A Methodology for Large-Scale Hardware Verification

Mark D. Aagaard[1], Robert B. Jones[2], Thomas F. Melham[3], John W. O'Leary[2], and Carl-Johan H. Seger[2]

[1] Performance Microprocessor Division, Intel Corporation, RA2-401, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA
[2] Strategic CAD Labs, Intel Corporation JFT-104, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA
[3] Department of Computing Science, University of Glasgow Glasgow, Scotland, G12 8QQ

Abstract. We present a formal verification methodology for datapath-dominated hardware. This provides a systematic but flexible framework within which to organize the activities undertaken in large-scale verification efforts and to structure the associated code and proof-script artifacts. The methodology deploys a combination of model checking and lightweight theorem proving in higher-order logic, tightly integrated within a general-purpose functional programming language that allows the framework to be easily customized and also serves as a specification language. We illustrate the methodology—which has has proved highly effective in large-scale industrial trials—with the verification of an IEEE-compliant, extended precision floating-point adder.

1 Introduction

Functional validation is one of the major challenges in chip design today, with simulation and testing a dominating element of the design effort [1]. Throughout the 1990s, formal verification [2] has emerged as a promising complement to simulation. A notable success is equivalence checking using BDDs, now widely-used for checking consistency between adjacent levels in the design flow. Research on the broader problem of functional validation has also delivered promising results in trials on industrial-scale designs [3, 4, 5].

Although algorithmic advances have increased the reach of formal verification, they have still failed to close the gap between the capability offered by verification 'point-tools' and modern design complexity. In response, we have coupled our research on verification technology and tools with research into verification *methodology*. The aim is to devise a systematic approach to organizing the activities undertaken in large-scale verification efforts and structuring the associated code and proof-script artifacts.

Algorithmic and tool research primarily address the well-known problem of model-checking *capacity* limits, while often overlooking the equally important

W.A. Hunt, Jr. and S.D. Johnson (Eds.): FMCAD 2000, LNCS 1954, pp. 263–282, 2000.

problem of managing the *complexity* of the verification activity itself. Like others, we attack capacity problems with technical innovations in the core verification algorithms. But almost any serious verification effort faces many practical difficulties other than capacity. For example, it almost certainly requires us to break a problem down into many model-checking runs—frequently many hundreds. Organizing all the cases to be considered into a coherent whole or even specifying them clearly (let alone discovering them) is complex, intellectually demanding, and error-prone.

Our methodology addressed this particular verification complexity problem by generating and organizing model-checking runs in a systematic way. More generally, the methodology gives guiding structure and sequence to the many interdependent and complex activities of a large verification effort. The methodology aims, on the one hand, to face the messy realities of design practice (e.g. rapid changes and incomplete specifications) and, on the other hand, to produce high-quality results that are understandable, maintainable—and possibly even reusable. In Section 2 of this paper, we detail aspects of this methodology that are critical for successful application of formal methods to large designs.

Our approach is applied in Forte, a formal verification environment that combines an efficient linear-time logic model checking algorithm (symbolic trajectory evaluation—STE [6]) with lightweight theorem proving in higher-order logic. These are interfaced to and tightly integrated with FL, a general-purpose functional programming language in the ML family [7]. This allows the environment to be customized and large proof efforts organized and scripted effectively. FL also serves as an expressive specification language extending the temporal logic primitives of STE. Section 3 gives a brief overview of Forte and supplies pointers to further information.

The methodology we advocate is separated into the following four distinct but overlapping activities: understanding and encapsulating the circuit, scalar verification, symbolic model-checking, and theorem proving. Each phase has a specific purpose and associated tasks, together with definable *artifacts* that result from these tasks. Section 4 provides a concrete illustration of these phases applied to the verification of an IEEE-compliant, extended precision floating-point adder.

2 Methodology

One of the defining aspects of circuit design is a complex set of trade-offs between design requirements. Different and often contradictory requirements exist—fast circuits, low power, small area, and efficient manufacturing testing. Creating an effective formal verification *methodology* can be as difficult as the process of design. An effective verification methodology must also meet several requirements:

– *Realism.* It cannot depend on resources that are not available in the design environment. For example, complete specifications are usually not available, and access to design engineers may be limited.

- *Incrementality and recoverability.* Preliminary results are needed early in a verification effort. There should be a smooth transition between simulation of special cases and a full proof, so that the effort spent delivers 'debugging value' very early on. If changing a specification, circuit, or library causes a previously-passing proof to fail, it should be possible to use incomplete proofs from earlier in the effort to help isolate the problem.
- *Transparency and confidence.* A methodology (and system) should be transparent, i.e. the verification engineer should know what has been proved and what has not. The methodology should also be *sound*, so that false positives are not possible.
- *Structure.* An effective methodology imposes structure on the overall verification effort. This not only helps new users learn, but also increases the productivity of experienced users.
- *Top-down/bottom-up.* A realistic methodology must support a mix of bottom-up and top-down techniques. The subtle features of designs and the capacity limits of model-checking are discovered through bottom-up exploration. Overall problem reduction is achieved by top-down decomposition with case splitting, induction strategies, and algorithm-specific techniques.
- *Debugging and feedback.* The bulk of any verification effort is debugging, so it is crucial to optimize the verification environment for proof *failure*, not success. Not only must the system quickly discover failures, it should provide focused feedback that enables a tight, rapid debug loop.
- *Regression.* The methodology should produce verification artifacts that are easy to maintain and adapt to changing specifications and designs. Test cases from initial proof development should continue to be usable in exploring these changes.
- *Effort reuse.* Verification is an expensive, human-intensive activity. Proof reuse should be supported to amortize the verification cost over design changes and even multiple design efforts. Specifications and high-level decomposition strategies are particularly important candidates for reuse.

The methodology detailed in this paper strikes a balance between these different requirements. It is divided into four distinct but overlapping phases of effort, which we introduce in the remainder of this section.

The first phase of the methodology is circuit *wiggling*—the process of educating oneself and one's tools about the circuit and its operating environment. The process begins by identifying an initial set of important signals. This is done by using a simulator with a 'don't care' value (called 'X') to observe the effect of driving the circuit's input and state signals. We say a circuit is 'wiggling' when enough knowledge has been gained about how to drive these signals to make defined (non-X) values appear at the circuit outputs. Efforts during this phase are not concerned that the outputs are correct, just that they appear as non-X values. But understanding the circuit's functionality at even this crudely abstract level evolves into an initial sketch of the specification.

Aside from knowledge, the primary artifact of this phase is a circuit API that defines an interface to the circuit nets and their timings.[1] Concretely, the API is a functional program (written in FL) that encapsulates the details of the circuit's timing and I/O protocol, allowing the functional specification and other later developments to abstract away from these details.

Targeted scalar verification, the second phase, begins by focusing on scalar (i.e. bit-pattern) input and state values selected to invoke easy-to-understand circuit behavior. The primary difference between wiggling and verification is that verification entails checking the circuit against a specification. Specifications are constructed incrementally. Simple portions are written first, using obvious scalar input and state vectors to check agreement with the circuit. As more of the required functionality is understood, and as the simple parts of the specification are debugged, additional functionality is added. As the specification is filled out, the scalar verification cases used are chosen to target several representative samples from each region of the input and state space explored, including boundary conditions (e.g. maximum and minimum values, zeros).

The main artifacts of this phase are a functional specification, an improved circuit API, and a set of scalar test vectors (which are saved to support regression testing). Discrepancies between the circuit and specification are usually a result of errors in the circuit API or the specification itself. But even at this early stage genuine design bugs could be discovered. When it becomes time consuming or difficult to find stimuli that result in discrepancies, it is time to move to symbolic model checking.

Symbolic model checking is the third phase of verification and marks the beginning of serious formal verification efforts.[2] Instead of driving circuit nets with scalar values, we present the circuit with symbolic values represented by BDD variables [9] and compute a symbolic representation of the resulting outputs. The model-checking engine being employed automatically verifies the resulting input/output relation against the specification, and in case of disagreement it generates scalar counterexamples—or even complete symbolic characterizations of the difference. (In fact, the algorithms in Forte are more subtle than this and operate incrementally.) Symbolic input values can be restricted by constraints, which are described by FL specifications developed in the course of the proof. For model-checking efficiency, these are encoded into parametric input functions [10].

Most hardware bugs are found during this phase of verification. Once symbolic model checking begins in earnest, BDD sizes will likely confront the capacity limits of the tool. This phase therefore includes the significant challenge of managing BDD complexity. It is often necessary to find good variable orderings, which is accomplished by a combination of automatic and manual techniques. This phase of effort also involves determining the limits of what can be checked by the model checking engines.

[1] We borrow the term API, *Application Program Interface*, from software engineering.
[2] Our use of 'symbolic' here differs from its meaning in SMV-style *Symbolic Model Checking* [8].

The main artifacts produced in this phase are an improved specification, BDD variable orderings, and a set of constraints that characterize regions of the input space for which model-checking is feasible. All these are realized as FL programs, and are easily inspected and modified by the user.

When capacity management with variable ordering provides diminishing returns, a verification decomposition strategy must be developed. This is where the worlds of model checking and theorem proving meet.

The final, *theorem proving* phase is to mechanically check the correctness of the specification and the soundness of the decomposition strategy. In addition to organizing the model checking decomposition, theorem proving helps the verification engineer discover missing details in the specification. This phase does not directly find bugs in the circuit. Instead, bugs in the proof may be detected, such as missing cases or incorrect reasoning. Fixing these bugs may require modifying the model-checking cases, which may then reveal circuit bugs.

In addition, the theorem proving phase may include proof-based analysis of the specification, typically by using a theorem prover to derive high-level properties from the specification. Because it will have been created in a bottom-up manner, there is a danger that the specification simply reverse engineers the circuit—including its bugs. The aim of this activity is therefore to gain extra confidence in the correctness of the specification by assessing it against properties independent of the circuit. These could be derived, for example, from standards such as the IEEE floating-point specification, the PCI bus specification, or a programmer's reference model for a microprocessor.

The artifacts of this phase are the final version of the functional specification, a top-level correctness statement, a collection of model checking runs, and a mechanized proof connecting the top-level correctness statement to the model checking runs. There may also be a collection of proofs of properties derived from the specification.

The four phases of verification are not strictly sequential—a fair amount of overlap and backtracking happens in practice. This is shown in Figure 1, a schematic time-line of how a typical verification might proceed through our methodology. Reading from top to bottom, the diagram shows the relative distribution of time spent on the major activities as a typical verification effort moves through the four phases of the methodology. The curves shown are, of course, only a rough guide; the exact distribution of effort will depend on the the specific verification project. Along the right-hand side of the diagram, definite criteria are listed for transitions between phases. The diagram also shows the point at which the main artifacts are finalized. Section 4 will illustrate each phase in the context of our motivating example.

Finally, finishing the theorem proving phase by no means terminates involvement with the artifacts of the proof effort. If the design is still 'live', any part of the verification code (from API on upwards) may have to be adapted to track design changes. The structure imposed on these artifacts by our methodology helps to localize changes textually. After design changes, scalar and symbolic simulations, model checking runs, and theorem proving scripts all need to be

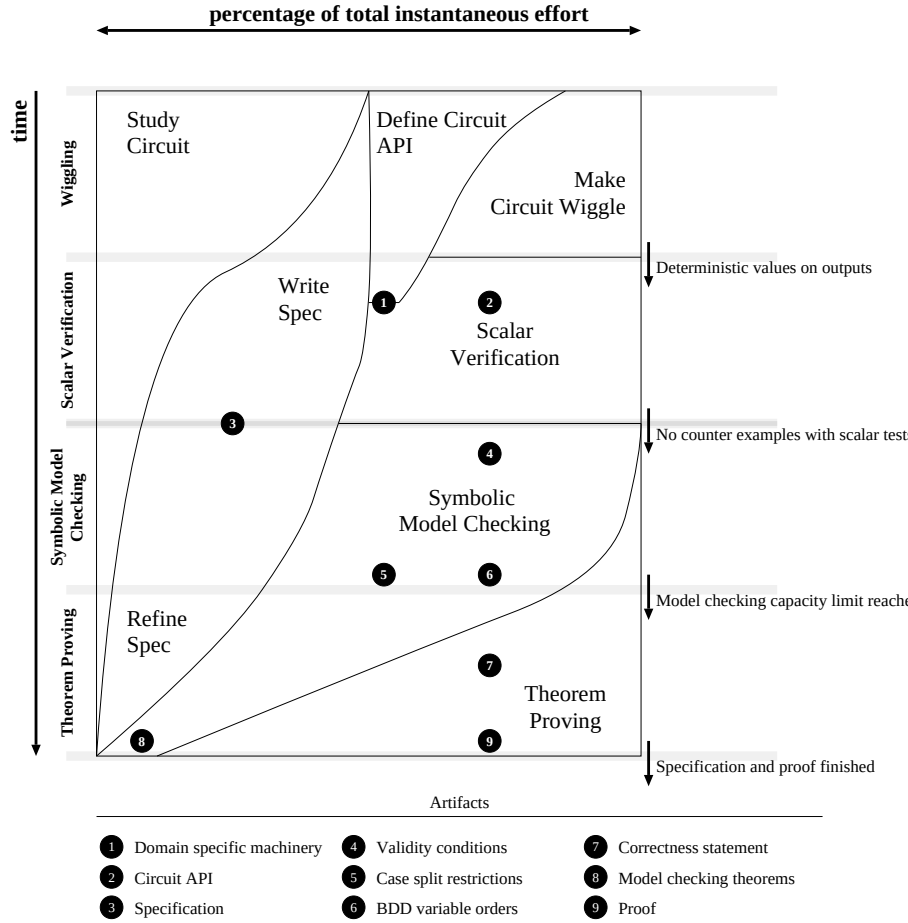

Fig. 1. Schematic time-line of verification activities.

re-run. In Forte, the concrete form of FL programs provides maintainable and incrementally executable scripts for all these verification activities. The results of one proof effort may also be re-used on future designs. If the algorithm does not change dramatically, this can be as easy as replacing the API.

3 The Forte Verification Environment

An effective methodology is obviously dependent on an effective verification platform. The methodology ideas described in this paper are embodied in Forte, our custom-built verification environment. Forte integrates model checking engines, BDDs, circuit manipulation functions, theorem proving, and a functional programming language. It builds formal circuit models from standard HDL sources,

and includes tightly integrated graphical tools for the display of circuit structures and waveforms.

In other publications, we have presented details about Forte's approach to combining model checking and theorem proving [11], symbolic simulation [10], and arithmetic verification [4]. The present paper focuses on our verification methodology, so we provide only a brief overview of Forte here. The key capabilities of Forte for this paper are the FL language, symbolic trajectory evaluation, and theorem proving.

The FL Language

FL is a strongly-typed, lazy, functional programming language. Syntactically, it borrows heavily from Edinburgh-ML [12]. Semantically, its core is similar to lazy-ML [13]. One distinguishing feature of FL is that BDDs are built into the language and every object of the boolean type `bool` is represented as a BDD.

The FL language lies at the very center of Forte. Through its embedded BDD package and primitive or defined functional entry-points, it provides a flexible interface for invoking and orchestrating model checking runs. It is also used as an extensible 'macro language' for expressing specifications, which are therefore human-readable but when executed compute efficiently checkable properties in a low-level temporal logic. Finally, it provides the control language for Forte's theorem prover and—through the concept of *lifted FL* [14]—the primitive syntax of its higher order logic.

This approach gives a generic, open framework where solutions can be tailored to individual verification problems. For each verification effort or project, the user 'programs up' in the FL functional language just the right specification constructs for the problem domain, as well as scripts for orchestrating verifications. By structuring these according to our methodology, much of this work can be reused over entire classes of verifications.

Symbolic Trajectory Evaluation

Symbolic trajectory evaluation (STE) is a formal verification method that can be viewed as a hybrid between a symbolic model checker and a symbolic simulator [6]. As a simulator, it can compute the result of executing a circuit with concrete boolean test vectors as inputs; as a *symbolic* simulator, it can compute symbolic expressions giving outputs as a function of arbitrary inputs; as a model-checker, it can automatically check the validity of a simple temporal logic formula for arbitrary inputs—computing an exact characterization of the region of disagreement in case the formula is not unconditionally satisfied. STE's seamless connection between simulation and verification is crucial to satisfying our methodology's requirement for early results.

STE performs model checking with a symbolic simulation-based algorithm that is significantly more efficient than more traditional symbolic model-checking approaches. STE is particularly well suited to handle datapath properties, and

it is used in our work to verify gate-level models against more abstract reference models. However, since symbolic trajectory evaluation is based on BDDs, there are circuit structures that cannot be handled monolithically. For example, multipliers are beyond the capability of STE alone.

Theorem Proving

Because model checkers are incapable of verifying certain circuit structures, and in any case have inescapable capacity limits, most verifications must be broken up into pieces. A decomposition typically either follows the structure of the circuit or partitions the input data space (though other problem-reduction strategies are also possible). To ensure that no mistakes are made in this partitioning process and to check that no verification conditions are forgotten, a mechanically-checked proof is needed.

In Forte, a lightweight theorem proving system called ThmTac is used for this task. It is implemented in FL and tightly integrated with the Forte model checkers; as a result, no translation or re-formulation of the verification results is needed before theorem proving can be performed. ThmTac is loosely based on the classic model of LCF proof systems [12]. It consists of an FL implementation of a Church-style formulation of higher-order logic, with an extensible theorem-proving infrastructure built on top.

Though the mechanism of lifted FL, the logical language of Forte actually shares an implementation of the underlying λ-calculus data structures with FL programs themselves. This means that lifted FL also allows the FL evaluation machinery to be invoked as an inference of the logic; essentially, any phrase that evaluates to true in FL can be 'lifted' to be a theorem in the logical language. In particular, a successful STE (or CTL) model-checking run in Forte is just a specific kind of function call that evaluates to true in FL. Any such model-checking run can therefore can be lifted to a higher-order logic theorem asserting its logical content, providing our methodology with a very smooth and flexible link between model checking and theorem proving.

On top of this language infrastructure, ThmTac has numerous FL libraries implementing our own versions of the conventional theorem-proving technology—tactics and tacticals, term rewriting, and so on. The ThmTac theorem-prover also has several special-purpose, integrated decision procedures, making reasoning (particularly about arithmetic results) significantly easier and more efficient. For ease of use and proof robustness, it also supports the 'declarative' proof-construction style [15].

4 Verifying a Floating Point Adder

In this section we will illustrate the application of our methodology in the verification of a floating-point adder. The adder performs IEEE-compliant floating-point addition and subtraction at single, double, and extended precisions, and

supports four rounding modes—toward 0, toward $-\infty$, toward $+\infty$, and to nearest. It was verified as part of a large-scale verification effort undertaken on the Intel Pentium® Pro processor [4].

Sections 4.1 to 4.4 show how the adder verification proceeded through the four phases of our methodology—understanding the circuit and making it 'wiggle', targeted scalar verification, symbolic model checking, and theorem proving.

4.1 Wiggling

The verification effort began with education about the circuit and its operating environment. This was accomplished by reading design documentation, perusing the RTL code, consulting other verification and validation engineers, and consulting the circuit designers. Notes from this activity evolved into an initial sketch of the circuit API, specifying the names and timing of circuit signals that are important to the verification.

Once an initial set of important signals has been identified, the process of getting the circuit up and *wiggling* begins, based on the use of symbolic trajectory evaluation as a scalar simulator. The aim of this is to develop an FL circuit API that acts as a kind of simulation 'test harness' for driving input signals and observing output signals. Developing this artifact involves discovering a mass of detail about the input and output protocols, including the exact timing of important signals, and encapsulating this in cleanly-structured FL code.

For the floating-point adder, the starting point was a small set of simulations on very simple input cases (for example, computing zero plus zero). The aim was to derive information about the relationship between the control inputs, state nodes, and outputs of interest to supplement what was available in the design documentation. Wiggling is a highly interactive process, with information being gleaned by analyzing circuit behavior with a waveform viewer, viewing circuit structure with a graphical browser, and writing FL functions and predicates that probe the circuit.

After it was understood how to drive the input and state signals in such a way as to reliably cause defined (non-X) values to appear at the outputs, we refined the API in order to reduce the number of signals that are driven and the length of time that they were driven. The aim was to make fewer assumptions about the input behavior, in order to strengthen the eventual verification result. Like the initial phase of wiggling, identifying the weakest necessary assumptions is an iterative process that benefits from further discussion with designers and architects, and study of the behavior of the circuit.

The final circuit API for the adder consists of two FL functions, one for inputs and one for outputs. The first function, `fadd_fsub_protocol`, describes the behavior of the circuit's inputs during an addition or subtraction operation. It takes the following arguments: `uop`, the bit-vector opcode of the instruction to be executed (`FADD` or `FSUB` in this example); `pc` and `rc`, the precision and rounding mode of the operation; and `A` and `B`, the two floating-point operands.

```
let fadd_fsub_protocol [uop, pc, rc, A, B] =
  // Drive clocks and resets
    gen_clocks and no_resets and
  // Opcode and inputs
    ((  (opcod isv uop) and   // Opcode
        (opcodv is T) and     // Opcode is valid
        (s1 isv A) and        // Operand 1
        (s1v is T) and        // Operand 1 is valid
        (s2 isv B) and        // Operand 2
        (s2v is T))           // Operand 2 is valid
     in_cycle 2) and
  // Round and precision control follow inputs by one cycle
    ((  (roundc isv rc) and   // Rounding mode
        (precc isv pc))       // Precision control
     in_cycle 3);
```

Within this API, we impose the conditions that the circuit is clocked correctly and that there is no reset during the execution of an operation. The function also drives the circuit's input nets with the opcode, the two input operands, and the rounding and precision controls—each in the correct clock cycle. Inside the functions called in the above definition is a mapping between signal names in FL and the actual node names in the source RTL, as well as a layer of code that establishes a unit-delay temporal abstraction.

The second API function, `fadd_fsub_result`, just observes the output `wbdata` at the appropriate clock cycle. It also asserts that the valid bit `wbdatav` is set.

```
let fadd_fsub_result [res] =
  ((  (wbdata isv res) and
      (wbdatav is T))
   in_cycle 5);
```

Once the API has been defined and validated, it establishes an abstract view of the circuit's I/O interface upon which the rest of the verification can build. This clean structuring mechanism helps makes higher levels of the proof effort reusable. And because the various mappings in the API are realized as FL source code, they can easily be inspected and understood if necessary—supporting the transparency requirement of our methodology.

4.2 Targeted Verification

Targeted verification begins essentially with regression testing, using the simple input cases from the wiggling phase. The difference between wiggling and verification is that verification entails checking the circuit against a specification. The objectives of this stage are a refined circuit API and a specification of the circuit functionality.

The specification to be developed is incorporated into the output part of the API in the previous section. For the adder, the function `fadd_fsub_result`

was revised to take as arguments `fspec`, a specification of the function to be performed by the datapath, and the same five input values as the input API.

```
let fadd_fsub_result fspec [uop, pc, rc, A, B] =
  // fspec maps inputs -> result
  let res = fspec [uop, pc, rc, A, B] in
  (wbdata isv res) and
  (wbdatav is T)
  in_cycle 5;
```

Based on the output values supplied, the revised API function uses `fspec` to compute the expected result `res`, and asserts that whatever is observed on the circuit's output signals must be identical to this.

Note that `fadd_fsub_result` is a higher-order function; its argument `fspec` is itself a function. The revised API makes no assumptions about the particular computation done by the specification; it could therefore be reused as the API for operations other than addition and subtraction. The use of a full-fledged programming language is what enables clean abstractions like the circuit API to be constructed.

Our specification of floating-point addition and subtraction was a straightforward adaptation of a textbook algorithm [16]. Initially, our specification supported double precision operations and rounding toward zero only, and was tested only for 'true' addition (that is, addition of operands with like signs, or subtraction of operands with different signs). Through verification aimed at particular corner cases, our specification was extended to support single, double and double-extended precisions, and four rounding modes.

Our final specification was the composition of five functional stages, shown schematically in Figure 2. In the first stage, the operands are inspected and the mantissa of the smaller operand is shifted right in preparation for addition. Addition or subtraction of the mantissas takes place in the second stage. Following addition, the mantissa is normalized—shifted left or right to align its significant bits with the binary point. Finally, the mantissa is rounded and truncated according to the given rounding mode and precision and renormalized if required.

We will not show the complete specification here, but the rounding function `RND` is illustrative of how a specification evolves in this phase of the methodology. The FL function `RND` takes as arguments the sign bit `s` and significand (or 'mantissa') `sgf` of the number to be rounded, as well as the precision and rounding controls:

```
let RND pc rc s sgf =
  // Extract lsb, guard, round sticky bits.
     let L = Lsb pc sgf in
     let G = Guard pc sgf in
     let RS = RoundS pc sgf in
  // Conditionally add one to LSB
     let rbit =
```

```
      (rc '=' TO_ZERO)    => F
    | (rc '=' TO_POS_INF) => ((NOT s) AND (G OR RS))
    | (rc '=' TO_NEG_INF) => (s AND (G OR RS))
    | (rc '=' TO_NEAREST) => (RS => G | (L AND G))
    | F in
// Result truncates mantissa to precision specified
// by pc, adds rbit and pads result with zeros.
   Result rbit pc sgf;
```

In our initial specification, `RND` simply truncated its result to the required number of bits (rounding toward zero). Later, as the other rounding modes were added, the concepts of guard and sticky bits were introduced and used. Each extension was justified by a particular set of test inputs.

For example, $1.0 \times 2^{30} - 1.0 \times 2^{0}$, in single precision, should yield 1.0×2^{30} when rounded toward $+\infty$ and $1.11111111111111111111111 \times 2^{29}$ when rounded toward $-\infty$. The correct behavior for such cases was usually deduced using pencil and paper, and was always checked against the actual behavior of the circuit. As a guard against the danger of inadvertently incorporating circuit bugs in our specification, we will later verify the specification itself against the behavior required by the IEEE 754 standard. Section 4.4 describes this aspect of the verification.

Because our specification is written in FL, it is fully executable. It also contains no temporal information, mentions no signal names from the RTL description, and says nothing about interface protocols. The circuit API supplies this information for a given implementation of the floating-point addition/subtraction algorithm. Decoupling the specification of the datapath functionality (which can remain constant for the life of a design) from the names of signals and their timing (which can change from week to week in a design project) supports our requirements of regression and reuse. Again, the use of FL is a key enabler.

Work in this phase is directed towards defining and debugging a specification, and so involves more thinking about the actual computation than in the wiggling phase. It still consists mostly of simulation, but more time is spent analyzing simulation results from the circuit and specification and less examining the internals of the circuit. When it becomes time consuming or difficult to find stimuli that result in a discrepancy between the circuit and specification, it is time to move on to symbolic model checking.

4.3 Symbolic Model Checking

The symbolic model checking phase marks the beginning of serious formal verification efforts. At this point symbolic Boolean (BDD) values are declared for each circuit input:

```
let OPCOD = variable_vector "opcod[opcod_w:0]";
let PC    = variable_vector "pc[1:0]";
let RC    = variable_vector "rc[1:0]";
```

```
let A     = variable_vector "A[fp_w:0]";
let B     = variable_vector "B[fp_w:0]";
```

Our eventual aim in this phase is to drive all input signals with symbolic Boolean values, and thus obtain exhaustive confirmation that the circuit conforms to its specification. However, STE also allows an arbitrary mix of scalar and symbolic simulation values, and so satisfies the methodology requirement for incrementality.

This phase brings with it two major challenges, the first of which is the capture of input validity conditions. Many circuits impose constraints on their environments and are guaranteed to work correctly only if those conditions are met. For example, circuits associated with decoding instructions may require that their inputs are legal instructions, or a non-pipelined execution unit may require that there is a certain latency between consecutive operations.

The floating-point adder also imposes constraints on its environment; each of its operands must either be zero (all exponent and mantissa bits zero) or normal (with exponent between 0 and maximum, and the mantissa of the form $1.b_1b_2b_3\ldots$). The FL function `VALID_FP` expresses this constraint:

```
let isNORM fp   = (exp fp '> 0) AND (exp fp '< MAX_EXP) AND
                  (Jbit fp);
let isZERO fp   = (exp fp '= 0) AND (man fp '= 0);

let VALID_FP fp = (isZERO fp) OR (isNORM fp);
```

In addition to data validity conditions, a huge variety of other conditions will typically be used to restrict the input and state spaces for a verification. These include isolation of the case that is being executed, cases that are believed to be buggy, and cases that are not yet included in the specification.

For the floating-point adder verification, we are interested only in the response of the circuit to `FADD` and `FSUB` opcodes. The validity condition therefore restricts the opcode, as well as ensures each operand is either zero or normal:

```
let Add = (OPCOD = FADD);
let Sub = (OPCOD = FSUB);

let ValidInput = (Add OR Sub) AND VALID_FP A AND VALID_FP B;
```

FL allows a user to implement whatever specialized 'vocabulary' of functions is needed to express input constraints like these in the most natural way. Here we have used FL to define a concise vocabulary for the domain of floating-point verification. This supports our goal of transparency, as definitions like `VALID_FP` and `ValidInput` can easily be inspected and understood. Definitions like these also provide infrastructure that all users can employ in later verification efforts.

The second major challenge in this phase is complexity management. Once symbolic model checking begins in earnest, BDD sizes will begin to exceed the capacity limits of the tool. Some reduction in BDD sizes can be obtained by

choosing good variable orderings, but a top-level strategy to combat verification complexity becomes crucial. For the floating-point adder, we divided the verification into numerous subcases—initially according to the whether the mantissa datapath performs addition or subtraction, and then according to the difference between the two exponents. The following definitions characterize the domain of operation of the mantissa datapath as 'true addition' or 'true subtraction':

```
let EqualSigns = ((sgn A) = (sgn B));

let TrueAdd =
  ValidInput AND
  ((Add AND EqualSigns) OR (Sub AND NOT EqualSigns));

let TrueSub =
  ValidInput AND
  ((Add AND NOT EqualSigns) OR (Sub AND EqualSigns));
```

In each domain of operation, we further decompose the model-checking problem according to the exponent difference. As an example, for 'true addition' we consider the following cases:

1. The exponent of operand `B` is much larger than the exponent of operand `A` (that is, the magnitude of operand `A` is almost negligible).
2. The exponent of operand `B` is larger than the exponent of operand `A` by 1, 2, ..., n; where n is the mantissa width.
3. The exponents of the operands are equal (only one case).
4. The exponent of operand `A` is larger than the exponent of operand `B` by 1, 2, ..., n; where n is the mantissa width.
5. The exponent of operand `A` is much larger than the exponent of operand `B`.

We compute the cases systematically in FL, as well as describe them in a transparent form, by defining auxiliary functions capturing relations between operand exponents. For example, `Exp1BiggerBy` expresses the condition that the exponent of operand `A` is larger by `n` than the exponent of operand `B`. `Exp1TooBig` captures the condition that the magnitude of operand `A` is almost negligible.

```
let Exp1BiggerBy n = ((exp A '-' exp B) '=' (nat_to_bv n));

let Exp1TooBig =
   ((exp A '>' exp B) AND                        // No wraparound
    ((exp A '-' exp B) '>' (nat_to_bv max))); // Vin1 >> Vin2
```

Each true addition case is then generated by an FL function `true_add_case` using these functions. Every case imposes the restriction `TrueAdd`, together with a condition generated from an integer that quantifies the exponent difference (and is also used later to compute an appropriate BDD variable ordering.) The function `ExpDiff` constructs an individual case from this integer. Finally, the list `true_add_cases` enumerates all the true addition cases.

```
let ExpDiff n =
  (n < ~max) => Exp2TooBig |
  (n >= ~max AND n < 0) => Exp2BiggerBy (~n) |
  (n = 0) => EqualExps |
  (n <= max AND n > 0) => Exp1BiggerBy n |
     Exp1TooBig;

let true_add_case n = (TrueAdd AND ExpDiff n, n);

let true_add_cases = map true_add_case (~max-1 upto max+1);
```

The FL source code for the case decomposition is a major artifact produced in this phase. For the floating-point adder, the case analysis given above follows an input space decomposition strategy; it is therefore specific to the algorithm, but is *not* dependent on fine details of circuit structure (which are hidden by the API). This artifact is therefore highly reusable in other settings that employ roughly the same algorithm.

Throughout this phase, model-checking is invoked through an FL verification function built on top of the circuit API. The function `prove` verifies one case, and is supplied with an input validity condition and an ordering parameter (the difference between the exponents of the operands).

```
let prove (vc, n) =
// Drive the inputs
   let ante = fadd_fsub_protocol [uop, pc, rc, in1, in2] in
// Specification's idea of the output
   let cons = fadd_fsub_result fadd_fsub [uop,pc,rc,in1,in2] in
// Install ordering, then run symbolic trajectory evaluation
   (Order n) fseq (STE_vc ckt vc ante cons);
```

This is where the circuit API interacts with the specification; the API function `fadd_fsub_protocol` computes the antecedent (stimulus) for a symbolic trajectory evaluation run, and the API function `fadd_fsub_result` is called with the datapath specification `fadd_fsub` to compute the consequent (expected result). `STE_vc` performs symbolic trajectory evaluation upon the circuit `ckt`, incorporating the supplied condition `vc` and the computed antecedent `ante` and consequent `cons`. The function `Order` generates and installs a BDD variable ordering, based on the parameter `n` (the difference between the exponents of the two operands being added or subtracted). The ordering is chosen to mimic the alignment of mantissa bits that will take place in computing the sum or difference of the operands.

Verification of all the true addition cases is accomplished by applying `prove` to the list of all cases:

```
let true_add_result = map prove true_add_cases;
```

`true_add_result` is a list of boolean values, each signifying success or failure of a particular case. In practice, since the cases are independent, we verify them in parallel on a network of workstations.

The case breakdown for true subtraction is similar, with the complication that additional case splits are needed when the exponents are equal or differ by 1. In these cases, the difference between the two mantissas may have many leading zeros and a large left shift may needed to renormalize. The true subtraction cases are generated with the help of further auxiliary functions in FL.

As verification proceeds in this phase, disagreements between the circuit and the specification are increasingly likely to indicate bugs in the circuit rather than the specification. Of course, subtle bugs may still lurk in the specification. For example, in model-checking the true subtraction cases we discovered a discrepancy between the specification and the circuit when the exponents differed by one and the difference between aligned mantissas was the smallest non-zero value. In this corner case, our specification incorrectly normalized the result before rounding; we corrected this by adding one extra bit to its internal mantissa.

4.4 Theorem Proving

The final, theorem proving, phase of the methodology is relatively open-ended and can include many possible activities. The most common activity is checking the validity of the problem decomposition devised in the model checking phase and ensuring completeness of coverage. Of course, a certain amount of simple checking can be done directly in FL. For example, we can compute in FL a list of all cases verified in the floating-point adder example:

```
let cases = true_sub_cases @ true_add_cases;
```

Then to check that the list is exhaustive, we just compute in FL a boolean (BDD) value `cases_exhaustive` as follows:

```
let all_cases_disjunction = OR_list (map fst cases);

let cases_exhaustive =
  (ValidInput AND (Add OR Sub)) ==> all_cases_disjunction;
```

The logical implication calculated here states that, under the assumptions the inputs are either normal or zero and the operation is legal, the disjunction of the list of all cases comes out true.

We use such mechanized checks because, with complicated specifications and environmental constraints, even painfully detailed code reviews will often miss important errors. In one verification script, we discovered quite late that we had duplicated an opcode in the instruction map, and so were not verifying one type of instruction. In another verification, a simple typo in an environmental constraint caused us to overlook an entire class of instructions.

The theorem proving phase goes well beyond simple computation of checks in FL. It provides a high degree of confidence in the validity and coverage of a given

decomposition, and it can handle decomposition strategies not easily analyzed by just computing BDDs. Moreover, through a computational reflection technique called lifted FL [14], it can also trivially employ the kind of evaluation methods described above as a proof strategy.

The aim of theorem-proving is to use deductive proof to knit together the (possibly hundreds of) individual model checking runs generated in the previous phase into a top-level correctness statement that says that the circuit satisfies the specification for all valid input and states. In the style encouraged by our methodology, a top-level correctness statement concisely ties together the three major elements of the whole proof effort—the circuit, the validity condition on the environment, and the specification. Writing the correctness statement in a standard, stylized form helps satisfy the *transparency* requirement for our methodology. A standard form makes it easier for a reader to extract pieces of critical information and answer questions about the verification. For example, one might ask, *'Was this circuit verified for all addressing modes?'*.

Proving a top-level correctness statement does not directly find bugs in the circuit. Instead, bugs may be found in the verification—missing cases, overly tight environmental constraints, or incorrect reasoning steps. Fixing these may require changing the model-checking cases and re-running some or all of them. Of course, these new model-checking runs may then reveal bugs in the circuit.

Technically, theorem proving is seamlessly integrated with model checking in Forte using the mechanism of lifted FL, which allows any FL phrase that evaluates to true to be converted into a theorem of higher order logic. The collection of successful verifications from the model-checking phase all evaluate to true, and so we can use this mechanism to convert their FL sources into theorems. By standard (and rather trivial) reasoning in higher order logic, these theorems can then be combined into the top-level correctness statement. A side-effect will be to have checked the decomposition soundness and case coverage.

The primary artifact of this activity—arguably of the entire effort—is a top-level correctness statement. Theorem proving glues the results of multiple model checking runs together to derive this theorem, which is itself beyond the capacity of model checking. But the final phase of the methodology can also provide an independent check on the 'quality' of the specification, by deriving independent properties from it.

For example, the floating point adder specification (which is essentially an algorithm) was validated against the more abstract IEEE standard specification of floating point addition. For this proof, we divide the algorithm into two stages (Figure 2). Stage 1 includes alignment, addition/subtraction, and normalization; stage 2 is the rounder. For each stage we have assumptions about the inputs of the algorithm, properties to prove about the output, and some auxiliary properties to prove that are relied upon by subsequent stages.

The overall properties of Stage 1 are established by 'cascading' properties of the alignment, add/subtract, and normalization algorithms. Proofs of this first stage verify `p`'s relation to the inputs and, roughly speaking, conclude that

$$(|\,p\,| \leq |\,in2{+}in1\,|) \wedge (|\,in2{+}in1\,| < |\,p\,|{+}\mathrm{ulp}\,) \wedge (p_s \equiv (|\,p\,| < |\,in2{+}in1\,|))$$

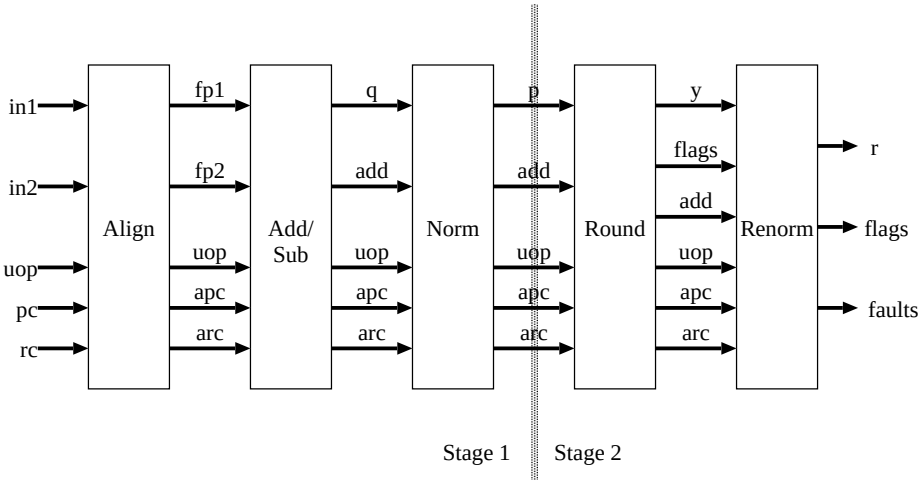

Fig. 2. Decomposition of reference model proof.

The value ulp is the distance between the rational number $|\,p\,|$ and the rational of next largest magnitude actually representable with the precision available.

The proof of stage 2 is decomposed into four cases, one for each rounding mode in the IEEE standard. The proof proceeds by assuming the above property of stage 1 and concluding an output specification for each rounding mode. For example, when rounding toward $-\infty$ we have

$$(r \leq \mathit{in2} + \mathit{in1}) \wedge (\mathit{in2} + \mathit{in1} < r + \mathrm{ulp}\,) \wedge$$
$$(s = -1 \wedge m = 2^p \supset r + \tfrac{1}{2}\mathrm{ulp}\; > \mathit{in2} + \mathit{in1})$$

This says the true sum of the inputs lies between the computed result r and the next representable rational number, as required by the IEEE standard. The third conjunct deals with an important special case. When r is negative ($s = -1$) and r's mantissa m is a power of two, then the pair of representable values on either side of r will have different exponents. The distance from r to the next largest representable value is then one-half the distance between r and the representable value of *next largest magnitude*.

The proof concluded by combining the theorem for stage 1 and the separate cases for stage 2 to derive one theorem specifying the outputs of the adder in terms of its inputs for all rounding modes (and, indeed, for several precisions).

In this example, the theorem proving phase produces two major results. First, it ensures that the decomposition from the top-level correctness statement to the model checking runs is correct. In principle, this could have been verified by an 'infinite capacity' model checker; the second result goes much further. By checking the specification against an abstract IEEE specification, theorem proving is used to derive more a obviously correct theorem—one that is beyond the expressive power of the specification language of the STE model-checker.

5 Conclusions

We have described a methodology for carrying out datapath verification on large hardware systems. The FADD example in this paper uses STE-based model checking; our framework also includes CTL-based model checking. Our verification methodology has evolved over a series of case studies carried out at Intel. These include verifications of an IA32 instruction-length decoder [10, 11] and of IEEE compliance of many of the major Intel Pentium® Pro floating point instructions [4]. The methodology is certainly not complete—industrial hardware verification is very challenging, and much further work remains in methodology and the underlying technology.

Our methodology relies heavily on the capabilities provided by the Forte system. While developing Forte, we have been conscious of the competing goals of capability and usability for the tools. We are also keenly aware that a routine verification for the technology or tool developer may be virtually impossible for others to duplicate. Our methodology aims to address these issues by targeting the usability of Forte and by providing information that will help others to use Forte successfully.

A typical introduction to Forte begins with the new user exploring conventional simulation in Forte. Once the user is comfortable with this mode of usage, symbolic simulation and model-checking are introduced. When the capacity limits of model-checking are reached, the user then learns about theorem-proving. This incremental and modular approach also has benefits in the desired usage model for Forte when verification engineers are using it on a regular basis. Given that there will be much more model-checking than theorem-proving, a validation team composed of a few theorem-proving experts and a larger number of model-checking experts can divide up the verification tasks.

Our methodology aims to create proof efforts that are relatively circuit-independent. This makes our proofs more robust during design evolution, as well as more reusable for future generations of designs implementing the same functionality.

Acknowledgments

Ching-Tsun Chou, Limor Fix, Brian Moore, and Eli Singerman made helpful comments on a draft of this paper. Thanks are also due to the anonymous referees for their comments.

References

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic Publishers, 2000.
[2] T. Kropf, *Introduction to Formal Hardware Verification*, Springer–Verlag, 1999.
[3] Á. Th. Eirksson, "The formal design of 1M-gate ASICs," in *Formal Methods in Computer-Aided Design*, G. Gopalakrishnan and P. Windley, Eds. 1998, vol. 1522 of *Lecture Notes in Computer Science*, pp. 49–63, Springer–Verlag.

[4] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technical Journal*, First quarter, 1999, Available at developer.intel.com/technology/itj/.

[5] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall, "Practical application of formal verification techniques on a frame mux/demux chip from Nortel Semiconductors," in *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf, Eds. 1999, vol. 1703 of *Lecture Notes in Computer Science*, pp. 110–124, Springer–Verlag.

[6] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.

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

[8] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

[9] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[10] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of boolean constraints," in *ACM/IEEE Design Automation Conference*, 1999.

[11] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *ACM/IEEE Design Automation Conference*, 1998, pp. 538–541.

[12] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[13] L. Augustson, "A compiler for lazy-ml," in *ACM Symposium on Functional Programming*, 1984, pp. 218–227.

[14] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Lifted-fl: A pragmatic implementation of combined model checking and theorem proving," in *Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds. 1999, vol. 1690 of *Lecture Notes in Computer Science*, pp. 323–340, Springer–Verlag.

[15] D. Syme, "Three tactic theorem proving," in *Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds. 1999, vol. 1690 of *Lecture Notes in Computer Science*, pp. 203–220, Springer–Verlag.

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