Proof Engineering in the Large: Formal Verification of Pentium®4 Floating-Point Divider

Roope Kaivola and Katherine Kohatsu

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

Abstract. We examine the challenges presented by large-scale formal verification of industrial-size circuits, based on our experiences in verifying the class of all micro-operations executing on the floating-point division and square root unit of the Intel IA-32 Pentium®4 microprocessor. The verification methodology is based on combining human-guided mechanised theorem-proving with low-level steps verified by fully automated model-checking. A key observation in the work is the need to explicitly address the issues of proof design and proof engineering, i.e. the process of creating proofs and the craft of structuring and formulating them, as concerns on their own right.

1 Introduction

Verification of large systems is discussed in an increasing number of published case studies. For many of these, the story-line may be paraphrased by *we used theory X and tool Y to verify system Z.* The verification of a system is considered an accomplishment on its own right, and the fact that it could be achieved at all is a contribution worth reporting. Given the current state of the art, we think this is quite justified.

Rather less has been said about the practice of applying formal verification on a large scale in a system development project [1,5,8]. Producing an isolated proof of correctness differs from such wide-scale application in the same way as writing a program to solve a single problem in a single set of circumstances differs from writing a general software system to solve a class of related problems in a variety of circumstances, evolving over time. Although the solution in both cases is likely to be fundamentally the same, the general case will require attention to issues that can be safely glossed over in the restricted case. In effect, when producing an isolated proof of correctness, the main concern is just that that the proof is provided, whereas in the general case, issues of how the proof is constructed and structured become equally important.

In this paper we examine some of the issues present in large-scale verification work, based on our experiences in verifying the family of all micro-operations executing in the division and square root unit of the Intel IA-32 Pentium®4 microprocessor. Although based on a single extended case study, we believe that many aspects of the work are of a more universal nature. Therefore, we have tried to phrase the discussion on the general level, drawing on the case study to illustrate various points in practice.

Our verification methodology is based on human-constructed, mechanically-checked proofs with completely automatically verified model-checking steps at the lowest level. The aim is to take advantage of automation to mechanise tedious low-level reasoning,

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 196–211, 2001.

while retaining the relatively complete freedom of the human verifier to set the overall verification strategy. We set out to perform a fully mechanically checked correctness proof in a single, unified framework, relating the high-level correctness statements all the way down to the actual register-transfer level description of the hardware. Technically the verification work was carried out in the the Forte verification framework, a combined model-checking and theorem-proving system built on top of the Voss system [9]. The interface language to Voss is `FL`, a strongly-typed functional language in the ML family [16], model checking is done via symbolic trajectory evaluation (STE) [18], and theorem proving is done in the ThmTac proof tool [2].

On a philosophical level, we approach verification much the same way as program construction, by emphasizing the role of the human verifier in decomposing the top-level problem to relatively simple steps amenable to automation, instead of striving at maximizing the amount of automation. Continuing the analogy, we identify two separate, although partly overlapping, aspects of proof construction: *proof design*, concerned with the problem of devising a proof of correctness for a given system in the first place, and *proof engineering*, concerned with the structure and formulation of such a proof.

Probably the most important observation in our work is that in large-scale application of formal verification, conscious attention needs to be paid to the proof design and engineering aspects, in addition to the conceptual argument behind the proof, or the fundamental aspects of the verification framework. In retrospect, this should not be surprising. After all, decades of experience have shown the crucial importance of careful software design and engineering practices for large-scale system development projects. However, in proof development we do not have the same wealth of established models on which to base the work as in software development. In our verification work, we failed to appreciate the need for clear development principles early enough. This resulted in extensive amounts of proof rewriting work later on, when the problems caused by poor choices in proof structuring and formulation became apparent.

We start by looking at the aims and challenges of applying formal verification in the large scale in Sect. 2. Section 3 introduces the Pentium 4 divider circuit. In Sect. 4 we outline our verification methodology, and in Sect. 5 the technical verification framework. Section 6 discusses our approach to proof design, and Sect. 7 gives an overview of the steps involved in the verification of one individual division micro-operation (for more proof details, see [13,14]). Then, in Sect. 8 we examine aspects of proof engineering in some more detail.

2 Large-Scale Verification

Before looking at our case study, let us discuss more generally our experiences regarding the challenges of applying formal verification as a routine part of an active industrial development project, as opposed to a one-off case study illustrating the feasibility of a particular verification approach.

A basic difference between the two is that in a development project, formal verification is not the main concern of the project, but only a fairly small part of it, one tool among others. This is reflected in both the properties and the systems to be verified. On the one hand, the choice of what is to be verified is based more on what is considered to be critical for the project, rather than what happens to suit well to a particular verification technique. Although available technology naturally sets limitations to what can be verified, in principle the verifier should be able to address any correctness issue that may be relevant to the final product. On the other hand, systems are less than perfect regarding the needs of verification. The verifier has little control over them, and cannot massage a system to make the verification problem easier.

In actual fact, in a hardware development project like ours, there is an inherent conflict between the goals of the hardware design and the needs of the formal verification. For design, performance considerations are the most crucial concerns, and simplicity and clarity come distant second. Verification, on the other hand, needs elegance and clarity, for making specifications understandable and any kind of formal reasoning possible. In effect, for formal verification we must create a clear and elegant abstract description of something that is not in itself clear and elegant at all.

The practical problems of formal verification start with the formulation of a precise specification. Written design specifications, if they exist, tend to overlook low-level details necessary for formal verification. Furthermore, in a system under development, current specifications often exist only in the minds of the designers. Therefore, writing a precise specification almost invariably involves some reverse engineering of the system, with the obvious danger that a specification replicates problems of the system.

The largest challenge in industrial formal verification is clearly just carrying out the verification at all. Given the complexity of industrial systems, and the level of support current tools provide, this is often a task requiring great ingenuity. To illustrate the size and complexity of current systems, a print-out of the Pentium 4 divider register-transfer-level source code, the basis of our verification work, is about one inch thick, and the unit is only a small fraction of the whole processor.

However, carrying out the verification as part of an active development project sets additional requirements beyond “just doing it”: we have to be able to make plans and promises about the verification before actually carrying it out, and then keep these promises. This means that the verification approach must be sufficiently predictable and well understood to make meaningful advance planning possible.

Probably the largest difference between an individual case study and systematic application of formal verification lies in the verification maintenance aspect. For an isolated case, a proof can be almost write-only, as after the verification has been completed, it will not need to be revisited. For an active development project the situation is quite the contrary: the verification will need to adapt to changes in the underlying system and the specification over the lifespan of a project. As a matter of fact, due to the high initial investment required by formal verification, it is natural to reuse the results in future projects, as well, so the verification is quite likely to outlive the project it was originally part of. In our case, the underlying system model changed sometimes several times a week, and we expect the proofs to be used for five years or more. It is also natural to carry out large verification tasks incrementally, starting at a more restricted set of behaviours and properties, generalising this step by step, which means that the verification needs to be carried out repeatedly, even if the underlying system does not change. All this means that for larger-scale formal verification, the robustness of the verification method and easy modifiability of proofs are extremely important.

While the accuracy of formal verification is naturally important in any setting, in an industrial project it is of special significance, in relation to more traditional testing-based approaches to validation. These methods are likely to be used in parallel with formal verification, and as they typically produce partial results much faster, simple errata appearing frequently are likely to be caught by testing long before formal verification would detect them. Therefore, the value of formal verification lies in its ability to discover the hard-to-find errata that testing would miss. In order to find these subtle problems, it is essential that both the model of the system and the properties to be verified reflect accurately the real system and its intended properties.

An ingredient in the accuracy of verification is the concern of reviewability. The specification of a system should be reasonably clear and crisp to be easily reviewable against informal notions of correctness, without understanding internal details of the system. It should also be easy to find out from a verification what exactly does it prove, how this is proved and, especially, what the underlying, unstated assumptions are.

3 Divider Circuit

To illustrate the Pentium 4 divider unit, consider first the simple iterative division-remainder algorithm sketched in Fig. 1. It takes two normal floating-point numbers N and D as input, and produces the rounded quotient Q of N divided by D. This algorithm is essentially the same as the one taught in school for pen-and-paper division, although in binary instead of decimal. The value of *iteration_count* depends on the required precision of result. The algorithm can be easily modified to compute the remainder R instead of the quotient Q by just switching the entity to be output.

The algorithm of Fig. 1 can also be used to compute square root of N with minor modifications. First, a preprocessing step aligning N is added so that $N_e - bias$, the unbiased exponent, becomes even. Second, both occurrences of D_m inside the loop are replaced with $2 * Q_m[i] + 2^{-i}$ (notice that the value varies between iterations), and third, the final exponent computation is replaced with $Q_e := (N_e - bias)/2 + bias$.

Figure 2 depicts a simplified hardware implementation of this division algorithm. The circuit has inputs for the dividend N, the divisor D and some control signals. Mantissa calculation is done in a feedback loop, one iteration per clock cycle, and exponent calculation is done in a separate subunit. As output, the circuit produces the result W of the required calculation and some control information, such as various flags. Correct behaviour of the circuit can be easily characterised by the formula $r(W) =$ round($r(N)/r(D)$) for division and by $r(W) =$ round($\sqrt{r(N)}$) for square root, where the precise meaning of the function 'round' depends on the intended rounding mode and precision, and the function r maps a floating-point representation to the real number it encodes.

Although similar in principle, current industrial hardware implementations of division algorithms are many magnitudes more complex. For example, they may use redundant or multiple representations of Q and R, produce more than one quotient bit per iteration, or perform speculative calculations [6]. The Pentium 4 divider unit is no exception: it implements a highly optimised double-pumped radix-2 SRT division algorithm, producing two quotient bits per clock cycle, and has over 7000 latches.

input: two normal floating-point numbers $N = (N_s, N_e, N_m)$ and $D = (D_s, D_e, D_m)$
    (we view abstractly N_e and D_e as natural numbers and N_m and D_m as fractions below)
variables: floating-point numbers $Q = (Q_s, Q_e, Q_m)$ and $R = (R_s, R_e, R_m)$, integers $imax$ and i

$i := 0$; $imax :=$ *iteration_count*;
$Q_m[0] := 0$; $R_m[0] := N_m$;
while $i < imax$ **do**
    /* determine quotient bit $q_i \in \{0, 1\}$ */
    if $R_m[i] < D_m$ **then** $q_i := 0$ **else** $q_i := 1$ **fi**
    /* update quotient and remainder accordingly */
    $Q_m[i+1] := Q_m[i] + 2^{-i} * q_i$; $R_m[i+1] := 2 * (R_m[i] - q_i * D_m)$; $i := i + 1$
od
$Q_s := N_s$ **xor** D_s; $Q_e := N_e - D_e + bias$; $Q_m := Q_m[imax]$;
output (**round**(Q_s, Q_e, Q_m))

Fig. 1. Simple iterative division algorithm

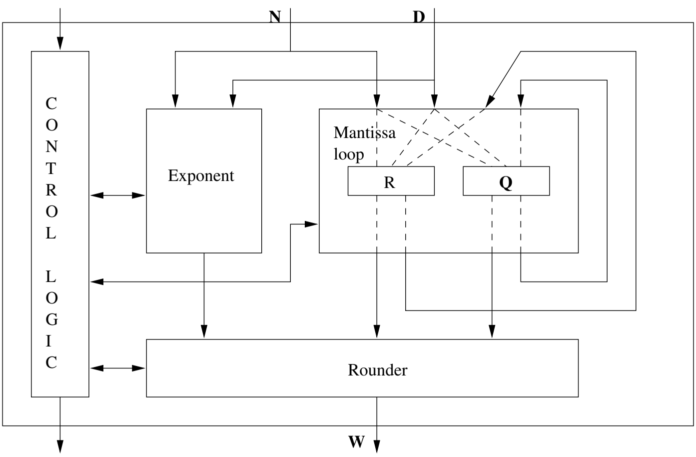

Fig. 2. Simple divider hardware

The Pentium 4 divider unit also supports a number of different variations of the basic division, remainder and square root operations. While the simplest ones differ only with respect to rounding precision, the circuit supports a collection of specialised micro-operations used primarily for microcode flows computing transcendental functions. Additionally, the circuit supports a collection of Single Instruction Multiple Data (SIMD) instructions called SSE (Streaming SIMD Extension) and SSE2, optimised for multimedia applications. For some of these, several passes of the mantissa loop are executed, and for some others, the normal full-width datapath is split into two halves, both effectively executing the same algorithm in parallel. Altogether the Pentium 4 divider unit supports about twenty materially different variants of the basic operations. For more discussion on Pentium 4 micro-architecture, see [10].

4 Verification Methodology

The goal of our verification methodology is to provide a completely machine-verified proof of correctness, with low-level steps justified by fully automated model-checking, relating high-level specifications all the way down to the actual description of the circuit in a unified framework. The four basic principles, mechanised verification, automation in the low level, actual model of the circuit, and uniform framework, are all answers to the challenges of large-scale verification. While we believe these principles to be quite uncontroversial, let us briefly outline the arguments behind them.

Consider mechanised proof-checking first. An alternative view would be to consider proofs as social objects, and trust that the scrutiny of sufficiently observant peers will find any mistakes [17]. Unfortunately, many of the proofs related to formal verification of circuits are rather boring from a mathematical perspective, and do not motivate qualified individuals to delve into them deeply enough. For example, in doing mechanised theorem-proving on our rounding specifications, we discovered a hole that had resisted close scrutiny for a long time. The second principle, automation, is necessary for the sheer size of industrial circuits. While automation does not necessarily need to imply model-checking, we are currently not aware of other sufficiently robust approaches.

Our decision to base the verification on the actual description of the circuit used in the design flow is motivated by the reliability and maintainability of the work. Many actual errata in circuits, e.g. the infamous Pentium FDIV erratum and all errata found in our work, are caused by low-level details. Furthermore, a separate high-level description would need to be constantly updated to reflect changes to the actual design.

Regarding the unified framework, some verification case studies use a variety of tools, e.g. the results of model-checking are transferred from one system to another for theorem-proving purposes. In our opinion this approach leaves room for error in the form of unstated or poorly understood assumptions underlying the translation of statements from one formalism to another. A single, tightly integrated environment also helps in making the verification more manageable and reviewable, as assumptions, qualifications and verified statements can be expressed uniformly.

Underlying our verification methodology is the philosophical belief that verification of systems should be an activity analogous to programming. We view programming as the human activity of organising individual primitive instructions, each of which can be mechanically executed by a computer in an efficient, dependable and predictable fashion, to a larger pattern to perform the intended high-level task. In the same way, we view verification as the human activity of organising individual primitive proof steps, each of which can be mechanically verified by a computer in an efficient, dependable and predictable fashion, to a larger pattern to establish the intended high-level specification.

Based on this view, we tend to emphasise the proof decomposition aspect over automation. We are naturally not in any way against the use of sophisticated algorithms: they can help verification just like a subroutine library can help programming. Nevertheless, our trust in our ability to carry out a given programming task is usually based more on our programming skills, rather than on being lucky enough to find a tool that already happens to perform the task. In the same way, in our opinion, our trust in being able to carry out a given verification task should be based primarily on our decomposition skills and the robustness of the underlying primitive verification steps.

5 Technical Verification Framework

Our verification framework consists of a collection of definition and theorem libraries built in the Forte environment, a combined model-checking and theorem-proving system. The interface and scripting language to Forte is `FL`, a strongly-typed functional language in the ML family [16]. It includes binary decision diagrams (BDDs) as first-class objects and symbolic trajectory evaluation as a built-in function.

Symbolic trajectory evaluation, based on traditional notions of digital circuit simulation, is an efficient method for determining the validity of a restricted class of temporal properties over circuits. It allows statements of the form $\models_{\text{ckt}} [ant \Longrightarrow\!\!\!> cons]$, where the *antecedent* (ant) gives an initial state and input stimuli to the circuit ckt, while the *consequent* ($cons$) specifies the desired response of the circuit. Formally, the meaning of the statement is: all sequences in the language of the circuit satisfying the antecedent will also satisfy the consequent. Antecedents and consequences are formed by conjunction from basic formulae of the form $\texttt{N}^t(node\ \texttt{is}\ value\ \texttt{when}\ guard)$, where t is an integer, $node$ is a signal in the circuit and $value$ and $guard$ are Boolean expressions. The meaning of a basic formula is "if $guard$ is true then at time t, $node$ has value $value$".

The efficiency of trajectory evaluation is based on built-in support for data abstraction via a lattice of simulation values. The simulation model used by Forte extends the conventional Boolean domain to a lattice by adding a a bottom element `X` and a top element $\top$. Intuitively the value `X` denotes lack of information: the signal could be either `T` or `F`. The essential relation between such four-valued and Boolean sequences is that any assertion verified over a sequence containing `X`s will hold for sequences with `X`s replaced with either `T` or `F` [4,3].

Theorem proving in Forte is done in the ThmTac proof tool, an LCF-style implementation of a higher-order classical logic. Its principal aim is to enable seamless transitions between model checking, where we *execute* `FL` functions, and theorem proving, where we *reason* about the behaviour of `FL` functions [2]. Roughly speaking, if a term does not include any free variables, contains quantification over Boolean domains only, is evaluatable within the computational resources available, and evaluates to true, we can turn it into a theorem and use it for reasoning.

As the restricted language used for trajectory evaluation is too weak to allow expression of many interesting properties, we use a variant of the traditional pre-postcondition framework (see e.g. [7]) for formulating temporal aspects of our specifications. In our approach specification statements are of the form $\{\phi_{in}\}(tr_{in}, ckt, tr_{out})\{\phi_{out}\}$ where a trajectory assertion $tr_{in}(x)$ binds a vector x of Booleans to some input signals of ckt at the time the input is intuitively read by the circuit, trajectory assertion $tr_{out}(y)$ binds a vector y similarly to some output signals, 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. Formally, this statement is shorthand for the formula:

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

Intuitively the formula states that for any vector of values x satisfying the precondition $\phi_{in}(x)$, 1) there is some output vector y such that for every execution e, if $tr_{in}(x)$ is true of e, then so is $tr_{out}(y)$, and 2) for every vector y for which 1 holds, the postcondition property $\phi_{out}(x, y)$ holds, or more loosely, that whenever precondition ϕ_{in} is satisfied, the circuit guarantees that postcondition ϕ_{out} is also satisfied.

The validity of a pre-postcondition statement in the form above can, in principle, be determined by direct evaluation, although computational resource requirements mean that in practice this is feasible only in limited circumstances. To allow reasoning about the flow of computation in a structured way, our proof framework includes general reasoning rules for pre-postcondition statements, such as precondition strengthening, postcondition weakening, conjunction, sequential composition and bounded iteration [13].

When dealing with arithmetic circuits, both specifications and reasoning are often naturally expressed in terms of arithmetics. As model-checking techniques using BDD based representations can only deal with bit-vector operations, our proof framework includes a library of provably correct bit-vector arithmetic operations, which have an exact correspondence with integer operations.

To support verification of floating-point operations, our proof framework includes a general-purpose theorem library for floating-point numbers and rounding [14]. Analogous to the case of integer arithmetics above, the library supports floating-point numbers and rounding at the bit-vector level for model-checking, and at the mathematical level for reasoning. As currently our framework does not support reals, only integers, we have adopted the work-around of multiplying all entities by a sufficiently big number 2^{BN} so that every real number that is relevant for our proofs maps to an integer.

6 Proof Design

Finding a proof for a given property and system is naturally always a heuristic process. Nevertheless, just as in program design, it is worth while to articulate general strategies for finding a solution, and to impose some structure on the process. In addition to offering guidelines for construction of future proofs, spelling out design principles gives a vocabulary for communicating and comparing solution strategies.

For low-level STE model-checking work, the methodology discussed in [1] gives us the basic structure for finding out circuit interfaces, describing them abstractly, and carrying out trajectory evaluation runs. On a higher level, our decomposition strategy is based on looking at the abstract algorithm the circuit is intended to compute.

We start by partitioning the algorithm to regions in such a way that the computation within each region in isolation can, in principle, be efficiently carried out for symbolic initial values using BDD's. For example, a region involving only addition and subtraction, or only shifts, is rather likely to have a concise BDD representation. For the division algorithm, the body of the loop forms a good candidate for a region.

For each region, we then try to locate the computations corresponding to the region in the circuit, find boundaries separating the computations, and signals corresponding to the variables of the abstract algorithm. At this point, we may notice that the algorithmic description of the circuit is too coarse to allow an adequate correspondence. For example, when trying to map the loop body of the division algorithm to the actual circuit, we notice that the circuit uses auxiliary entities, effectively different approximate representations of Q and R. In this case, we will need to refine the abstract algorithm. When mapping entities of the abstract algorithm to the circuit, the relation between the levels is not necessarily one-to-one: a single abstract variable may correspond to a non-trivial function of a collection of signal vectors, with different timing characteristics.

Once we have located the regions and boundaries in the circuit, we verify that each region in the circuit implements correctly the corresponding region of the algorithm. As the original partitioning to regions was chosen so that calculations within each abstract region could be efficiently carried out using BDD's, there is a fairly good likelihood that we can also efficiently simulate the behaviour of the circuit region with STE. While it may naturally happen that an intermediate value in a region is not concisely representable by BDD's even if the boundaries are, we have so far never encountered such a case.

After having model-checked each region separately, the corresponding relations are combined using theorem proving, in the way described by the abstract algorithm to yield a proof of the top-level correctness statement. In other words, our proof design approach is top-down, whereas proof construction takes place bottom-up.

The decomposition of the abstract algorithm to the regions used in verification does not need to coincide with the decomposition used in implementing the algorithm in the circuit. In fact, one perhaps surprising observation in this work has been that this is hardly ever the case: the regions and boundaries used for verification rarely bear much resemblance to the module structure, nor to the latch boundaries of the circuit. On second thought, this may not be so surprising: modules of the circuit are more tied to the physical area, and especially in later stages of a project, circuit logic tends to be moved from one module to another, or from one side of a latch to another, with fairly little regard to its conceptual position in the overall computations. Consequently, circuit modules do not usually have a clear algorithmic characterization useful to us.

We believe that this proof design approach is quite widely applicable to various kinds of datapath-oriented circuits. As neither the proof nor the verifier needs to know the exact way computations within a circuit region are actually carried out, concrete proof plans can be made in advance. The approach also appears to be robust regarding changes to the design: most changes are likely to take place at a local level, so while they may require adjusting the boundaries or the description of the intended computation within a region, the overall proof structure does not need to change.

7 Divider Verification Outline

An informal specification of the circuit's correctness is quite easy to come up with:

> *IF a division operation is started AND the input values N and D are within the range handled by hardware AND the environment behaves according to the expected protocol AND the circuit is internally in normal operating state, THEN at the time the circuit produces output W, the equation $r(W) = round(r(N)/r(D))$ holds.*

When formalising this, the part concerning the relation of input and output data values is straightforward, as it follows from the IEEE specification on floating-point arithmetics [11], although formalisation of the standard itself is non-trivial. However, the problem lies at the left side of the implication: what are "normal internal operating state" and "expected environment protocol"? Characterising these very circuit-dependent aspects required a fair amount of investigative work. To increase confidence in the correct characterisation of the environment assumptions, we also used an existing test suite to check for their validity in a variety of circumstances using traditional test-based validation. In principle the environment assumptions could have been verified in the context of the whole processor, but we did not have the resources for this.

Further, mentioning the internal operating state in the specification violates the principle of external visibility. Therefore, we needed to strengthen the statement by proving separately that *whenever a division operation can be started, the circuit is internally in normal operating state*, which allows us to discharge the last conjunct of the antecedent. This proof was carried out in a fairly traditional temporal-logic-based framework. However, as it is separate from the main datapath proofs, we shall not discuss it here.

The top-level correctness statement can then be formalised by

$$\{IN\} \ \ (tin, ckt, tout) \ \ \{IO\} \tag{1}$$

where the precondition *IN* formalises the four conjuncts of the antecedent of the informal specification, and the postcondition *IO* is defined by

$$\begin{aligned} IO = \exists Q.\,(\ ri(W) \ &= \ \text{round}_{ri}(Q)\ ) \ \wedge \\ (\ Q * ri(D) \ &\leq \ ri(N) * 2^{BN} \ \leq \ (Q + \epsilon) * ri(D)\ ) \end{aligned} \tag{2}$$

and where trajectory function *tin* binds N and D to input data signals at the start of the operation, and *tout* binds W to output signals at the time the output is ready. The formula *IO* is slightly more complex than the intuitive specification: the extra entities Q, intuitively denoting the unrounded quotient, and ϵ, denoting some fixed small value, are needed because of the lack of real numbers in our current framework. The function round_{ri} is a rounding function working on the integer representation of reals.

As the algorithm and the hardware are iterative in nature, the verification is based on a loop invariant for the mantissa calculation. At the top level, there is a natural mathematical invariant MI_i relating the quotient and remainder mantissas $Q_m[i]$ and $R_m[i]$ to the input numbers D and N, derived from the defining equation of division:

$$MI_i = (N_m = Q_m[i] * D_m + 2^{-i} * R_m[i]) \ \ \wedge \ \ (R_m[i] < 2 * D_m) \tag{3}$$

Due to the multiplication operation in this invariant, it is not amenable to verification by direct model-checking. Therefore, the problem is further decomposed into verification of MI_1 after first iteration, and verification of an equation MR_i between current and previous loop values for each subsequent iteration. The equation MR_i is based on the recurrence relation the loop is supposed to compute. Further, to verify the relation MR_i by model-checking, two bit-vector relations are introduced: a bit-vector recurrence relation BR_i that coincides with the mathematical relation MR_i, and a low-level bit-vector invariant BI_i expressing a consistency constraint on loop data.

Using this decomposition, verification of the mantissa computation in the circuit consists of the following steps:

$$\{IN\} \ \ (tin, ckt, tl_0) \ \ \{BI_1 \wedge MI_1\} \tag{4}$$

$$\forall i.\ (0 \leq i < imax)\ \Rightarrow\ (\{BI_i\}\ (tl_i, ckt, tl_{i+1})\ \{BI_{i+1} \wedge BR_i\}) \tag{5}$$
$$\forall i.\ (0 \leq i < imax)\ \Rightarrow\ (BR_i \Rightarrow MR_i) \tag{6}$$
$$\forall i.\ (0 \leq i < imax)\ \Rightarrow\ (MI_i \wedge MR_i \Rightarrow MI_{i+1}) \tag{7}$$

where trajectory function tl_i binds R_m, Q_m and other data items to corresponding signals for iteration i. Statements 4 and 5 are verified directly by model-checking. Considering our proof design strategy, the binding functions tl_i express the boundaries of the regions for model-checking. Statement 6 involves reasoning about the correspondence between bit-vector operations and their arithmetic counterparts, and statement 7 relies on pure arithmetic reasoning. Using pre-postcondition reasoning, steps 4–7 can then be combined to a correctness statement for the complete mantissa computation:

$$\{IN\}\ (tin, ckt, tl_{imax})\ \{BI_{imax} \wedge MI_{imax}\} \tag{8}$$

The correctness of the final rounding stage can be expressed by the formula:

$$\begin{aligned} MRND &= (\ ri(W)\ =\ \mathrm{round}_{ri}(ri(s,e,m))\ ) \quad \text{where} \\ s &= sgn(N)\ XOR\ sgn(D) \\ e &= exp(N) - exp(D) + bias \\ m &= Q_m[imax] \end{aligned} \tag{9}$$

To verify the rounding stage by model-checking, two further bit-vector relations are needed: a bit-vector version *BRND* of the mathematical relation *MRND*, and an auxiliary relation *AUX*, which expresses constraints on the final loop output necessary for the proper behaviour of the rounder. Then, verification of the rounding stage consists of the following steps:

$$\{BI_{imax} \wedge AUX\}\ (tl_{imax}, ckt, tout)\ \{BRND\} \tag{10}$$
$$BRND \Rightarrow MRND \tag{11}$$
$$BI_{imax} \wedge MI_{imax} \Rightarrow AUX \tag{12}$$
$$MI_{imax} \wedge MRND \Rightarrow IO \tag{13}$$

Statement 10 is verified by direct model-checking. This is a good example of the differences between boundaries used for verification and those of the circuit units: the starting boundary tl_{imax} is not at the rounder input in the circuit, but inside the mantissa loop. Statement 11 reasons about the correspondence between bit-vector and mathematical versions of rounding, and statements 12 and 13 involve mostly arithmetic reasoning. Using pre-postcondition reasoning, steps 10–13 then yield:

$$\{BI_{imax} \wedge MI_{imax}\}\ (tl_{imax}, ckt, tout)\ \{IO\} \tag{14}$$

Finally, statements 8 and 14 can be joined by sequential composition to show the top-level correctness statement 1. For more proof details, see [13].

While this discussion has concentrated on the division operation, the proof for square root is analogous. The most crucial issue in the model-checking part of the verification was determining the boundary tl_i and the invariant BI_i exactly: some parts are easy, like the location or the expected ranges of data values in the loop, but some are extremely implementation-dependent.

8 Proof Engineering

Once we had completed a proof for the first micro-operation, effectively the proof outlined in Sect. 7 and reported in [13], we believed that the largest body of work was behind us, and the proofs for the remaining cases would fall out easily. We were wrong. The length of the basic proof for one micro-operation, excluding library code, is about 15 000 lines, about 85% of which is related to the theorem-proving effort, and 15% to the model-checking. The naive solution of replicating the code for each micro-operation and then making necessary changes would have resulted in 20 * 15 000 = 300 000 lines of code, clearly an organisational and maintenance nightmare.

Consequently, we had to reformulate the proofs avoiding code duplication, while accounting for the differences between micro-operations. These differences come in various flavours: The loop body of the division and square root operations is different, although the general structure is the same. Mantissa loop is executed for a different number of times for different precision modes. For certain SIMD operations, the datapath is split to two halves, and both halves implement the same algorithm as the full width datapath, except for certain details. The two halves are nearly, but not exactly symmetric. Flag and fault behaviour for customised micro-operations differs from standard ones. For some SIMD operations, multiple passes of mantissa loop are executed, and for some others, fault behaviour reflects several pieces of data. The rounder depends on different constraints on the loop output for different micro-operations.

It turned out that most of the theorems we proved for the first completed proof were not sufficiently general. Various values, assumptions and definitions were hard-coded into the proofs, although they vary between micro-operations. This lack of generality caused an extensive amount of proof rewriting. Actually, we spent more time rewriting proofs than writing them in the first place. Although some consolation is provided by anecdotal evidence in the literature [15,12] that we were not the only ones encountering the problem, this clearly is not a preferable state of affairs. We basically made the same mistake as starting a software project by rushing to write program code, without precise planning of the overall structure of the system. Moreover, we did not write our original proofs in a fashion that would have supported modifications and maintenance.

So, how do we write robust, understandable, and maintainable proofs? It appears to us that theorem-proving is often used in a fairly static setting, where maintainability is not a crucial concern, so we did not seem to have too many models. While we can naturally learn from principles used to enhance software maintainability, we cannot just simply equate the problems of software and proof maintenance. As a matter of fact, we would argue that the latter is considerably harder than the former. First, for software only the semantics of objects matter: we can freely reformulate a definition as long as its denotation does not change. For proofs, on the other hand, syntactic structure of terms matters as well. Secondly, the rigour required for formal reasoning leaves much less leeway for sticky-tape solutions than in the case of software. Thirdly, the arduousness of theorem proving means that proof reformulation is harder than program reformulation. This conspires against maintainability in two ways: once a proof has been created, no matter how imperfect, there is a great temptation to leave it as it is, but when we will need to modify it later, the penalty is even higher.

Many specific questions emerge in relation to writing robust and maintainable proofs. For example, if we classify objects related to proofs to term language definitions, claims regarding such definitions, and proofs of claims, what principles are relevant for formulation of objects in each group? How to structure a theorem hierarchy? How to formulate the proof of an individual theorem? How to represent proof hierarchies which are similar except for some details? While we cannot claim to have the best possible solutions, we were forced to explicitly address these issues during our verification work.

Let us start from the question of maintainable definitions. We adopted the practice of formulating definitions as hierarchies of definition layers, with each layer concentrating on a separate aspect. The layers are very thin, usually a single level of a definition is less than five lines long. For example, consider the definition of *MRND* (equation 9 in Sect. 7). As written there, it contains the aspects of sign and exponent calculation, mantissa definition, conversion from floating-point representation to a number and rounding. To be able to reason about these aspects separately, we define each of them as a separate layer. Then we can easily change parts of a definition, e.g. to reuse parts of *MRND* for square root proofs by changing sign and exponent calculation.

The layering is repeated in the formulation of claims. For each layer present in a definition, we write a separate claim, with the intention that it can be proved on the basis of the definition of the current layer and claims relating to the layer immediately below. This induces a natural theorem hierarchy. We also try to state claims always in terms of relation names, instead of the actual relations. So, instead of *if* $a < 2^{23}$ *then* ... we write *if (in_bounds* a*) then* ... where *in_bounds* $x = x < 2^{23}$. In this way, even if the actual bound used in *in_bounds* will change, subsequent theorems will be unaffected as long as the current theorem remains true. This isolates effects of changes and improves maintainability. On the negative side, the approach may lead to a proliferation of claims.

Given a hierarchy of definitions and a claim relating to some layer, how do we write a maintainable and understandable proof for the claim? A proof composed of many simple manually crafted steps is more likely to need editing, even for small modifications in the claim, than a proof with fewer, more automated steps. However, when modifications are needed, they are likely to be easier to make in the former case than in the latter, as it is easier to recreate the conceptual argument behind the proof from its code representation. Thus, in our experience, for small changes in the claim, highly automated proofs are superior, but for larger changes, manually crafted proofs are more maintainable. Unfortunately, without foreseeing what kinds of changes will be required, it is hard to plan for the best outcome.

The core issue in avoiding code duplication is proving each argument only once. However, it is often easier to prove n instances of a general theorem, than the theorem itself, so there is a tradeoff between maintainability and ease of proving. We usually opted for proving the general case for any $n > 2$, unless the general proof was fundamentally more difficult.

To improve understandability and manageability, we organised all our proofs in modules that, conceptually, are given a set of objects and theorems concerning those objects, and provide another set of objects and theorems. In modules we followed a principle of locality, and packaged all proofs requiring access to the internal details of a definition together with the definition itself. This greatly increases maintainability of code, as changes to a definition only require changes to the local module, as long as externally visible theorems are not affected. Modularity also allows us to represent proof hierarchies differing only in some details without code duplication, by using alternate modules for the differing aspects of the proof. This way, we managed to capture all the different proof variations in only about 45 000 lines of proof code.

We found the layering of definitions, claims and proofs to be indispensable for creating maintainable proofs. However, splitting definitions to minute steps has a negative effect on reviewability. To alleviate this problem, we experimented with proofs using two sets of definitions: monolithic ones, used in the top-level specifications for reviewability, and layered ones, used for the rest of the proof. The transition from layered to monolithic definitions then takes place just below the top level.

Technically our proof development environment was rather austere. In ThmTac a user writes down a textual representation of proof steps, and a proof is an `FL` term like any other, so no special proof capture or replay mechanism is needed. For interactive proof development we used a ThmTac interface for stepping through a proof, and a mechanism for assuming theorems without having to evaluate their proofs. We used no special module, version or proof consistency management tools. While they might have helped on some occasions, the real problem we faced was not in these aspects, but in deciding how to write down the proof. To guarantee that all pieces fit together after a round of changes, we revalidated the whole proof hierarchy from scratch overnight. What would have made our work easier would have been a mechanism for supporting and enforcing good code writing practices, such as module visibility rules. Now, our work environment did not support modules directly and we had to emulate them by conditional load sequences. Another item in our wish-list would have been tool support for incremental proof changes, such as analysing the precise point in which an old proof and an attempted new proof diverge.

Looking back on the lessons learned during our verification work, we would advocate the strategy of starting the verification by a quick, semi-informal decomposition of the top-level property to model-checkable portions, carrying out the model-checking, and then planning the complete formal proof structure in great detail before starting any theorem-proving. The proof planning stage should result in a documented description of the modules, definitions, claims, required proofs and their relations. Precision in this stage is crucial, as the high cost of proof changes makes it important to get the proof structure right at the first attempt.

9 Conclusion

We have examined verification methodology, proof design and proof engineering aspects relevant to large-scale industrial application of formal verification, based on our experiences in verifying the Pentium 4 divider unit, to our knowledge one of the largest industrial hardware verification case studies. The verification took about two and a half person-years of work in total, and it was carried out in parallel with later stages of the circuit design, before silicon was produced. No errata were found in the original design, but applying the proof suite to a proliferation project caught a few rather tricky errata caused by unintended interactions between micro-operations executing in parallel.

Considering the technical requirements of industrial verification, in our experience it is more important that tools are robust and dependable than that they are technologically the most advanced ones: all the basic techniques that we are using have been well known for years. In a combined theorem-proving and model-checking approach, we found tight integration of the techniques to be necessary: theorem-proving is used in all stages of our verification work, from justifying model-checking optimizations in the low level, to very abstract reasoning in the high level. We also found the open-endedness of general theorem-proving indispensable for the work.

During the course of our work we identified a number of solutions to practical problems arising in proof design and engineering. Nevertheless, we would like the message of the paper to be not so much of a solution but of a problem statement: How do we write large proofs in a manageable way? If we are to apply formal verification, especially formal theorem-proving, as a routine part of industrial design flow, we must have models and principles addressing this issue, as otherwise the work will become infeasible.

References

1. M. D. Aagaard, R. B. Jones, T. F. Melhan, J. W. O'Leary, and C.-J. H. Seger. A methodology for large-scale hardware verification. In *FMCAD*, volume 1954 of *LNCS*, pages 263–282. Springer, 2000.
2. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In *TPHOLs*. Springer, 1999.
3. M. D. Aagaard, T. F. Melham, and J. W. O'Leary. Xs are for trajectory evaluation, Booleans are for theorem proving. In *CHARME*. Springer, 1999.
4. C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *CAV*. Springer, 1999.
5. P. Curzon. The importance of proof maintenance and reengineering. In J. Alves-Foss, editor, *HOL-95: B-track Proceedings*, pages 17–31, Sept. 1995.
6. M. D. Ercegovac and T. Lang. *Division and Square Root, Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.
7. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
8. I. Hayes. Applying formal verification to software development in industry. In I. Hayes, editor, *Specification case studies*, pages 285–310. Prentice-Hall, 1987.
9. S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification*, chapter 1, pages 3–78. Springer Verlag; New York, 1997.
10. 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.
11. IEEE. *IEEE Standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985, 1985.
12. M. Jones and G. Gopalakrishnan. A PCI formalization and refinement. In *TPHOLs 2000, Supplemental Proceedings*, OGI Technical Report CSE 00-090, pages 115–124, 2000.
13. R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. In *TPHOLs*, volume 1869 of *LNCS*, pages 338–355. Springer, 2000.
14. R. Kaivola and N. Narasimhan. Multiplier verification with symbolic simulation and theorem proving. Submitted for publication, 2001.
15. A. Mokkedem and T. Leonard. Formal verification of the Alpha 21364 network protocol. In *TPHOLs*, volume 1869 of *LNCS*, pages 443–461. Springer, 2000.
16. L. Paulson. *ML for the Working Programmer,*. Cambridge University Press, 1996.

17. 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.
18. 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.