Chapter 1
Use of Formal Verification at Centaur Technology

Warren A. Hunt, Jr., Sol Swords,
Jared Davis, Anna Slobodova

1.1 Introduction

We have developed a formal-methods-based hardware verification toolflow to help ensure the correctness of our X86-compatible microprocessors. Our toolflow uses the ACL2 theorem-proving system as a design database and a verification engine. We verify Verilog designs by first translating them into a formally defined hardware description language, and then using a variety of automated verification algorithms controlled by theorem-proving scripts.

In this chapter, we describe our approach to verifying components of VIA Centaur's 64-bit Nano, X86-compatible microprocessor. We have successfully verified a number of media-unit operations, such as the packed addition/subtraction instructions. We have verified the integer multiplication unit, and we are in the process of verifying microcode sequences that perform arithmetic operations.

1.1.1 Overview of Verification Methodology

In our verification process, we first translate the Verilog RTL source code of Centaur's design into `EMOD`, a formally defined HDL. This process captures a design as an ACL2 object that can be interpreted by an ACL2-based HDL simulator. The HDL simulator is used both to run concrete test cases and to extract symbolic representations of the circuit logic of blocks of interest. We then use a combination of theorem proving and equivalence checking to prove

Centaur Technology
7600C North Capital of Texas Hwy
Austin, TX 78731
e-mail: {hunt,sswords,jared,anna}@centtech.com

that the functionality of the circuit in question is equivalent to a higher-level specification. A completed verification yields an ACL2 theorem that precisely states what we have proven.

We have developed a deep-embedding of our hardware description language, `EMOD` [13], in the ACL2 logic. We describe the `EMOD` language in Section 1.4.1. Our implementation includes a syntax checker for well-formed `EMOD` modules and an interpreter that gives meaning to such modules. The `EMOD` interpreter can operate in several different modes to perform concrete or symbolic simulations, analyze dependencies, and estimate delays. Simulations may use either a Boolean or four-valued logic mode, and symbolic simulations may use either binary-decision diagrams (BDDs) [7] or and-inverter graphs (AIGs) as the representation for symbolic bits. We believe our approach to representing the hardware design reduces the risk of translation errors, since we may perform co-simulation between Verilog and `EMOD` to ensure the veracity of the translation. We can also translate the design as represented in the `EMOD` language back to Verilog.

Our Verilog translator consists of a parser and a series of code transformations that simplify the design until it can be be easily translated into the `EMOD` language, which lacks features such as continuous assignments and always blocks. We describe the translator in Section 1.2.

To prove that output from a hardware simulation is equivalent to that produced by a specification function, we produce BDDs representing both the hardware and the specification outputs, and compare them for equivalence. We use case-splitting to avoid certain BDD-size explosions. We sometimes use AIGs as an intermediate form before creating the BDDs to avoid size explosions. To produce ACL2 theorems using these methods, we have created a verified symbolic execution framework that uses these procedures. We describe our proof methodology in Section 1.3.

1.1.2 Timeline

The integration of formal methods into Centaur's design methodology has been ongoing for several years. Hunt first met with Centaur representatives in April, 2007. This led to Hunt and Swords to joining Centaur in June of 2007, to see if our existing (ACL2-based) tools could be usefully deployed on Centaur verification problems. Our use of formal methods is not new, and AMD [19] has been using ACL2 for many years for floating-point hardware verification. However, there are several things that differentiate our effort from all others: the Centaur design is converted into our `EMOD` formalized hardware description language (described later), our verification (BDD and AIG) algorithms are themselves verified, and all of our claims are all checked as ACL2 theorems.

The use of formal methods to aid hardware design has been ongoing for many years. Possibly the earliest adopter was IBM with equivalence checking mechanisms that they developed in the early 1980s; IBM protected these mechanisms as trade secrets. With the development of simple microprocessor verification examples, such as the FM8501 [10] and the VIPER [6], and introduction of BDDs [7], commercial organizations started integrating some use of formal methods into their design flow. A big impetus for the use of formal methods came from the Intel FDIV bug [18].

Work that allowed us to get an immediate start was just being finished when our Centaur-based effort began. Boyer and Hunt had implemented BDDs [4, 5] with an extended version of ACL2 that included unique object representation and function memoization [4]. Separately, Hunt and Reeber had previously embedded the DE2 HDL into ACL2 [11], and this greatly influenced the development of the `EMOD` HDL.

Our initial efforts were directed along two fronts: analyzing microcode for integer division and verifying the floating-point addition/subtraction hardware. Our analysis of the microcode for the integer divide algorithm involved creating an abstraction of the microcode with ACL2 functions and then using the ACL2 theorem-prover to mechanically check that our model of the divide microcode computes the correct answer. This effort discovered an anomaly that was subsequently corrected.

Our work on the verification of the floating-point addition/subtraction hardware was much more involved. Because of the size of the design—some 34,000 lines of Verilog—it was necessary for us to create a translator from Verilog into our `EMOD` hardware description language. We enhanced a Verilog parser, written by Terry Parks (of Centaur), so that it emitted an `EMOD`-language version of the floating-point hardware design; this translator created an `EMOD`-language representation of the entire module hierarchy, including all interface and wire names. The semantics of the `EMOD` language are given by the `EMOD` simulator which allows an `EMOD`-language-based design to be simulated or symbolically simulated with a variety (e.g., BDDs, AIGs) of mechanisms. Simultaneously, we developed an extension to ACL2 that provides a symbolic simulator for the entire ACL2 logic; this system was called **G**. Given these components, we were able to attempt the verification of Centaur's designs; this was done by comparing the symbolic equations produced by the `EMOD` HDL symbolic simulator to the equations produced by the **G**-based symbolic simulation of our ACL2 floating-point specifications.

Our verification of Centaur's floating-point addition/subtraction instructions led to the discovery of two design flaws: for two of the four floating-point adders, the floating-point control flag inputs arrived one cycle early; and for one pair of 80-bit numbers (described more fully later) the sum/difference was incorrect. Both of these very subtle problems were fixed. This work was completed within the first year of our efforts at Centaur. This effort strained our Verilog translator and illuminated areas where we wanted to better integrate symbolic simulation into the ACL2 system.

In the summer of 2008, Davis arrived and began developing a more capable Verilog translator named VL. The new translator was itself written in ACL2, and it was designed with simplicity and assurance in mind. The translator has provisions for translating Verilog annotations and property specifications into the `EMOD` language.

Starting in the summer of 2008, Swords began an effort to build a verified version of the ACL2 **G** symbolic simulator, called **GL** (for **G** in the **L**ogic). This new system represents symbolic ACL2 expressions as ACL2 data objects, which allows proofs to be carried out which show such objects are manipulated correctly.

In the fall of 2008, Slobodova joined Centaur as manager of the formal verification team, and began using these tools to verify a number of different (integer, floating-point) multiplier implementations. These multipliers are actually quite complicated as they can be reconfigured on a clock-by-clock basis to create different (e.g., four 32x32-bit or one 64x64-bit) multipliers. The verification of the multipliers has stressed the capacity of our tools in a variety of ways and this effort has led to many improvements in capacity and speed.

By the spring of 2009, the outcome of the two efforts mentioned above resulted in the replacement of our original prototype Verilog translator with VL, and the replacement of the **G** system with **GL**, a verified symbolic simulator for ACL2 functions. All of our proofs are now carried out using these new tools.

1.1.3 Centaur Media Unit

As an example to illustrate our methodology, we will discuss our verification of the floating-point addition instructions implemented in the media unit of Centaur's CPU design. The part of the media unit that handles floating-point addition and subtraction is called the **fadd** unit; this unit is highly optimized for low latency arithmetic operations and implements SIMD 32- and 64-bit floating-point additions as well as scalar X87 80-bit floating point additions. All floating-point addition operations are performed with a two-cycle latency; the **fadd** unit can also forward results internally so that operations may be chained.

The **fadd** RTL-level design is composed of 680 modules, which we convert from Verilog into our `EMOD` hardware description language; it is this `EMOD` form of Centaur's design that we subject to analysis. The physical implementation is composed of 432,322 transistors, almost evenly split between PMOS and NMOS devices. This represents less than 5% of the total transistors in the implementation, but its 33,700 line Verilog description represents more than 6% of the CN design specification. The **fadd** unit has 374 output signals

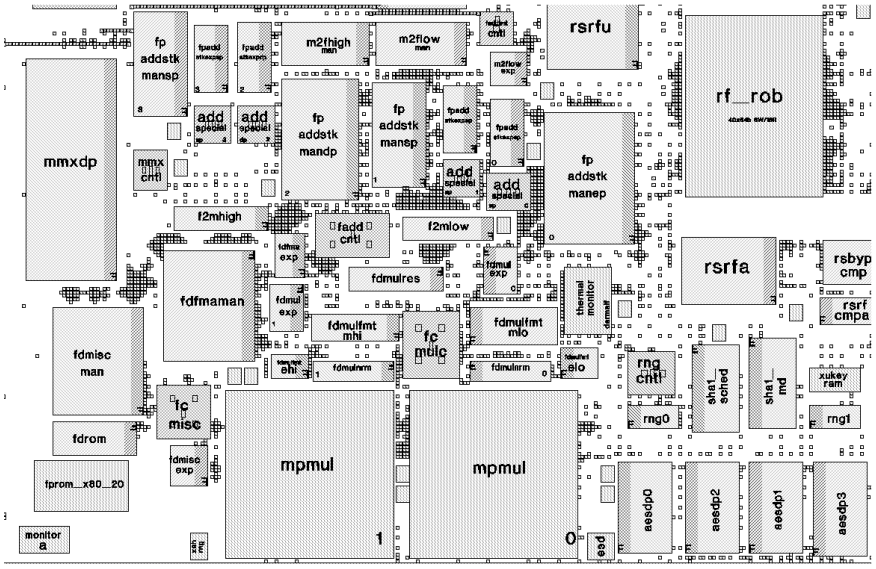


Fig. 1.1 Floorplan of CN Media Unit

and 1074 inputs including 26 clock inputs. Multiple clock inputs are used to manage power usage.

The **fadd** unit is composed of four adders: two 32-bit units, one 64-bit unit, and one 80-bit unit (see Fig. 1.2). When a 32-bit packed addition is requested, all four units are used, and the 64-bit and 80-bit adders each take 32-bit operands and produce a 32-bit results. When a 64-bit packed addition is requested, the 64-bit and 80-bit adders each take 64-bit operands and produce a 64-bit result. The **fadd** unit can only add one pair 80-bit operands per clock cycle. Other combinations are possible when a memory-resident operand is added to a register-resident, X87-style, 80-bit operand; the **fadd** unit also manages such X87-mode, mixed-size addition requests.

There are multiple paths through the addition logic that operate in parallel. The relevant path for a particular pair of operands is determined by characteristics such as the operand types (NaN, zero, denormal, etc.) and their sign and exponent bits, and the result from that path is selected as the result of the addition.

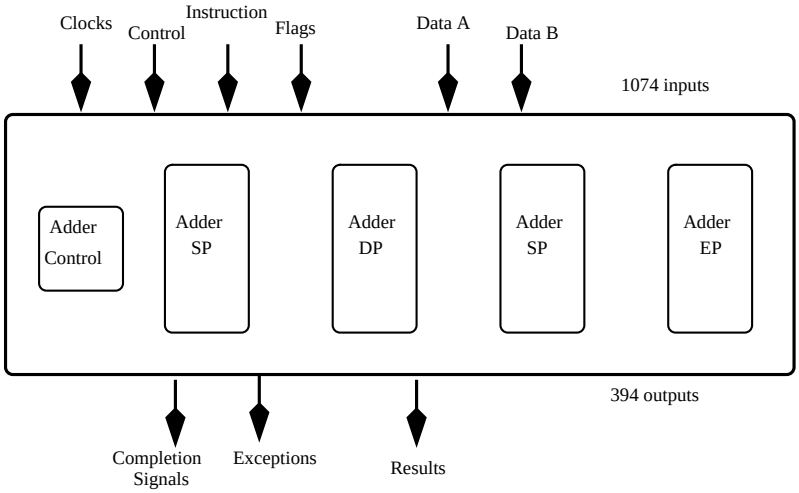


Fig. 1.2 Adder Units Inside of **fadd**

1.2 Modeling Effort

The specification of the CN processor consists of over half a million lines of Verilog; this Verilog is frequently updated by the logic designers. To bring this design into our `EMOD` HDL, we have developed a translator named VL. This is a challenge since Verilog is such a large language with no formal semantics. Our work is based on the IEEE Verilog 1364-2005 standard [1], and we do not yet support the SystemVerilog extensions. This standard usually explains things well, but sometimes it is vague; in these cases, we have carried out thousands of tests and attempted to emulate the behavior of Cadence's Verilog simulator.

VL needs to produce a "sound" translation or our verification results may be meaningless. Because of this, we have written VL in the purely-functional programming language of the ACL2 theorem prover, and our emphasis from the start has been on correctness rather than performance. For instance, our parser is written in a particularly naive way: to begin, each source file is read, in its entirety, into a simple list of *extended characters*, which associate each character with its filename and position. This makes the remaining steps in the parsing process ordinary list-transforming functions,

- read : filename → echar list
- preprocess : echar list → echar list
- lex : echar list → token list
- eat-comments : token list → token list × comment map
- parse : token list → module list

The parser itself is written in a conventional, recursive-descent style, which is a good match for Verilog since keywords like `module`, `assign`, etc., usually say what comes next. Since the entire list of tokens has been computed before parsing begins, we can take advantage of arbitrary look-ahead, and backtracking is completely straightforward.

This simple-minded approach lends itself well to informal validation. For instance, since we actually construct each intermediate list, we can add assertions relating them to one another, e.g., we can test that flattening the parsed input is equal to the original input. Since our functions operate on lists, instead of files, it is very easy to write unit tests directly in our source code, and we have developed a number of these tests. Furthermore, since these routines are written in the ACL2 theorem prover, we can actually prove some theorems about the parser, e.g., on success it produces a list of syntactically well-formed modules.

1.2.1 Conversion to the `EMOD` *Language*

To implement the translation into `EMOD`, we adopt a program-transformation-like [23] style: to begin with, the entire parse tree for the Verilog sources is constructed; we then apply a number of rewriting passes to the tree which result in simpler Verilog versions of each module. The final conversion into `EMOD` is really almost incidental, with the resulting `EMOD` modules differing from our most-simplified Verilog modules only in syntax.

Each transformation tends to be fairly short and easy to understand, and can be studied in isolation, either informally or with the theorem prover. Since each rewriting pass produces well-formed Verilog modules, we can simulate the original and simplified Verilog modules against each other, either at the end of the simplification process or at some intermediate point.

We can also run a number of common sanity checks after each rewrite to catch any gross errors. These sorts of checks serve to answer questions such as:

- Are only supported constructs used?
- Are only defined modules instanced?
- Is each module's namespace free of collisions?
- Are the ports compatible with the port declarations?
- Are the port and wire declarations compatible?
- Have we determined the size of every declaration?
- Have the widths and signs of all expressions been determined?
- Are the widths of the arguments to every submodule correct?
- Are the indices for every bit- and part-select in bounds?

These sorts of checks are often useful as "guards" (preconditions) for our transformation steps. In a few cases, we also prove, using ACL2, that some

of these properties will be satisfied after a certain transformation is run. But usually we do not try to do this because it is easier to just run the checks after each transformation.

We now present an overview of our transformation sequence.

1.2.1.1 Unparameterization.

Verilog modules can have parameters, e.g., an `adder` module might take input wires of some arbitrary `width`, and other modules can then instantiate `adder` with different `width`s, say 8, 16, and 32. Our first transformation is to eliminate parametrized modules, e.g., we would introduce three new modules, `adder$width=8`, `adder$width=16`, and `adder$width=32`, and change the instances of `adder` to point to these new modules as appropriate.

1.2.1.2 Declaring Implicit and Port Wires.

Verilog permits undeclared identifiers to be used as one-bit wires. We would like to prohibit this to reduce the chance of typos (a la Perl's *use strict*), but this idea is unpopular so we only issue warnings. In this transformation, we add a `wire` declaration for each undeclared wire and each port which has been declared to be an `input` or `output` but which has not also been declared as a `wire`.

1.2.1.3 Standardizing Argument Lists.

Modules may be instantiated using either positional or named argument lists. For instance, given a module `M` with ports `a`, `b`, and `c`, the following instances of `M` are equivalent:

```
M my_instance(1, 2, 3);
M my_instance(.b(2), .c(3), .a(1));
```

In this transformation, we convert all instances to the positional style and annotate the arguments as inputs or outputs.

1.2.1.4 Resolving Ranges.

Wires and registers in Verilog can have widths. For instance,

```
wire [3:0] w;
```

declares a four-bit wire, `w`, whose bits are `w[3]` through `w[0]`. Unparameterization sometimes leaves us with expressions here, e.g., in the `adder` module, we might have

```
wire [width-1:0] a;
```

which, in `adder$width=8`, will become

```
wire [8-1:0] a;
```

We now resolve these expressions to constants. The specification seems vague about how these expressions are to be evaluated (e.g., with respect to widths and signedness), so we are quite careful and only allow signed, 32-bit, overflow-free computations of `+`, `-`, and `*`.

1.2.1.5 Operator Rewriting.

We can reduce the variety of operators we need to deal with by simply rewriting some operators away. In particular, we perform rewrites such as

$$\texttt{a \&\& b} \rightarrow \texttt{(|a) \& (|b)},$$
$$\texttt{a != b} \rightarrow \texttt{|(a \^{} b)}, \text{ and}$$
$$\texttt{a < b} \rightarrow \sim\texttt{(a >= b)}.$$

This process eliminates all logical operators (`&&`, `||`, and `!`), equality comparisons (`==` and `!=`), negated reduction operators (∼`&`, ∼`|`, and ∼`^`), and standardizes all inequality comparisons (`<`, `>`, `<=`, and `>=`) to the `>=` format. We have a considerable simulation test suite to validate these rewrites.

1.2.1.6 Sign and Width Computation.

We now annotate every expression with its type (sign) and width. This is tricky. The rules for determining widths are quite complicated, and if they are not properly implemented then, for instance, carries might be inappropriately kept or dropped. It took a lot of experimenting with Cadence and many readings of the standard to be sure we had it right.

1.2.1.7 Expression Splitting.

After the widths have been computed, we introduce explicit wires to hold the intermediate values in expressions,

```
assign w = (a + b) - c;
  →
wire [width:0] newname;
assign newname = a + b;
assign w = newname - c;
```

We also split inputs to module and gate instances,

```
my_mod my_inst(a + b, ...);
  →
wire [width:0] newname;
assign newname = a + b;
my_mod my_inst(newname, ...);
```

1.2.1.8 Making Truncation Explicit.

Verilog allows for implicit truncations in assignment statements; for instance, one can assign the result of a five-bit addition `a + b` to a three-bit bus (collection of wires), `w`. We now make these truncations explicit by introducing a new wire for the intermediate result, for example,

```
wire [4:0] newname;
assign newname = a + b;
assign w = newname[2:0];
```

We print warnings about such truncations since they are not good form and may point to problems.

1.2.1.9 Eliminating Assignments.

We now replace all assignments with module instances. First, we develop a way to generate modules to perform each operation at a given width, and we write these modules using only gates and submodule instances. Next, we replace each assignment with an instance of the appropriate module, e.g.,

```
assign w = a + b;
  →
VL_13_BIT_PLUS newname(w, a, b);
```

This is one of our more complicated transformations, so we have developed a test suite which, for instance, uses Cadence to exhaustively test `VL_4_BIT_PLUS` against an ordinary addition operation. We are careful to handle the X and Z behavior appropriately. We go out of our way so that all of `w`'s bits become X if any bit of `a` or `b` is X or Z, even though this makes our generated adders more complex.

1.2.1.10 Eliminating Instance Arrays.

Gate and module instances can be put into arrays,

```
and foo [13:0] (o, a, b);
```

declares fourteen and-gates. We now convert such arrays into explicit instances, such as, `foo0`, ..., `foo13`. The rules for partitioning the bits of the arguments are not too difficult.

1.2.1.11 Eliminating Higher-Arity Gates.

Primitive gate instances in Verilog can use variable-length argument lists;
`not multinot(o1, ..., on, i);`
represents a `not` gate with one input, `i`, and n outputs, $o_1, \ldots o_n$. We now split these up into lists of gates,
`not multinot_1(o1, i);`
...
`not multinot_n(on, i);`
Afterwards, each `not` and `buf` gate has one input and output, and each `and`, `or`, `nand`, `nor`, `xor`, and `xnor` gate has two inputs and one output.

We have left out a few other rewrites like naming any unnamed instances, eliminating supply wires, and some minor optimizations. But the basic idea is that, taken all together, our simplifications leave us with a new list of modules where only simple gate and module instances are used. This design lets us focus on each task separately instead of needing to consider all of Verilog at once.

1.2.2 Modeling Flow

It takes around twenty minutes to run our full translation process on the whole of CN. A lot of memory is needed, and we ordinarily use a machine with 64 GB of physical memory to do the translation. Not all modules can be translated successfully (e.g., because they use constructs which are not yet supported). However, a large portion of the chip is fully supported.

The translator is run against multiple versions of the chip each night, and the resulting `EMOD` modules are stored on disk into files that can be loaded into an ACL2 executable in seconds. This process also results in internal web pages that allow the original source code, translated source code, and warnings about each module to be easily viewed, and some other Lint-like reports for the benefit of the logic designers and verification engineers.

1.3 Verification Method

Our verification efforts so far have concentrated on proving the functional correctness of instructions running on certain execution units; that is, showing that they operate equivalently to a high-level specification. However, we believe our methodology would also be useful for proving non-functional properties of the design.

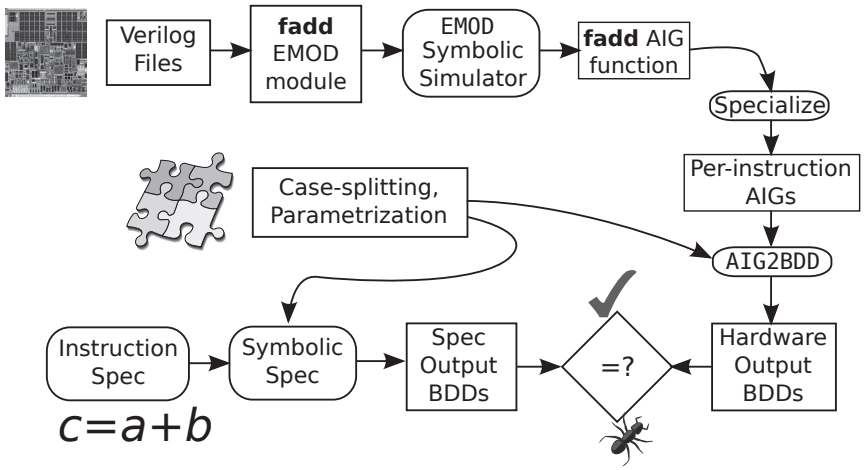


Fig. 1.3 Verification Method

Our specifications are functions written in ACL2. They are executable, and can therefore be used to run tests against the hardware model or a known implementation. In most cases, we write specifications that operate at the integer level on vectors of signals. Often these specifications are simple enough that we are satisfied that they are correct by examination; by comparison with the RTL designs of the corresponding hardware units, they are very small indeed. For floating-point addition we use a low-level integer-based specification that is somewhat optimized for symbolic execution performance and is relatively complicated compared to our other specifications. However, this specification has been separately proven equivalent to a high-level, rational-number-based specification. Before this proof was completed, we had also tested the specification by running it on millions of inputs and comparing the results to those produced by running the same floating-point operations directly on the local CPU.

Figure 1.3 shows the verification methodology we used in proving the correctness of the **fadd** unit's floating-point addition instructions. We compare the result of symbolic simulations of an instruction specification and our model of the **fadd** hardware. To obtain our model of the hardware, we translate the **fadd** unit's Verilog design into our `EMOD` hardware description language. We then run an AIG-based symbolic simulation of the **fadd** model using the `EMOD` symbolic simulator; the results of this simulation describe the outputs of the **fadd** unit as 4-valued functions of the inputs, and we represent these functions with AIGs. We then specialize these functions by setting input control bits to values appropriate for the desired instruction. To compare

these functions with those produced by the specification, we then convert these AIGs into BDDs.

For many instructions, it is feasible to simply construct BDDs representing the outputs as functions of the inputs, and we therefore may verify these instructions directly using symbolic simulation. For the case of floating-point addition, however, there is a capacity problem due to the shifted addition of mantissas. We therefore use case splitting via BDD parametrization [2,16] to restrict the analysis to subsets of the input space. This allows us to choose a BDD variable ordering specially for each input subset, which is essential to avoid this blowup. For each case split, we run a symbolic simulation of the instruction specification and an AIG-to-BDD conversion of the specialized AIGs for the instruction. If corresponding BDDs from these results are equal, this shows that the **fadd** unit operates identically to the specification function on the subset of the input space covered by the case split; otherwise, we can generate counterexamples by analyzing the differences in the outputs.

For each instruction, we produce a theorem stating that evaluation of the instruction-specialized AIGs yields the same result as the instruction's specification function. This theorem is proven using the GL symbolic simulation framework [5], which automates the process of proving theorems by BDD-based symbolic execution, optionally with parametrized case-splitting. Much of the complexity of the flow is hidden from the user by the automation provided by GL; the user provides the statement of the desired theorem and high-level descriptions of the case split, symbolic simulation inputs, and suitable BDD variable orderings. BDD parametrization and the AIG to BDD conversion algorithm are used automatically based on these parameters. The statement of the theorem is independent of the symbolic execution mechanism; it is stated in terms of universally quantified variables which collectively represent a (concrete) input vector for the design.

In the following subsections we will describe in more detail the case-splitting mechanism, the process of translating the Verilog design into an `EMOD` description, and the methods of symbolic simulation used for the **fadd** unit model and the instruction specification.

1.3.1 Case-Splitting and Parametrization

For verifying the floating-point addition instructions, we use case splitting to avoid BDD blowup that occurs due to a non-constant shift of the operand mantissas based on the difference in their exponents. By choosing case-splitting boundaries appropriately, the shift amount can be reduced to a constant. The strategy for choosing these boundaries is documented by others [2,8,15,20], and we believe it to be reusable for new designs.

In total, we split into 138 cases for single, 298 for double, and 858 for extended precision. Most of these cases cover input subsets over which the ex-

ponent difference of the two operands is constant and either all input vectors are effective additions or all are effective subtractions. Exponent differences greater than the maximum shift amount are considered as a block. Special inputs such as NaNs and infinities are considered separately. For performance reasons, we use a finer-grained case-split for extended precision than for single or double precision.

For each case split, we restrict the simulation coverage to the chosen subset of the input space using BDD parametrization. This generates a symbolic input vector (a BDD for each input bit) that covers exactly and only the appropriate set of inputs; we describe BDD parametrization in more detail in Sec. 1.4.3. Each such symbolic input vector is used in both an AIG-to-BDD conversion and a symbolic simulation of the specification. The BDD variable ordering is chosen specifically for each case split, thereby reducing the overall size of the intermediate BDDs. No knowledge of the design was used to determine the case-splitting approach.

1.3.2 Symbolic Simulation of the Hardware Model

We use the `EMOD` symbolic simulator to obtain Boolean formulas (AIGs) representing the outputs of a unit in terms of its inputs. In such simulations, we use a four-valued logic in which each signal may take values 1 (true), 0 (false), X (unknown), or Z (floating). This is encoded using two AIGs (onset and offset) per signal. The Boolean values taken by each AIG determine the value taken by the signal as in Fig. 1.4.

The **fadd** unit is mainly a pipeline, where each instruction is bounded by a fixed latency. To verify its instructions, we set all bits of the initial state to unknown (X) values – the onsets and offsets of all non-clock inputs are set to free Boolean variables at each cycle, so that every input signal but the clocks can take any of the four values. We then symbolically simulate it for a fixed number of cycles. This results in a fully general formula for each output in terms of the inputs at each clock cycle.

		Offset	
		1	0
Onset	1	X	1
	0	0	Z

Fig. 1.4

To obtain symbolic outputs for a particular instruction, we restrict the fully-general output formulas by setting control signals to the values required for performing the given instruction, and any signals we know to be irrelevant to unknown (X) input values. This reduces the number of variables present in these functions and keeps our result as general as possible. Constant propagation with these specified values restricts the AIGs to formulas in terms of only the inputs relevant to the instruction we are considering. For the floating-point addition instructions of the **fadd** unit, the remaining inputs

are the operands and the status register, which are the same as the inputs to the specification function.

The theorems produced by our verifications typically say that for any well-formed input vector, the evaluation of the instruction-specialized AIGs using the variable assignment generated from the input vector is equivalent to the output of the specification function on that input vector. Such a theorem may often be proven automatically, given appropriate BDD ordering and case-splitting, by the GL symbolic execution framework. GL has built in the notion of symbolically evaluating an AIG using BDDs, effectively converting the Boolean function representation from one form to the other. It uses the procedure `AIG2BDD` described in Sec. 1.4.4 for this process; this algorithm avoids computing certain intermediate-value BDDs that are irrelevant to the final outputs, which helps to solve some BDD size explosions.

1.3.3 Symbolic Simulation of Specification

The specification for an instruction is generally an ACL2 function that takes integers or Booleans representing some of the inputs to a block and produces integers or Booleans representing the relevant outputs. Such functions are usually defined in terms of word-level primitives such as shifts, bit-wise logical operations, plus and minus. For the floating-point addition instructions, the function takes integers representing the operands and the control register and produces integers representing the result and the flag register. It is optimized for symbolic simulation performance rather than referential clarity; however, it has separately been proven equivalent to a high-level, rational arithmetic-based specification of the IEEE floating-point standard [14]. Additionally, it has been tested against floating-point instructions running on Intel and AMD CPUs on many millions of input operand pairs, including a test suite designed to detect floating-point corner-cases [22] as well as random tests.

To support symbolic simulation of our specifications, we developed the GL symbolic execution framework for ACL2 [5]. The GL framework allows user-provided ACL2 code to be symbolically executed using a BDD-based symbolic object representation. The symbolic execution engine is itself verified in ACL2 so that its results provably reflect the behavior of the function that was symbolically executed. GL also provides automation for proving theorems based on such symbolic executions. Since these theorems do not depend on any unverified routines, they offer the same degree of assurance as any proof in ACL2: that is, they can be trusted if ACL2 itself can be trusted.

GL automates several of the steps in our verification methodology. For a theorem in which we show that the evaluation of an AIG representation of the circuit produces results equivalent to a specification function, the GL symbolic execution encompasses the AIG-to-BDD transformation and the comparison of the results, as well as the counterexample generation if there

is a bug. If the proof requires case-splitting, the parametrization mechanism is also handled by GL. The user specifies the BDD variable ordering used to construct the symbolic input vectors, as well as the case split. To specify the case split, the user provides a predicate which determines whether an input vector is covered by a given case; like the theorem itself, this predicate is written at the level of concrete objects. Typically, all computations at the symbolic (BDD) level are performed by GL; the user programs only at the concrete level.

1.3.4 Comparison of Specification to Hardware Model

For each case split in which the results from the symbolic simulations of the specification and the hardware model are equal, this serves to prove that for any concrete input vector drawn from the coverage set of the case, a simulation of the **fadd** model will produce the same result as the instruction specification. If the results are not equal, we can generate a counterexample by finding a satisfying assignment for the XOR of two corresponding output BDDs.

To prove the top-level theorem that the **fadd** unit produces the same result as the specification for all legal concrete inputs, we must also prove that the union of all such input subsets covers the entire set of legal inputs. This is handled automatically by the GL framework. For each case, GL produces a BDD representing the indicator function of the coverage set (the function which is true on inputs that are elements of the set and false on inputs that are not.) As in [8], the OR of all such BDDs is shown to be implied by the indicator function BDD of the set of legal inputs; therefore, if an input vector is legal then it is in one or more of the coverage sets of the case split.

1.4 Mechanisms used to Achieve the Verification

1.4.1 `EMOD` *Symbolic Simulator*

The `EMOD` interpreter is capable of running various simulations and analyses on a hardware model; examples include concrete-value simulations in two- or four-value mode, symbolic simulations in two- or four-value mode using AIGs or BDDs as the Boolean function representations, and delay and dependency analyses. The interpreter can also easily be extended with new analyses. The language supports multiple clocks with different timing behavior, clock gating, and both latch- and flip-flop-based sequential designs as well as implicitly clocked finite state machines. Its language for representing hardware models

```
(defm *half-adder-module*
  `(:i (a b)
    :o (sum carry)
    :occs
    ((:u o0 :o (sum)   :op ,*xor2* :i (a b))
     (:u o1 :o (carry) :op ,*and2* :i (a b)))))

(defm  *one-bit-cntr*
  `(:i (c-in reset-)
    :o (out c)
    :occs
    ((:u o2 :o out         :op ,*ff* :i (sum-reset))
     (:u o0 :o (sum c)     :op ,*half-adder-module* :i (c-in out))
     (:u o1 :o (sum-reset) :op ,*and2* :i (sum reset-)))))
```

Fig. 1.5 `EMOD` examples

is a hierarchical, gate-level HDL. A hardware model in the `EMOD` language is either a primitive module (such as basic logic gates, latches and flip-flops), or a hierarchically defined module, containing a list of submodules and a description of their interconnections. The semantics of primitive modules are built into the `EMOD` interpreter, whereas hierarchical modules are simulated by recursively simulating submodules.

A pair of small example modules, ***half-adder-module*** and ***one-bit-cntr***, are shown in Fig. 1.5. Both are hierarchically defined since they each have a list of occurrences labelled `:occs`. Connectivity between submodules, inputs, and outputs is defined by the `:i` (input) and `:o` (output) fields of the modules and the occurrences. We translate the Verilog RTL design unit into this format for our analysis.

A novel feature of our approach is that we can actually print the theorem we are checking; thus, we have an explicit, circuit-model representation that includes all of the original hierarchy, annotations, and wire names. This is different than all other approaches of which we are aware; for instance, the Forte tool reads Intel design descriptions and builds a FSM in its memory image. Our representation allows us to search the design using database-like commands to inspect our representation of Centaur's design; this explicit representation also enables easy tool construction for users as they can write ACL2 programs to investigate the design in a manner of their choosing.

1.4.2 BDDs and AIGs

BDDs and AIGs both are data objects that represent Boolean-valued functions of Boolean variables. We have defined evaluators for both BDDs and AIGs in ACL2. The BDD (resp. AIG) evaluator, given a BDD (AIG) and an assignment of Boolean values to the relevant variables, produces the Boolean

value of the function it represents at that variable assignment. Here, for brevity, we use the notation $\langle x \rangle_{\mathrm{bdd}}\,(env)$ or $\langle x \rangle_{\mathrm{aig}}\,(env)$ for the evaluation of x with variable assignment env. We use the same notation when x is a list to denote the mapping of $\langle \_ \rangle_{\mathrm{bdd}}\,(env)$ over the elements of x.

The BDD and AIG logical operators are defined in the ACL2 logic and proven correct relative to the evaluator functions. For example, the following theorem shows the correctness of the BDD AND operator (written $\wedge_{\mathrm{bdd}}$); similar theorems are proven for every basic BDD and AIG operator such as NOT, OR, XOR, and ITE:

$$\langle a \wedge_{\mathrm{bdd}} b \rangle_{\mathrm{bdd}}\,(env) = \langle a \rangle_{\mathrm{bdd}}\,(env) \wedge \langle b \rangle_{\mathrm{bdd}}\,(env)$$

1.4.3 Parametrization

BDD parametrization is also implemented in ACL2. The parametrization algorithm is described in [2]; we describe its interface here. Assume that a hardware model has n input bits. To run a symbolic simulation over all 2^n possible input vectors, one possible set of symbolic inputs is n distinct BDD variables – say, $\mathbf{v} = [v_0, \ldots, v_{n-1}]$. This provides complete coverage because $\langle \mathbf{v} \rangle_{\mathrm{bdd}}\,(env)$ may equal any list of n Booleans. (In fact, if env has length n, then $\langle \mathbf{v} \rangle_{\mathrm{bdd}}\,(env) = env$.) However, to avoid BDD blowups, we sometimes run symbolic simulations that each cover only a subset of the well-formed inputs. For each such case, we first represent the desired coverage set as a BDD p, so that an input vector $\mathbf{w}$ is in the coverage set if and only if $\langle p \rangle_{\mathrm{bdd}}\,(\mathbf{w})$. We then parametrize $\mathbf{v}$ by predicate p and use the resulting BDDs $\mathbf{v}_p$ as the symbolic inputs. These parametrized BDDs have the following important properties, which have been proven in ACL2:

- The parametrized BDDs $\mathbf{v}_p$ evaluate under every environment to a list of Booleans satisfying the parametrization predicate p:

  $$\forall \mathbf{w}\ .\ \langle p \rangle_{\mathrm{bdd}}\,(\langle \mathbf{v}_p \rangle_{\mathrm{bdd}}\,(\mathbf{w}))$$

  Therefore, a concrete input vector is only covered by a symbolic simulation of $\mathbf{v}_p$ if it satisfies p.
- Any input vector $\mathbf{u}$ that does satisfy p is covered by $\mathbf{v}_p$; that is, there is some environment under which $\mathbf{v}_p$ evaluates to $\mathbf{u}$:

  $$\langle p \rangle_{\mathrm{bdd}}\,(\mathbf{u}) \Rightarrow \exists \mathbf{u}'.\langle \mathbf{v}_p \rangle_{\mathrm{bdd}}\,(\mathbf{u}') = \mathbf{u}.$$

  Therefore, any concrete input vector satisfying p will be covered by a symbolic simulation of $\mathbf{v}_p$.

It can be nontrivial to produce "by hand" a BDD p that correctly represents a particular subset of the input space. Instead, this is handled by

the GL symbolic execution framework. The user defines an ACL2 function that takes an input vector and determines whether or not that input vector is in a particular desired subset; GL then symbolically executes this function on (unparametrized) symbolic inputs, yielding a symbolic Boolean value (represented as a BDD) that exactly represents the accepted subset of the inputs.

1.4.4 AIG-to-BDD translation

In the symbolic simulation process for the **fadd** unit, we obtain AIGs representing the outputs as a function of the primary inputs and subsequently assign parametrized input BDDs to each primary input, computing BDDs representing the function composition of the AIG with the input BDDs. A straightforward (but inefficient) method to obtain this composition is an algorithm that recursively computes the BDD corresponding to each AIG node: at a primary input, look up the assigned BDD; at an AND node, compute the BDD AND of the BDDs corresponding to the child nodes; at a NOT node, compute the BDD NOT of the BDD corresponding to the negated node. This method proves to be impractical for our purpose; we describe here the algorithm `AIG2BDD` that we use instead.

To improve the efficiency of the straightforward recursive algorithm, one necessary modification is to memoize it so as to traverse the AIG as a DAG (without examining the same node twice) rather than as a tree: due to multiple fanouts in the hardware model, most AIGs produced would take time exponential in the logic depth if traversed as a tree. The second important improvement is to attempt to avoid computing the full BDD translation of nodes that are not relevant to the primary outputs. For example, if there is a multiplexer present in the circuit and its selector is set to 1 for all settings of the inputs possible under the current parametrization, then the value of the unselected input is irrelevant unless it has another fanout that is relevant. In AIGs, such irrelevant branches appear as fanins to ANDs in which the other fanin is unconditionally false. More generally, an AND of two child AIGs a and b can be reduced to a if it can be shown that $a \Rightarrow b$ (though the most common occurrence of this is when a is unconditionally false.) The `AIG2BDD` algorithm applies in iterative stages two methods that can each detect certain of these situations without fully translating b to a BDD. In both methods, we calculate exact BDD translations for nodes, beginning at the leaves and moving towards the root, until some node's translation exceeds a BDD size limit. We replace the over-sized BDD with a new representation that loses some information but allows the computation to continue while avoiding blowup. When the primary outputs are computed, we check to see whether or not they are exact BDD translations. If so, we are done; if not, we increase the size limit and try again. During each iteration of the translation,

we check each AND node for an irrelevant branch; if a branch is irrelevant it is removed from the AIG so that it will be ignored in subsequent iterations. We use the weaker of the two methods first with small size limits, then switch to the stronger method at a larger size limit.

In the weaker method, the translated value of each AIG node is two BDDs that are upper and lower bounds for its exact BDD translation, in the sense that the lower-bound BDD implies the exact BDD and the exact BDD implies the upper-bound BDD. If the upper and lower bound BDDs for a node are equal, then they both represent the exact BDD translation for the node. When a BDD larger than the size limit is produced, it is thrown away and the constant-$true$ and constant-$false$ BDDs are instead used for its upper and lower bounds. If an AND node $a \wedge b$ is encountered for which the upper bound for a implies the lower bound for b, then we have $a \Rightarrow b$; therefore we may replace the AND node with a. Thus using the weak method we can, for example, replace an AIG representing $a \wedge (a \vee b)$ with a whenever the BDD translation of a is known exactly, without computing the exact translation for b.

In the stronger method, instead of approximating BDDs by an upper and lower bound, fresh BDD variables are introduced to replace over-sized BDDs. (We necessarily take care that these variables are not reused.) The BDD associated with a node is its exact translation if it references only the variables used in the primary input assignments. This catches certain additional pruning opportunities that the weaker method might miss, such as $b \neq (a \neq b) \rightarrow a$.

These two AIG-to-BDD translation methods, as well as the combined method `AIG2BDD` that uses both in stages, have been proven in ACL2 to be equivalent, when they produce an exact result, to the naive AIG-to-BDD translation algorithm described above.

When symbolically simulating the **fadd** unit, using input parametrization in conjunction with the `AIG2BDD` procedure works around the problem that BDD variable orderings that are efficient for one execution path are inefficient for another. Input parametrization allows cases where one execution path is selected to be analyzed separately from cases where others are used. However, a naive method of building BDDs from the hardware model might still construct the BDDs of the intermediate signals produced by multiple paths, leading to blowups. The `AIG2BDD` procedure ensures that unused paths do not cause a blowup.

1.4.5 GL Symbolic Execution Framework

The GL framework is designed to allow proof by symbolic execution within ACL2, particularly targeted at hardware verification. A typical theorem to be proven by GL consists of some hypotheses which restrict our consideration

to a finite (but often large) set of input vectors, and a conclusion which states some desired property that should hold on every input vector within this set. For example, to prove the correctness of a 16-bit adder circuit, we could hypothesize that inputs x and y are both 16-bit natural numbers, and conclude that when x and y are given as inputs to the circuit, the result produced is $x+y$. To prove this, the user specifies what shape of input objects should be used for symbolic execution (in this case, 16-bit natural numbers.) This shape specification also gives the BDD ordering for the bits of x and y. From the shape specification, GL constructs symbolic objects representing x and y. It then symbolically executes the conclusion. Ideally, the result of this symbolic execution will be a symbolic object that can syntactically be determined to always represent *true*. If not, GL will extract counterexamples from the resulting symbolic object, giving concrete values of x and y that falsify the conjecture. When the symbolic execution produces a true result, the final step in proving this theorem is to show that the symbolic objects used as inputs to the simulation cover the finite set of concrete inputs recognized by the hypothesis. In this example, 16-bit symbolic natural numbers suffice to cover the input space provided all the bits are free, independent variables; smaller symbolic naturals would not be adequate.

Symbolic objects are structures that describe functions over Booleans. Depending on the shape of such objects, they may take as their values any object in the ACL2 universe. For example, we represent symbolic integers as the pairing of a tag, which distinguishes such an object from other symbolic types such as Booleans and ordered pairs, and a list of BDDs which represent the two's-complement digits of the integer. We define an evaluator function for symbolic objects, which gives the concrete value represented by an object under an assignment of Booleans to each variable. For the integer example, the evaluator recognizes the tag and evaluates each BDD in the representation under the given assignment. Then it produces the integer whose two's-complement representation matches the resulting list of bits.

To perform a symbolic execution, we employ two methods. We may create a *symbolic counterpart* f_{sym} for a user-specified function f. f_{sym} is an executable ACL2 function that operates on symbolic objects in the same way as f operates on concrete objects. It is defined by examining the definition of f, creating symbolic counterparts recursively for all its subfunctions, and nesting them in the same manner as in the definition. Alternatively, we may symbolically interpret an ACL2 term under an assignment of symbolic objects to that term's free variables. In this case, we walk over the given term. At each function call, we either call that function's symbolic counterpart if it exists, or else look up the function's definition and recursively symbolically interpret it under an assignment that pairs its formals with the corresponding symbolic values produced by the given actual parameters.

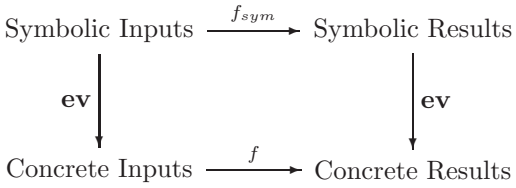
In both methods of symbolic execution, it is necessary for any ACL2 primitives to have predefined symbolic counterparts, since they do not have definitions. We have defined many of these functions manually and proven the

correctness of their symbolic counterparts. For example, the symbolic counterpart of + is defined such that on symbolic integers, it performs a BDD-level ripple-carry algorithm, producing a new symbolic integer that provably always evaluates to the sum of the evaluations of the inputs. We have also manually defined symbolic counterparts for certain functions for which symbolic interpretation of the ACL2 definitions would be inefficient. For example, the bit-wise negation function *lognot* is defined as $\mathit{lognot}(x) = (-x) - 1$, but for symbolic execution it is more efficient to perform the bit-wise negation directly by negating the BDDs in the symbolic integer representation; in fact, we define the negation operator in terms of *lognot*, rather than the reverse.

The correctness condition for a symbolic counterpart f_{sym} states a correspondence between the operation of f_{sym} on symbolic objects and the operation of f on concrete objects. Namely, the evaluation of the (symbolic) result of f_{sym} on some symbolic inputs is the same as the (concrete) result of running f on the evaluations of those inputs:

$$\mathbf{ev}\left(f_{sym}\left(s\right), a\right) = f\left(\mathbf{ev}\left(s, a\right)\right)$$

The following diagram illustrates the correspondence:



Each primitive symbolic counterpart we have defined is proven (using standard ACL2 proof methodology) to provide this correctness condition. The correctness of symbolic counterparts of functions defined in terms of these primitives follows from this; the correctness proofs are automated in the routine that creates symbolic counterparts. The symbolic interpreter is also itself verified; its correctness condition is similar. Suppose we symbolically interpret a term x with a binding of its variables v_i to symbolic objects s_i, yielding a symbolic result. We have proven that the evaluation of this result under assignment a equals the result of running the term x with its variables v_i each bound to $\mathbf{ev}(s_i, a)$.

These correctness conditions allow theorems to be proven using symbolic execution. Consider our previous example of the 16-bit adder. Suppose we symbolically execute the conclusion of our theorem on symbolic inputs s_x, s_y and the result is an object that evaluates to *true* under every variable assignment:

$$\forall a \;.\; \mathbf{ev}\left(\mathit{conc}_{sym}\left(s_x, s_y\right), a\right).$$

By the symbolic execution correctness condition, this commutes to:

$$\forall a \;.\; \mathit{conc}\left(\mathbf{ev}\left(s_x, a\right), \mathbf{ev}\left(s_y, a\right)\right).$$

That is, the conclusion holds of any pair of values x and y such that s_x and s_y evaluate to that pair under some assignment. The coverage side condition then requires us to show that s_x and s_y are general enough to cover any pair that satisfies the theorem's hypothesis. Once this is proven, the proof of the theorem (hypotheses imply conclusion) is complete.

1.5 Verification Results and Observations

We have used our ACL2-based verification methodology to prove the correctness of several instructions in Centaur's execution cluster, including packed floating-point addition/subtractions and comparisons, format conversions, logical operations, shuffles, and integer and packed-integer multiplication. We have also verified the one-cycle invariant for the divider and proven the correctness of several microcode routines.

Our floating-point addition verification was performed using a machine with four Intel Xeon X7350 processors running at 2.93 GHz with 128 GB of system memory. However, each of our verification runs is a single-threaded procedure, and we limit our memory usage to 35GB for each process so that we can run single, double, and extended-precision verifications concurrently on this machine without swapping. The symbolic simulations run in 8 minutes 40 seconds for single precision, 36 minutes for double precision, and 48 minutes for extended precision. Proof scripts required to complete the three theorems take an additional 10 minutes of real time when using multiple processors, totalling 25 minutes of CPU time. The process of reading the Verilog design into ACL2, which is done as part of a process that additionally reads in a number of other units, takes about 17 minutes. In total it takes about 75 minutes of real time (125 minutes of CPU time) to reread the design from Verilog sources and complete verifications of all three instructions.

We found two bugs with our verification process, which began after the floating-point addition instructions had been thoroughly checked using a testing-based methodology. The first bug was a timing anomaly affecting SSE addition instructions, which we found during our initial investigation of the media unit. Later, a bug in the extended precision instruction was detected by symbolic simulation. This bug affected a total of four pairs of input operands out of the 2^{160} possible, producing a denormal result of twice the correct magnitude. Because of the small number of inputs affected, it is unlikely that random testing would have encountered the bug; directed testing had also not detected it. Both bugs have been fixed in the current design.

Working in an industrial environment forced us to be able to respond to design changes quickly. Every night, we run our Verilog translator on the entire 570,000 lines of Verilog that comprise the Centaur CN and produce output for all of the Verilog that we can translate. We build a new copy of ACL2 with our `EMOD` representation of the design already included so when

we sit down in the morning, we are ready to work with the current version of the design. Also, each night, we re-run many of the verifications that have been done previously to make sure that recent changes are safe. Each week, we attempt to re-run our entire regression suite of previously proven results.

Our major challenges involved getting our toolsuite to be sufficiently robust, getting the specification correct, dealing with the complicated clocking and power-saving schemes employed, and creating a suitable circuit input environment. It is difficult for us to provide a meaningful labor estimate for this verification because we were developing the translator, flow, our tools, our understanding of floating-point arithmetic, and our specification style simultaneously. Now, we could likely check another IEEE-compatible floating-point design in the time it would take us to understand the clocking and input requirements. Centaur will certainly be using this methodology in the future; it is much faster, cheaper, and more thorough than non-exhaustive simulation. Although our verification approach is currently dependent on BDDs, we have considered what would be required to have an AIG-and-SAT flow.

The improvements in ACL2 that permitted this verification will be included in future ACL2 releases. The specifics of Centaur's two-cycle, floating-point design are considered proprietary. We plan to publish our ACL2-checked proof that our integer-level specification is equal to our IEEE floating-point specification; this level of proof is similar to work by Harrison [9].

1.6 Related Work

Several groups have completed floating-point addition and other related verifications. Notably, Intel has largely supplanted testing-based validation of the execution unit, instead using full formal verification based on symbolic trajectory evaluation in the Forte/reFLect system [17]. Also, IBM has integrated formal verification based on model-checking and equivalence checking into their mainstream verification flow [3]. Our formal verification has only covered a small fraction of the instructions that run on Centaur's execution unit; we hope to expand this coverage in the future. However, we differ from previous verifications in that we obtained our result using verified automated methods within a general-purpose theorem prover, and in that we base our verification on a formally defined HDL operating on a data representation mechanically translated from the RTL design.

An AMD floating-point addition design was verified using ACL2. It was proven to comply with the primary requirement of the IEEE 754 floating-point addition specification, namely that the result of the addition operation must equal the result obtained by performing the addition at infinite precision and subsequently rounding to the required precision [19]. The design was represented in ACL2 by mechanically translating the RTL design into ACL2 functions. A top-level function representing the full addition unit was proven

to always compute a result satisfying the specification. This theorem was proved in ACL2 by the usual method of mechanical theorem proving, wherein numerous human-crafted lemmas are proven until they suffice to prove the final theorem. A drawback to this method is that even small changes to the RTL design may require the proof script to be updated. We avoid this pitfall by using a symbolic simulation-based methodology. Our method also differs in that we use a deep embedding scheme, translating the RTL design to be verified into a data object in an HDL rather than a set of special-purpose functions.

We described our floating-point addition verification previously [12]. Among bit-level symbolic simulation-based floating-point addition verifications, many have used a similar case-splitting and BDD parametrization scheme as ours [2,15,20,21]. The symbolic simulation frameworks used in all of these verifications, including the symbolic trajectory evaluation implementation in Intel's Forte prover, are themselves unverified programs. Similarly, the floating-point verification described in [8] uses the SMV model checker and a separate argument that its case-split provides full coverage. To obtain more confidence in our results, we construct our symbolic simulation mechanisms within the theorem prover and prove that they yield sound results. Combining tool verifications with the results of our symbolic simulations yields a theorem showing that the instruction implementation equals its specification.

1.7 Conclusion

In the verification methodology used at Centaur, we use a combination of symbolic simulation and conventional theorem proving to verify equivalences between hardware models and specifications written as ACL2 functions. Because our toolflow consists, to a large extent, of programs that have been verified by the ACL2 theorem prover, we are able to obtain ACL2 theorems reflecting our verification results even though the proofs are done in large part through symbolic simulation.

We model the design using a deep embedding in the `EMOD` formal HDL, which we obtain by automatic translation of the Verilog RTL design. We run a new translation nightly so as to keep current on the design effort, because we primarily use "black box" verification methods on the design. We rarely need to update proof scripts in response to design changes.

Our verification efforts have yielded correctness proofs for several instructions including floating-point addition, subtraction, and integer multiplication, conversions between integer and float formats, and comparisons. These proof efforts resulted in the discovery of two flaws in floating-point addition instructions that had escaped extensive simulation; these flaws have been corrected in Centaur's current design.

1.8 Acknowledgements

We would like to acknowledge the support of Centaur Technology, Inc., and ForrestHunt, Inc. We would also like to thank Bob Boyer for development of much of the technology behind `EMOD` and the ACL2 BDD package, Terry Parks for developing a very detailed floating-point addition specification, and Robert Krug for his proof that our integer-level, floating-point addition specification performs the rational arithmetic and rounding specified by the IEEE floating-point standard.

References

1. *IEEE Standard (1364-2005) for Verilog Hardware Description Language*. IEEE, 2005.
2. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of boolean constraints. In *Proceedings of the 36th Design Automation Conference*, pages 402–407, 1999.
3. Jason Baumgartner. Integrating FV into main-stream verification: The IBM experience. Tutorial given at FMCAD, 2006. Available at `http://domino.research.ibm.com/comm/research_projects.nsf/pages/sixthsense.presentations.html`.
4. Robert S. Boyer and Warren A. Hunt, Jr. Function memoization and unique object representation for ACL2 functions. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 81–89, New York, NY, USA, 2006. ACM.
5. Robert S. Boyer and Warren A. Hunt, Jr. Symbolic simulation in ACL2. *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, 2009.
6. Bishop Brock and Warren A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. *NASA Contractor Report*, 187540, 1991.
7. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
8. Yirng-An Chen and Randal E. Bryant. Verification of floating-point adders. *Computer Aided Verification*, 1427, 1998.
9. John Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*, volume 3965 of *Lecture Notes in Computer Science*, pages 211–242, Bertinoro, Italy, 2006. Springer-Verlag.
10. Warren A. Hunt, Jr. *FM8501: a verified microprocessor*. Springer-Verlag, London, UK, 1994.
11. Warren A. Hunt, Jr. and Erik Reeber. Formalization of the DE2 Language. *Proceedings of the 13th Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, pages 20–34, 2005.
12. Warren A. Hunt, Jr. and Sol Swords. Centaur technology media unit verification: Case study: Floating-point addition. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 353–367. Springer, 2009.
13. Warren A. Hunt, Jr. and Sol Swords. Use of the E language. In *Hardware design and Functional Languages*, 2009.
14. IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*, IEEE std 754™-2008 edition.

15. Christian Jacobi, Kai Weber, Viresh Paruthi, and Jason Baumgartner. Automatic formal verification of fused-multiply-add FPUs. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1298–1303 Vol. 2, 2005.
16. Robert B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.
17. Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel®Core™i7 processor execution engine validation. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 414–429. Springer, 2009.
18. Dick Price. Pentium FDIV flaw – lessons learned. *IEEE Micro*, 15(2):88–87, 1995.
19. David Russinoff. A case study in formal verification of Register-Transfer logic with ACL2: the floating point adder of the AMD Athlon (TM) processor. In *Formal Methods in Computer-Aided Design*, pages 22–55, 2000.
20. Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381, 2005.
21. Anna Slobodová. Challenges for formal verification in industrial setting. In *Formal Methods: Applications and Technology*, pages 1–22, 2006.
22. University of California at Berkeley, Department of Electrical Engineering and Computer Science, Industrial Liaison Program. *A compact test suite for P754 arithmetic – version 2.0*.
23. Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40:831–873, July 2005.