Centaur Technology Media Unit Verification
Case Study: Floating-Point Addition

Warren A. Hunt Jr. and Sol Swords

Centaur Technology
7600C North Capitol of Texas Hwy
Austin, TX 78731
{hunt,sswords}@centtech.com

Abstract. We have verified floating-point addition/subtraction instructions for the media unit from Centaur's 64-bit, X86-compatible microprocessor. This unit implements over one hundred instructions, with the most complex being floating-point addition/subtraction. This media unit can add/subtract four pairs of floating-point numbers every clock cycle with an industry-leading two-cycle latency.

Using the ACL2 theorem proving system, we model the media unit by translating its Verilog design into an HDL that we have deeply embedded in the ACL2 logic. We specify the addition/subtraction instructions as integer-based ACL2 functions. Using a combination of AIG- and BDD-based symbolic simulation, case splitting, and theorem proving, we produce a mechanically checked theorem in ACL2 for each instruction examined stating that the hardware model yields the same result as the instruction specification.

In pursuit of these verifications, we implemented a formal HDL and symbolic simulation framework, including formally verified BDD and AIG operators, within the ACL2 theorem proving system. The HDL includes an extensible interpreter capable of performing concrete and symbolic simulations as well as non-functional analyses. We know of no other symbolic simulation-based floating-point verification that is performed within a single formal system and produces a theorem in that system without relying on unverified external tools.

1 Introduction

Media units in contemporary microprocessors contain sophisticated implementations to provide low-latency arithmetic operations. We have used the ACL2 theorem-proving system [1] to mechanically verify a number of media instructions for Centaur's X86-compatible, 64-bit CN microprocessor; this unit implements over 100 of the SSE media instructions and X87 addition and subtraction. The most complex instructions we verified are the packed addition/subtraction instructions; it is these instructions that are the focus of this paper.

Floating-point addition/subtraction implementations must precisely contend with a variety of numeric input conditions: normal, denormal, zero, infinity, quiet not-a-number (QNaN), and signaling not-a-number (SNaN). If the inputs can be added, they first are aligned, then the addition or subtraction and rounding

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 353–367, 2009.

occurs, and finally the result flags are calculated and set.The result produced itself may be any of the six types just mentioned except for an SNaN. When performing packed (with multiple operand pairs) addition/subtraction, the computation of the result flags involves merging the flag results for all input pairs. Note that there is essentially no difference between addition and subtraction: an addition operation may effectively be a subtraction because one of the inputs represents a negative number. For the remainder of this paper we will only write addition, but the reader should always keep in mind that a subtraction may occur.

Our tool flow involves translating the Verilog representation of the entire media unit to our formally defined HDL [2]. This process captures Centaur's design as an ACL2 constant that we interpret with our ACL2-based HDL simulator. Our HDL simulator can be used to symbolically simulate the formal circuit description; we use a combination of theorem proving and equivalence checking to prove that the circuit's outputs are equivalent to an integer-level specification of floating-point operations. Separately, we have verified that our integer-level specification actually implements IEEE floating-point addition [3]; this proof is not discussed here.

Our formally defined HDL is a deep-embedding of the `EMOD` HDL [4] inside of the ACL2 logic. We have defined a function that determines whether a module has a well-formed syntax, and we have also defined an interpreter that gives meaning to such modules. Our `EMOD` interpreter provides multiple interpretations, thus a module can be evaluated with either a two-valued (Boolean) logic or a four-valued logic. In addition, the `EMOD` interpreter can compute dependencies and estimate circuit delays; the `EMOD` interpreter has an input parameter that allows a user to specify which evaluation mechanism to use. We are not aware of any other verification of a commercial floating-point design using such a deep-embedding approach. This approach reduces the risk of translation errors, as it is possible to 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.

We begin by describing Centaur's media unit and the process by which we capture this unit's design in our HDL. Once we have a formal representation of the media unit, we perform symbolic simulation to produce a set of equations that represent the operation of the media unit. We then refine these equations given that we want to verify three sizes of addition operations, and then use parametrization and equivalence checking to prove that the media unit can add four 32-bit pairs, two 64-bit pairs, or one 80-bit pair of numbers. This yields a theorem for each instruction stating the equivalence of its hardware implementation (as interpreted in our HDL) with its specification function. We close by relating our efforts to previous work and describing how our effort is unique.

2 Centaur CN's Media Unit

Our effort is focused on Centaur's 64-bit CN processor shown in Figs. 1 and 2. The CN family of processors is compliant with AMD and Intel's 64-bit

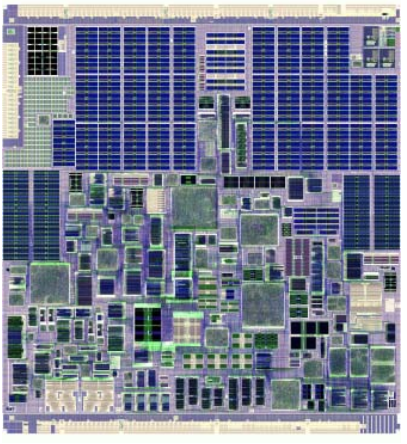

Fig. 1. Die Plot of Centaur CN

Fig. 2. Floor Plan of Centaur CN

processors, and its implementation supports almost 40 operating systems and four hypervisors. The Centaur CN media unit handles X87 and SSE instructions. The **fadd** unit, a major component of the media unit, itself implements single, double, and extended floating-point addition and subtraction, both scalar and packed; floating-point to integer conversions and integer to floating-point conversions; SSE logical and integer operations; and other operations. Our investigation was focused on the two-cycle latency floating-point addition instructions of the **fadd** unit; we believe that this design is the lowest clock-cycle latency floating-point implementation presently available.

The Centaur media unit is part of the execution cluster; operands and instructions are provided to this unit by the instruction scheduler. Logical operation results are available in one clock cycle while floating-point addition, subtraction, and conversions are available in two cycles. The lower-left-hand corner of the die contains the media unit; the relevant parts can be seen in Fig. 3. Multiple 128-bit inputs and outputs are provided and this unit can forward results internally so operations may be chained.

Fig. 3. Floorplan of CN Media Unit

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 and 1074 inputs including 26 clock inputs. Multiple inputs representing the same clock are used in order to manage power usage.

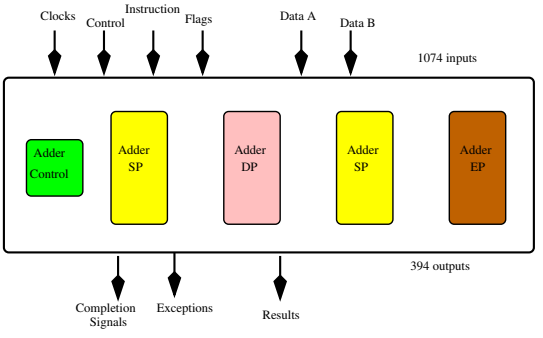

Fig. 4. Adder Units Inside of **fadd**

The **fadd** unit is composed of four adders: two 32-bit units, one 64-bit unit, and one 80-bit unit (see Fig. 4). 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.

The critical speed path through **fadd** is the floating-point addition logic. 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.

Floating-point numbers are composed of three parts: a sign, a mantissa, and an exponent. Floating-point addition operands are identified as numbers, either normal, denormal or infinity, or as non-numbers (NaNs), either quiet (QNaN) or signaling (SNaN). Any NaN operand produces a QNaN result but may also cause an exception.

In addition to the floating-point operands, the setting of the floating-point control flags must be considered when adding. For instance, the flush-to-zero (FTZ) flag indicates that when a denormal result is internally produced, zero should be stored. Producing result flags is complicated by flag priority and the interaction of flag settings with input mask settings when performing packed operations. For example, when one pair of operands would produce an invalid exception and another pair would produce a denormal exception, the denormal flag is only set when the invalid exception is masked.

Floating-point addition proceeds in steps. First, several special cases are detected. If a NaN operand is identified, then a NaN result is produced. Infinite operands also require special consideration, as does the case where both input

operands are either zero or denormal with the denormals-are-zero control flag set. If none of these cases hold, the mantissas of the two operands are shifted relative to one another as determined by the difference in their exponents, and the shifted mantissas are added or subtracted as appropriate and rounded according to the rounding mode specified by the control flags. Finally, the correct exponent for the rounded result is calculated and is examined to determine whether an overflow or underflow occurred.

3 Verification Method

As shown in Fig. 5, our verification methodology compares the result of symbolic simulations of an instruction specification and a translation of the **fadd** unit's Verilog design into our `EMOD` hardware description language. By running an AIG-based symbolic simulation of the **fadd** model using the `EMOD` symbolic simulator, we represent the outputs of the **fadd** unit as fully general functions of the inputs using AIGs as the Boolean function representation. We then specialize these functions by setting input control bits to values appropriate for the desired instruction. In order to compare these functions with those produced by the specification, we then convert these AIGs into BDDs.

In order to make this feasible, we use case splitting via BDD parametrization [5,6] 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 blowup due to adding mantissas composed of poorly aligned BDD variables. 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. Checking that corresponding BDDs from these results are equal shows that the **fadd** unit operates identically to the specification function on the subset

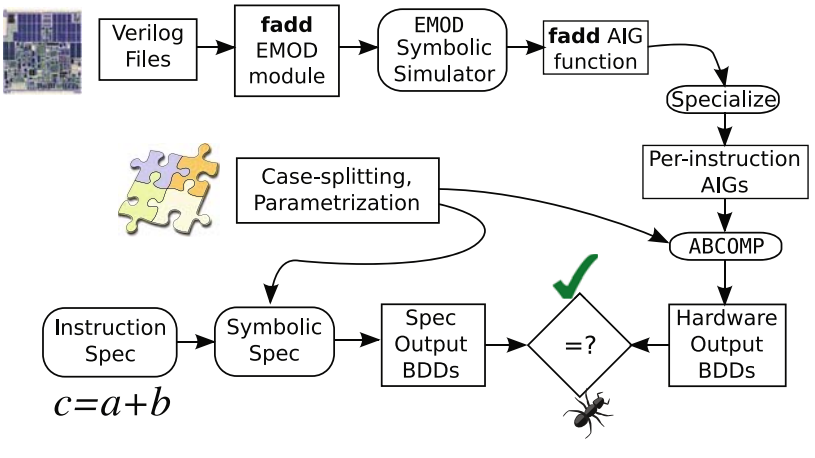

Fig. 5. Verification Method

of the input space covered case split; otherwise, we can generate counterexamples by analyzing the differences in the outputs.

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.

3.1 Case-Splitting and Parametrization

A major obstacle to symbolic simulation of floating-point adders is a 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 [5, 7, 8, 9], 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 exponent 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. 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.

3.2 Symbolic Simulation of the Hardware Model

The object we actually consider for verification is a representation of Centaur's Verilog description of the **fadd** unit. This object is created by translating Verilog to the `EMOD` language by way of a parser and translator written in ACL2 [2]. The Verilog to `EMOD` translator parses the Verilog design and performs a synthesis step resulting in a gate-level model that is then directly translated into the `EMOD` language.

Our translator targets a limited subset of Verilog, mainly related to register-transfer-level constructs like gate and submodule instances and arrays, continuous assignments, and very limited "always" blocks for modeling latches and flops. This allows translation of most of the 550,000 lines of Verilog used to describe the full Centaur design. Transistor-level constructs are not supported. Certain other Verilog features, such as strings, real variables, hierarchical identifiers, and multi-dimensional arrays are not supported, but in some of these cases we may be able to extend the translator.

To obtain formulas representing the outputs of the **fadd** unit in terms of its primary inputs, we use the `EMOD` simulator to perform an AIG-based symbolic simulation of the **fadd** model. We use a four-valued logic in this simulation, 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. 6.

In this simulation, all bits of the initial state are set to unknown (X) values and the onsets and offsets of all non-clock inputs are assigned to unique Boolean variables at each cycle, so that every input signal but the clocks can take any of the four values. 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. 6.

Many of the primary inputs of the **fadd** unit are either irrelevant to the operation of the addition instructions or must be set to exact values in order for the addition instruction to occur. We therefore restrict the fully-general output formulas by setting control signals to the values required for an instruction and setting irrelevant signals 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 operand and status register inputs, which are the same as the inputs to the specification function. In fact, our verification results in a proof that the Boolean functions represented by these specialized AIGs are equivalent to the output bits of the instruction specification.

As the final step in symbolically simulating the **fadd** unit, for each case split, we derive from the specialized AIGs a set of BDDs that express the **fadd** unit results for input vectors covered by the case. To do this, we assign to each bit of the operands and status register the corresponding BDD constructed by the parametrization for the case split. We then use the AIG/BDD composition procedure `AIG2BDD` described in Sec. 4.4 to produce BDDs representing the primary outputs. `AIG2BDD` avoids computing certain intermediate-value BDDs that are irrelevant to the final output, which helps to solve some BDD size explosions. This AIG-to-BDD conversion concludes the symbolic simulation flow on the hardware side; the resulting BDDs may then be compared to the results from the symbolic simulation of the instruction specification.

3.3 Symbolic Simulation of Specification

The specification for each instruction is an ACL2 function that takes integers representing the operands and the control register and produces integers representing the result and the flag register. It is defined in terms of word-level primitives such as shift, bitwise AND, plus and minus. 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 [10]. 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 [11] as well as random tests.

We use the GL symbolic execution framework [12] for ACL2 in order to perform the symbolic simulation of the specification. The GL framework contains a routine that apples a code transform to the definition of an ACL2 user function to produce a BDD-based symbolic simulator function and proves that the symbolic simulator results reflect the behavior of the original function. The symbolic simulator function operates on data structures that use BDDs to symbolically represent Booleans and lists of BDDs to symbolically represent two's-complement integers.

To symbolically simulate the specification function, we use the GL code transform to produce such a symbolic simulator function. For each case split, we use BDDs resulting from parametrization to construct symbolic integer data structures representing the operands and the status register. We run the symbolic simulator function on these inputs, obtaining symbolic integer data structures representing the result and flag register. BDDs extracted from these data structures (representing individual bits of the result and flag register) can be compared with the BDDs obtained by symbolically simulating the hardware model on the same case split.

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. For each case split, we produce 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.

4 Mechanisms Used to Achieve the Verification

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

```
(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. 7. EMOD examples

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 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. 7. 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 RTL design of the **fadd** 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, Forte reads Intel design descriptions and builds a FSM in the Forte tool 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.

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. The BDD and AIG evaluators accept different formats for the variable assignment argument: the BDD evaluator expects a list of Booleans that correspond positionally to the variables at the levels of the BDDs, whereas the AIG evaluator expects a list of key/value pairs associating the primary inputs of the AIG with their Boolean values. 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:

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

4.3 Parametrization

BDD parametrization is also implemented in ACL2. The parametrization algorithm is described in [5]; 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, $\boldsymbol{v} = [v_0, \ldots, v_{n-1}]$. This provides complete coverage because $\langle \boldsymbol{v} \rangle_{\mathrm{bdd}}(env)$ may equal any list of n Booleans. (In fact, if env has length n, then $\langle \boldsymbol{v} \rangle_{\mathrm{bdd}}(env) = env$.) However, to avoid BDD blowups, we wish to 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 $\boldsymbol{w}$ is in the coverage set if and only if $\langle p \rangle_{\mathrm{bdd}}(\boldsymbol{w})$. To do this, we parametrize $\boldsymbol{v}$ by predicate p and use the resulting BDDs $\boldsymbol{v}_p$ as the symbolic inputs. The following theorems hold of the parametrization transform and have been proved in ACL2:

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

and, assuming $\boldsymbol{u}$ is a list of n Booleans,

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

The first property shows that the parametrized BDDs always evaluate to a list of Booleans that satisfies p; therefore, a concrete input vector is only covered by a symbolic simulation of $\boldsymbol{v}_p$ if it satisfies p. The second property shows that any input vector that does satisfy p will be covered by such a symbolic simulation.

It can be nontrivial to produce "by hand" a BDD p that correctly represents a particular subset of the input space. Instead, we define an ACL2 function that determines whether or not a pair of input operands is in a particular desired

subset. We then run the GL code transform to produce a symbolic analogue for this function. Running this symbolic simulator on (unparametrized) symbolic inputs yields a symbolic Boolean value (represented as a BDD) that exactly represents the accepted subset of the inputs.

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 multiplexor present in the circuit and, for the current parametrized subset of the inputs, the selector is always set to 1, 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 until some node's translation exceeds a BDD size limit. We replace the oversized 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 irrelevnant 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 each is 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 as long as 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 oversized 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 straightforward AIG-to-BDD translation algorithm described above.

When symbolically simulating the **fadd** unit, using input parameterization 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.

5 Results of Verification

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 design flaws in the media unit during 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.

6 Related Work

Several groups have completed floating-point addition and other related verifications. 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 [13]. 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.

Among bit-level symbolic simulation-based floating-point addition verifications, many have used a similar case-splitting and BDD parametrization scheme as ours [5, 9, 7, 14]. 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. In order 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.

7 Observations

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 550,000 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 every verification that has been done to date to make sure that recent changes are safe.

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 for the 1074 inputs. 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 specification style simultaneously. Now, we could 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.

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 [15].

8 Conclusion

The verification of Centaur CN floating-point addition instructions was performed through a combination of theorem proving, symbolic simulation, and equivalence checking. Centaur's CN media unit has an industry-leading, two-cycle latency, which complicates the implementation. Our efforts resulted in the discovery of two flaws that had escaped extensive simulation; these flaws have been corrected in Centaur's current design. In contrast to other floating-point adder verifications, our hardware model is represented by deep embedding in an extensible formal HDL with an interpreter written in ACL2. Furthermore, we programmed and verified our symbolic simulation procedures inside the ACL2 theorem prover, allowing us to obtain an ACL2 theorem stating the equivalence between each instruction's execution on the hardware model and its specification.

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 BDD package, Terry Parks for developing a very detailed floating-point addition specification, and Jared Davis for authoring our Verilog-to-E translator.

References

1. Kaufmann, M., Moore, J.S., Boyer, R.S.: ACL2 version 3.4 (2009), `http://www.cs.utexas.edu/~moore/acl2/`
2. Davis, J.: VL Verilog translator (unpublished)
3. Krug, R.: Correctness proof of ACL2 floating-point addition specification (unpublished)
4. Hunt Jr., W.A., Swords, S.: Use of the E language. In: Hardware design and Functional Languages (2009)
5. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Formal verification using parametric representations of boolean constraints. In: Proceedings of the 36th Design Automation Conference, pp. 402–407 (1999)
6. Jones, R.B.: Symbolic Simulation Methods for Industrial Formal Verification. Kluwer Academic Publishers, Dordrecht (2002)
7. Jacobi, C., Weber, K., Paruthi, V., Baumgartner, J.: Automatic formal verification of fused-multiply-add FPUs. In: Proceedings of Design, Automation and Test in Europe, vol. 2, pp. 1298–1303 (2005)
8. Chen, Y., Bryant, R.E.: Verification of floating-point adders. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427. Springer, Heidelberg (1998)
9. Seger, C.J.H., Jones, R.B., O'Leary, J.W., Melham, T., Aagaard, M.D., Barrett, C., Syme, D.: An industrially effective environment for formal hardware verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24(9), 1381 (2005)
10. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE std 754^{TM}-2008 edn. (2008)
11. 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
12. Boyer, R.S., Warren, A., Hunt, J.: Symbolic simulation in ACL2. In: Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications (2009)
13. Russinoff, D.: A case study in formal verification of Register-Transfer logic with ACL2: the floating point adder of the AMD Athlon (TM) processor. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 3–36. Springer, Heidelberg (2000)
14. Slobodová, A.: Challenges for formal verification in industrial setting. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 1–22. Springer, Heidelberg (2007)
15. Harrison, J.: Floating-point verification using theorem proving. In: Bernardo, M., Cimatti, A. (eds.) SFM 2006. LNCS, vol. 3965, pp. 211–242. Springer, Heidelberg (2006)