissas of the multiplier and the multiplicand are the inputs to the multiplier circuit. The exponents are summed up separately. Depending on the rounding mode, the product mantissa and exponent are then rounded appropriately by the rounder logic to produce the final result.

The Wallace tree of the multiply unit is large enough to allow pipelined execution of all precision multiply operations except extended-precision, which in turn can only be executed every other clock cycle. Floating-point multiplication is a two pass operation. It requires doing multiplies with full precision (64x64) and since the Pentium *4* multiplier does not have a full 64x64 multiplier array, two passes are needed for full evaluation of all these multiplies. Therefore, the lower order bits of the multiplication result are generated in the first pass. This intermediate result is then fed

back into the adder network to be combined with the more significant partial products to generate the full product. For a given rounding mode and precision mode, this result is then rounded in the rounder logic to generate the final product of the mantissa. The exponents of the input operands are added and based on the range of the full product, are adjusted appropriately by a separate logic. For more details, please refer to [9].

4 Floating-Point Numbers and Rounding

Our proof framework includes a general-purpose definition and theorem library for floating-point numbers and rounding. The library supports floating-point numbers and rounding at the bit-vector level for model-checking, and at the mathematical level for reasoning. Furthermore, the mathematical level is divided into a bottom and top layer: the bottom layer deals with floating-point numbers as sign-exponent-mantissa triples, and the top layer deals with the real numbers encoded by the triples. We represent a binary floating-point number as a triple $p = (s, e, m)$ where s is the sign bit, e is the exponent bit-vector and m is the mantissa bit-vector. The real number $r(p)$ encoded by this triple is $(-1)^{\widehat{s}} * 2^{\widehat{e}-bias} * \widehat{m} * 2^{-manln+1}$, where $\widehat{x}$ is the integer encoded by the bit vector x, and $bias$ is some fixed *exponent bias*. Here the mantissa m has intuitively $manln - 1$ fraction bits and one bit to the left of the binary point, so m always encodes a value strictly less than 2. Currently our framework only supports integers. As a work-around, when converting a floating-point triple p to a number, we do not convert it to a real as in $r(p)$ above, but to an integer that corresponds to the real value multiplied by a big number, as follows: $ri(p) = (-1)^{\widehat{s}} * 2^{\widehat{e}-bias+BN} * \widehat{m}$, where $p = (s, e, m)$. The number BN is chosen to be sufficiently large so that every real number that is representable as a floating-point triple and every half-point of two such numbers maps to an integer when multiplied by 2^{BN}. In this case all the reasoning related to rounding can be carried out with integers.

For example, Figure 3 contains definitions for rounding to positive infinity for floating-point triples in the lower mathematical level. The bit-vector level definitions are obtained from these by replacing mathematical operations with bit-vector ones. Figure 4 contains definitions for rounding to positive infinity dealing with the numbers encoded by floating-point triples. A correspondence theorem between the levels states that $ri(\text{round\_pinf}_{fp}(\ r,\ p\ )) = \text{round\_pinf}_{ri}(\ r,\ ri(p)\ )$ for all normal floating-point triples p. A similar framework is in place for the other three IEEE rounding modes: round to zero, round to negative infinity and round to nearest.

5 Verification Overview

An intuitive specification for IEEE floating-point multiplication can be given as follows:

> *IF a floating-point multiplication operation is started AND the inputs to the circuit are $S1$ and $S2$ AND expected internal constraints to the circuit hold initially AND expected environment constraints hold throughout the execution of the operation, THEN at the time the circuit produces output W, the equation $\widehat{W} = round(\widehat{S1} * \widehat{S2})$ holds.*

$\widehat{x}$ denotes the arithmetic equivalent of the bit-vector x and the function $round$ captures the conceptual notion of rounding the product.

The details of the internal and environment constraints and the mechanism for starting an operation may be dependent on the implementation and the protocol it enjoys with the hardware around it. For instance, these constraints could include the behavior of reset, unit clock, when the operation is initiated, legal ranges for the input sources etc.

$$\text{prev}_{fp}(r,\ p) = (sgn(p),\ exp(p),\ man(p)/2^{len(p)-r},\ r)$$
$$\text{prop\_next}_{fp}(r,\ p) = (sgn(p),\ exp(p),\ man(p)/2^{len(p)-r} + 1,\ r)$$
$$\text{EQ}_{fp}(p,\ q) = (sgn(p) = sgn(q)) \wedge (exp(p) = exp(q)) \wedge (man(p) * 2^{len(q)} = man(q) * 2^{len(p)})$$
$$\text{is\_normal}(\ p\ ) = (2^{len(p)} \leq man(p)) \wedge (man(p) < 2^{len(p)+1})$$
$$\text{norm\_down}(p) = \text{is\_normal}(p) \Rightarrow p \mid (sgn(p), exp(p) + 1, man(p)/2, len(p));$$
$$\text{next}_{fp}(r,\ p) = (\text{EQ}_{fp}(p,\ \text{prev}_{fp}(r,\ p\ )) \Rightarrow \text{prev}_{fp}(r,\ p) \mid (\text{norm\_down}\ (\text{prop\_next}_{fp}(r,\ p))$$
$$\text{round\_pinf}_{fp}(r,\ p) = sgn(p) \Rightarrow \text{prev}_{fp}(r,\ p) \mid \text{next}_{fp}(r,\ p)$$

The argument r to the rounding function reflects the number of significant fraction bits after rounding. The notation $\ldots \Rightarrow \ldots \mid \ldots$ denotes the if-then-else construct. Note that $/, *, 2^n$ etc. are integer operations.

Figure 3. Round to Positive Infinity for Floating-point Triples

$$\text{prev}_{ri}(r,\ p) = (p/2^{\log_2(p)-r}) * 2^{\log_2(p)-r}$$
$$\text{prop\_next}_{ri}(r,\ p) = (p/2^{\log_2(p)-r} + 1) * 2^{\log_2(p)-r}$$
$$\text{next}_{ri}(r,\ p) = (p = \text{prev}_{ri}(r,\ p)) \Rightarrow \text{prev}_{ri}(r,\ p) \mid \text{prop\_next}_{ri}(r,\ p)$$
$$\text{round\_pinf}_{ri}(r,\ p) = (p < 0) \Rightarrow (-\text{prev}_{ri}(r,\ |p|)) \mid \text{next}_{ri}(r,\ |p|)$$

Figure 4. Round to Positive Infinity for Reals

The structure of the multiplication algorithm and the circuit suggests a natural decomposition: First verify separately the generation of each partial product, and then verify that the rounded summation of these partial products equals the output ie. If we verify for each partial product i that the following holds

$$\widehat{pp(i)} = \mathit{Booth}(i, \widehat{S1}) * \widehat{S2}$$

and that

$$\widehat{W} = round(r, P),$$

where, r is the precision mode and P is the arithmetic equivalent of a floating-point triple (s, e, m) with an additional mantissa fraction length l such that

$$\begin{aligned} s &= \mathit{sgn}(S1(i))\ \mathit{XOR}\ \mathit{sgn}(S2(i)) \\ e &= \mathit{exp}(\widehat{S1(i)}) + \mathit{exp}(\widehat{S2(i)}) - bias \\ m &= \Sigma_i \widehat{pp(i)}\ *\ 2^{N*i} \end{aligned}$$

where the summation is over all partial products, and the Booth-encoding is done to radix 2^N.

We decompose the verification tasks into four distinct steps:

A For each partial product, verify that the expected bit-vector relation holds between it and the input sources

B Verify that the expected bit-vector relation holds between the partial products and the rounded product at the output of the rounder logic.

C Show that the bit-vector relations in A and B imply the corresponding mathematical relations

D Show that the mathematical relations between the inputs and the partial products, and the relation between the partial products and the output imply the expected mathematical relation between the input and the output

Different kinds of reasoning are required for these steps: [A] and [B] involve model-checking, step [C] requires reasoning about the correspondence between bit-vector and mathematical relations, and step [D] applies reasoning concerning arithmetics and the flow of computation. Steps [C] and [D] are performed in the theorem proving domain using the ThmTac proof tool.

Verification of the actual multiplier hardware followed the above decomposition idea closely. Nevertheless, mapping the abstract decomposition idea into the actual hardware was not trivial. Due to efficiency considerations, a conceptually clear algorithm tends to become considerably more fudged as we get closer to its circuit details. For example, some computations may be either advanced or deferred from their logical place, abstract values may be represented by combinations of signals, and local optimizations may make it hard to follow the progress of computation in the circuit.

6 Verifying Floating-Point Multiplication

The overall input-output correctness relation for IEEE floating-point multiplication can be stated as follows:

$$\begin{aligned} \mathit{IOSPEC}(i, o) \equiv_{df}\ & \exists P.(\mathit{ri}(S1(i)) * \mathit{ri}(S2(i)) = P * 2^{BN}) \\ & \wedge (\mathit{ri}(W(o)) = \mathrm{round}_{ri}(r,\ P) \end{aligned}$$

$S1(i)$, $S2(i)$ and $W(o)$ are functions extracting slices corresponding to the individual input and output data values. r specifies the number of significant fraction bits in the rounded result, and round_{ri} depends on the intended rounding mode for the operation. The extra entity P in this definition, intuitively denoting the unrounded result, is needed because of the lack of real numbers in our current proof framework.

The top-level correctness predicate for the multiplier is formally stated as a pre-postcondition triple.

$$\{\mathit{INIT}\}(tr_{in}, ckt, tr_{out})\{\mathit{IOSPEC}\}$$

INIT represents the conjunction of the initial constraints on the input sources and the environment constraints on the trajectory that binds the control signals to the multiplier. The trajectory assertion $tr_{in}(i)$ binds $S1(i)$ and $S2(i)$ to input data signals, and a vector $ctl(i)$ of control values to relevant control signals at the time the operation is started, and $tr_{out}(o)$ binds $W(o)$ similarly to output signals at the time the output is ready. Intuitively $tr_{in}(i)$ and $tr_{out}(o)$ associate the input and output states i and o with particular signal values at particular times in the circuit. Informally, the statement specifies that the execution of the multiplier logic begun in any state satisfying the precondition predicate *INIT* would terminate in a state satisfying the postcondition predicate *IOSPEC*. The trajectory function tr_{in}, binds the input sources and control signals to symbolic boolean values at the precise instant when the multiplier circuit is expected to read them. Similarly, tr_{out} is the corresponding trajectory function on the multiplier output and binds it to symbolic boolean values at the time when the output is ready. The hardware, ckt includes the multiplier logic, the the rounder logic and associated control circuitry.

The first step in the verification (proof step [A]) establishes the correctness statement for the Booth encoder and the partial product generator.

$$\forall k, 0 \leq k < M.\ \{\mathit{INIT}\}(tr_{in}, ckt, tr_{pp_k})\{\mathit{PPSPEC}_k\} \quad (1)$$

Here M is the maximum number of partial products generated during the multiplication operation, and tr_{pp_k} binds values corresponding to partial product k to the relevant signals. The postcondition PPSPEC_k captures the specification of partial product PP_k as a function of the input sources:

$$\mathit{PPSPEC}_k(i, o) \equiv_{df}\ \widehat{pp_k(o)} = \mathit{Booth}(\widehat{S1(i)}, k) * \widehat{S2(i)}$$

We model-checked the M predicates in Formula (1) separately. No significant tool capacity issues were encountered during this exercise. The exercise cumulatively took just 20 minutes to run on a Pentium *4* system and consumed around 250 MBytes of memory[1] Using a library of verified transformation rules discussed in Section 2, equivalent arithmetic statements were derived from these model-checked results.

Next we model-checked a side relation involving the last partial product.

$$\{INIT\}(tr_{in}, ckt, tr_{pp_M})\{AUX\} \quad (2)$$
$$\text{where } AUX(i,o) \equiv_{df} LB <= \widehat{pp_M(o)} <= UB$$

where LB and UB are bounds for the potential values of the partial product. The relevance of this will become clear later in the proof flow.

We then establish the relation between the partial products and the rounded product at the output of the rounder logic (Proof Step [B]).

$$\{INIT \wedge AUX \wedge RANGE\}(tr_{pp}, ckt, tr_{out})\{ADDSPEC\} \quad (3)$$

$$ADDSPEC(i,o) \equiv_{df} EQ_{fp}(W(o), (\text{round}_{fp}(r,(s,e,m))))$$
$$where, \quad s = sgn(S1(i)) \; XOR \; sgn(S2(i))$$
$$e = exp(S1(i)) + exp(S2(i)) - bias$$
$$m = \sum_{k=1}^{M} \widehat{pp_k(i)} \; * \; 2^{N*(k-1)}$$

The relation given in Equation 3 is model-checked after disconnecting the signals in the hardware that logically map to the partial products from their fan-in. This has the effect of model-checking the relation against the portion of the multiplier that comprises the pipelined Adder network, the rounder logic, and the control circuitry. The precondition needs to be appropriately strengthened by *AUX* and *RANGE* in order to constrain the partial product vectors to legal values. We also know a priori that the final unrounded mantissa after multiplication is a floating-point number in the interval [1, 4). The precondition *RANGE* captures this notion by constraining the range of the product mantissa to be in the bounded interval $2^l \leq m < 2^{(l+2)}$. The verification of Relation 3 involves a fairly significant model-checking effort. You may recall from the circuit description in Section 3 that floating-point multiplication operations describe two iterations of the CSA Wallace network. The model-checking effort included the entire adder network plus the feedback logic and furthermore, subsumed the entire rounding logic as well. The verification of Relation 3 thus involved some 45 million BDD nodes and the model-checking took about 30 minutes on a 1.7 GHz Pentium *4* machine using about 1.5 GBytes of memory.

In our efforts to compose the above model-checked results to imply the overall correctness statement, we go through an intermediate step (Proof Step [C]) to transform the model-checked relations specified in the bit-vector domain using BDDs to the mathematical domain so that it is amenable to reasoning in a theorem–proving environment. We developed a library of bit-vector integer arithmetic operations to help us bridge this gap and formally verify the transformations to be sound.

As a preliminary sub-task in Proof Step [D], we combined the M different model-checked results given by relation (1) to imply the correctness of the entire partial product generation algorithm. We achieved this by applying a post-condition conjunction inference rule[2].

$$\{INIT\}(tr_{in}, ckt, tr_{pp})\{PPSPEC\} \quad (4)$$
$$PPSPEC \equiv_{df} \bigwedge_{k=1}^{M} PPSPEC_k$$

Verifying the above composite relation is beyond the capacity of model-checking but since it only requires reasoning about conjuncted results that have been individually model-checked, we were able to easily discharge the obligation in the theorem proving domain using ThmTac.

Next, we use a postcondition conjunction rule to combine relations 2 and 4

$$\{INIT\}(tr_{in}, ckt, tr_{pp})\{AUX \; \wedge \; PPSPEC\} \quad (5)$$

Formulas 3 and 5 with the help of an appropriate collection of lemmas ought to imply the high-level correctness statement. In fact, much of the resulting theorem proving effort is involved in stating and verifying lemmas and connect them up appropriately in order to discharge the top level theorem. You may recall that in verifying Formula 3, we had to disconnect the circuit at the partial product generation stage. The theorem proving effort essentially serves to "stitch together" the apparently disjoint relations by connecting the partial products back up to the input sources. We use ThmTac to help us bridge this gap and ultimately establish a condition that relates the unrounded product, P (see definition in Section 5) to the input sources. We formally state this requirement as

$$(INIT \; \wedge \; PPSPEC) \; \Rightarrow \; MULT \quad (6)$$

where *MULT* is the equality relation $\widehat{P} = \widehat{S2} * \widehat{S1}$. From the definition of P, the mantissa component of the unrounded product m is defined in terms of the partial products. We would like to instead specify it directly in terms of the input sources and the relation given by Formula 6 is crucial to bridging this gap. The resulting proof is fairly involved and essentially boils down to verifying the correctness of the booth encoding algorithm.

$$PPSPEC \Rightarrow \Sigma_i \widehat{pp(i)} \; * \; 2^{N*i} = \widehat{S2} \; * \; \Sigma_i Booth(i, \widehat{S1})$$

[1] A significant portion of which is used to load the circuit alone

[2] For a general formulation of the inference rules, see [11]

From Formulas 4 and 6 and a pre-post transfer inference rule, we get

$$\{INIT\}(tr_{in}, ckt, tr_{pp})\{PPSPEC \wedge MULT\} \quad (7)$$

Based on Formulas 3 and 6, and using a pre-condition strengthening inference rule, we next establish

$$\{AUX \wedge RANGE \wedge PPSPEC \wedge MULT\} \ (tr_{pp}, ckt, tr_{out}) \ \{ADDSPEC\}$$

Using a pre-post condition transfer inference rule, we get

$$\{AUX \wedge RANGE \wedge PPSPEC \wedge MULT\} \ (tr_{pp}, ckt, tr_{out}) \ \{ADDSPEC \wedge MULT\} \quad (8)$$

We next establish that the correct range for the mantissa product is correctly implied by the partial product specification *PPSPEC* and the relation *MULT* that connects the partial product sum to the product of the source inputs.

$$(INIT \wedge PPSPEC \wedge MULT) \Rightarrow RANGE$$

With this, and the relations given by Formulas 5 and 7, we can now state the following

$$\{INIT\}(tr_{in}, ckt, tr_{pp}\} \ \{AUX \wedge RANGE \wedge PPSPEC \wedge MULT\} \quad (9)$$

With an additional statement that relates the three conditions *ADDSPEC*, *MULT* and *IOSPEC*, we are ready to compose the input-output correctness relation.

$$ADDSPEC \wedge MULT \Rightarrow IOSPEC \quad (10)$$

Here *IOSPEC* is the top-level IEEE specification for floating point multiplication. Observe that *ADDSPEC* is described with round$_{fp}$ while *IOSPEC* is defined in terms of the mathematical notion of rounding. The translation from round$_{fp}$ to round$_{ri}$, is achieved by applications of the rounding theorem library discussed in Section 4.

The input-output correctness relation follows quite easily from the Formulas 8,9 and 10.

$$\{INIT\}(tr_{in}, ckt, tr_{out})\{IOSPEC\} \quad (11)$$

In addition to the rounding specification, the functions round$_{fp}$ and round$_{ri}$ also incorporate the specification for flags and faults.

Relations specified by Formulas 1, 2 and 3 were established using the STE model-checker. The three separate model-checking runs cumulatively took about an hour to complete on a Pentium *4* machine. Composing the model-checked results and stating and verifying the rest of the relations was done using the ThmTac proof tool. A total of 86 theorems and lemmas were stated and verified during the proof development.

7 Discussion

The input-output statement captures the IEEE specification for floating-point multiplication. This includes checking for the correctness of the rounded result and the right generation of flags and faults. The specification assumes that the input sources are normal and denormals are handled in software.

The input-output specification for the multiplier checked for the correctness of the rounded result and the generation of flags and faults. The verification was completed for all 4 IEEE rounding modes and the 3 precision modes: single, double and extended. It took three person months to develop the complete proof framework and verify the Pentium *4* multiplier against its IEEE floating-point specification. The underlying libraries supporting bit-vector integer arithmetic operations and the theory of pre-postcondition triples were already in place at the time, and thus significantly reduced the proof development time.

The STE model-checking effort is BDD based. BDDs are well understood and there has been considerable activity in academic and industrial circles to research and develop better and more efficient BDD packages. Therefore it is conceivable that BDD model-checking techniques can benefit more readily from these advances than most other approaches [4, 17].

An observation about the model-checking effort stated in Proof Step [B] in Section 6. At the outset, we had limited our scope to verifying the relation between the partial products and the unrounded product at the output of the Adder network. We were then hoping to establish as a separate model-checking exercise, the relation between the unrounded product and the rounded result at the output of the rounding logic. However, this proved to be an extremely challenging task. The BDD representation for the sum and carry that make up the unrounded result was complicated and grew exponentially as the size of the multiplication operation. Including the rounding logic in the model-checking exercise much to our surprise, allowed for fairly compact BDD representations and we were able to quite easily establish rather straightforwardly the relation in Proof Step [B]. The benefits of this were twofold - the model-checking specification could now be expressed in a more natural way and secondly,much of the logic and the signal timing issues internal to the adder network and rounder logic could now be easily abstracted away. In contrast model-checking techniques based on BMDs in particular, cannot take advantage of this. The rounder logic cannot be efficiently represented using BMDs and therefore such techniques use a mix of BMDs and BDDs to complete the verification and subsequently the resulting specification tends to have its fair share of implementation details.

Underlying our verification methodology is the philo-

sophical belief that verifying hardware systems is an activity analogous to software programming. It involves organizing primitive proof steps, each of which can be mechanically verified, into a cohesive solution that establishes the intended high-level specification. Based on this view, our trust in being able to complete a verification task is based primarily on our decomposition skills and the robustness of the underlying primitive verification steps. To be sure, push-button automation has its place in this activity but it can only serve as a supplement to the above mentioned, when tackling a hard verification problem. For more discussion please see [12].

The success of our approach is particularly significant in an industrial setting. It is well known that success of any formal verification effort is particularly sensitive to the way hardware is described. Most work in multiplier verification that we come across, either involve verifying designs that have been tailored for verification or are at a sufficiently high level of abstraction so that much of the challenging problems arising due to design size and complexity are not even encountered. We started work on the Pentium *4* multiplier logic fairly late in the CPU design project after much of the design was already in place. As a result, we could not provide any input to the design team to make the RTL logic more verification friendly. The design was implemented in all its detail with performance criteria like throughput, area, power consumption etc. being the primary concerns. This compounded our verification problem. Decomposing our model-checking tasks purely based on functionality of the hardware was not feasible anymore since quite often logic related to a certain functionality was now spread across the design. Furthermore, we did not have the luxury of being able to modify the design when it severely stressed our verification tools. Instead, we had to investigate alternate approaches to specifying and verifying these portions of the design. Presence of complex testability logic, power saving mechanisms and close interaction with other hardware logic added to our verification woes. The multiplier circuit is in itself a challenging verification problem. The presence of these additional circuitry adds a significant dimension to our verification effort.

Our proof for floating-point multiplication includes the multiplier, the rounder logic and the associated control circuitry. We apply our proof technique to the actual hardware model that makes it to silicon, thus increasing our confidence in the correctness of the final product. The multiplier RTL implementation exceeds 5600 state elements and has approximately $250,000$ gates. The floating-point multiplier resides in a tightly integrated RTL environment that includes the divider, square root and the rounder logic. This description spans some $40,000$ HDL lines. In our experience, no single technique, either state-based model checking or theorem proving would be able to pragmatically handle such a large verification exercise. Verification claims based solely on either of these approaches appear to make limiting assumptions about the design or verify only an abstracted view, later synthesized to hardware.

Our verification approach is primarily characterized by three attributes: 1) Completeness, 2) Ease of re-use and 3) Scalability. The advantages of our approach are manifold. The input-output correctness relation is both succinct and complete, free from implementation details. This makes it easier to review the correctness of the specification and furthermore, by distancing ourselves from the design, we avoid mimicking its behavior in our proofs. Our approach ensures complete verification, in that we work on a representation of the actual circuit and our proof decomposition is guaranteed to be sound by the underlying deductive engine. With relatively minor effort, we have been able to apply our general proof framework to the verification of the entire family of multiplication operations performed by the Pentium *4* multiplier. Writing specifications at a high-level of abstraction permits us to do this. For instance, the 2 pass nature of the floating-point operation discussed in Section 3 and later in Section 6 had no significant impact on the verification and the same model-checking framework could be used to verify the other multiplication operations. None of the reasoning developed using the ThmTac proof tool involves any references to circuit details. The deductive framework is thus quite robust evidenced by the ease with which they have been ported to multiplier proofs for other Intel CPU design proliferations.

No report on a Formal verification approach is really complete without saying a few words on its bug detecting capabilities. We started this effort rather late in the project design cycle after a considerable amount of time and effort had already been invested in traditional simulation–based testing and other formal verification techniques. As a result, no additional errors surfaced during our verification effort. However, we have had a lot of success when we ported the entire proof framework to newer design proliferations and quickly isolated hard to detect corner cases that were being violated in the implementation.

Formal Verification proofs are notorious for getting out-of-date with designs changes and it usually requires some effort to keep them in synchronization with live industrial circuits. On the other hand, feasibility of any formal verification technique employed in an industrial setting, is primarily governed by its capacity to handle large and arbitrary complex hardware and the support it provides to proof development so as to resist obsolescence in the face of continual design changes. Our proof framework is robust enough to handle the frequent and non-trivial functional changes made to the Multiplier RTL hardware. One of the primary reasons for this is that the verification environment allows us to verify large portions of the hardware while making

sparing assumptions on the design environment and keeping our references to the internal logic to a minimum. Moreover, we crafted the specifications carefully so as to capture the intended behavior without paying much attention to the actual implementation.

8 Conclusion

We presented a formal verification case study of an industrial-strength floating-point multiplier circuit, using a combination of theorem-proving and BDD-based model-checking techniques. By performing the entire verification under a unified framework, we eliminate the possibility of introducing a whole class of errors that may arise from incorrect translation or unsound assumptions, into our proofs. We believe our approach has the added benefits of providing a sound framework for tight integration of the verification techniques and a mechanization for the formal proof of correctness of an industrial strength multiplier logic.

Acknowledgments

We would like to thank Tom Schubert and Bob Brennan for the opportunity to perform this work and Mark Bouler for many helpful discussions.

References

[1] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ICCAD*, pages 7–10. IEEE Comp. Soc. Press, Washington D.C. Nov. 1995.

[2] A. D. Booth. A signed binary multiplication technique. *Quart. J. of Mech. Appl. Math*, 4(2), 1951.

[3] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with applications to integer multiplication. *IEEE Trans. on Comp.*, C-40(2):205–213, Feb. 1991.

[4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *DAC*, pages 535–541. ACM Press, June 1995.

[5] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. In *ICCAD*, pages 159–163. IEEE Computer Society, 1995.

[6] 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 DAC*. IEEE Computer Society, June 1996.

[7] Engberg, U., Gronning, P., Lamport, L. Mechanical verification of concurrent systems with TLA. In *CAV*, LNCS, New York, June 1992. Springer-Verlag.

[8] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[9] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D, Kyker, A. and Roussel, P. The microarchitecture of the Pentium® 4 processor. *Intel Technology Journal*, Q1, Feb. 2001.

[10] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design and Test*, pages 16–25, July 2001.

[11] R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. In J. Harrison and M. Aagaard, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 338–355. Springer Verlag; New York, 2000.

[12] Kaivola, R. and Kohatsu, K. Proof engineering in the large: Formal verification of Pentium® 4 floating-point divider. In Margaria, T. and Melham, T., editor, *CHARME*, LNCS, pages 196–211, Scotland, UK, September 2001. Springer.

[13] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.

[14] Kapur, D., Subramaniam, M. Using an induction prover for verifying arithmetic circuits. *International Journal on Software Tools for Technology Transfer*, 3(1):32–65, 2000.

[15] Kimura, S. Residue BDD and its application to the verification of arithmetic circuits. In *DAC*, pages 542–545, San Francisco, CA, June 1995. IEEE Computer Society Press.

[16] Kurshan, R.P., Lamport, L. Verification of a multiplier: 64 bits and beyond. In Courcoubetis, editor, *CAV*, volume 697 of *LNCS*. Springer-Verlag, July 1993.

[17] J. W. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, Feb. 1999.

[18] L. Paulson. *ML for the Working Programmer,*. Cambridge University Press, 1996.

[19] Pierre, L. VHDL description and formal verification of systolic multipliers. In *CHDL*, N. Holland, 1993.

[20] Ravi, K et al. Modular verification of multipliers. In *FMCAD*, volume 1196 of *LNCS*, pages 49–63. Springer, 1996.

[21] 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. *London Mathematical Society Journal of Computational Mathematics*, 1:148–200, 1998.

[22] C.-J. Seger. Voss — A formal hardware verification system user's guide. Technical Report 93-45, Dept. of Comp. Sci., Univ. of British Columbia, 1993.

[23] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, Mar. 1995.