Verification of All Circuits in a Floating-Point Unit Using Word-Level Model Checking

Yirng-An Chen*, Edmund Clarke*, Pei-Hsin Ho**, Yatin Hoskote**, Timothy Kam**, Manpreet Khaira**, John O'Leary**, Xudong Zhao**

Abstract This paper presents the formal verification of all sub-circuits in a floating-point arithmetic unit (FPU) from an Intel microprocessor using a word-level model checker. This work repre- sents the first large-scale application of word-level model checking techniques. The FPU can perform addition, subtraction, multiplication, square root, division, remainder, and rounding operations; verifying such a broad range of functionality required coupling the model checker with a number of other techniques, such as property decomposition, property specific model abstraction, and latch removal. We will illustrate the verification techniques used by us with respect to the design of the Weitek WTL3170/3171 Sparc floating point coprocessor. The principal contribution of this paper is a practical verification methodology explaining what techniques to apply (and where to apply them) when verifying floating-point arithmetic circuits. We have applied our methods to the floating-point unit of a state-of-the- art Intel microprocessor, which is capable of extended precision (64-bit mantissa) computa- tion. The success of this effort demonstrates that word-level model checking, with the help of other verification techniques, can verify arithmetic circuits of the size and complexity found in industry.

1 Introduction

The floating-point division flaw [SB94, Coe95] in Intel Corp.'s Pentium underscores how hard the task of verifying a floating-point arithmetic unit is, and how high the cost of a floating-point arithmetic bug can be. About one trillion test vectors were used and none uncovered the bug. The recall and replacement of the chips in the field cost Intel $470 million. Since the Pentium processor flaw came to light, there has naturally been new interest in improved methods for functional verification of arithmetic hardware - especially in formal methods, which provide exhaustive coverage of the implementation's behavior. This paper describes the formal verification of a complete floating-point unit, described at the structural level in a hardware description language, using word-level model checking techniques and gives verification results for a recent Intel microprocessor. We have verified the correct implementation of the addition, subtraction, multiplication, square root, division, remainder, and rounding operations. This work is the first large-scale application of word-level model checking techniques.

The principal contribution of this paper is to demonstrate how word-level model checking can be applied to practically and efficiently verify arithmetic circuits in state-of-the-art microprocessors. We will focus on the techniques we found useful for various classes of circuits, rather than the details of the model checking algorithm itself which are

* Intel Development Labs, 5200 NE Elam Young Parkway, M/S JFT-102, Hillsboro, OR 97124-6497 USA

** School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA

covered in [CKZ96]. We illustrate our techniques with respect to the design of the Weitek WTL3170/3171 floating-point coprocessors. We chose the Weitek part because substantial detail has been published about its architecture and algorithms[BSC+90], though we think it is simpler than the Intel design we actually verified. However, we emphasize that the verification methodology we propose is very general and our techniques are applicable to other floating-point and integer arithmetic circuits as well. We will report the results of verifying the FPU from an Intel microprocessor using these techniques in a separate section.

Previous work in formally verifying arithmetic hardware has either used BDD-based algorithms [Bry91, Bry95], or theorem proving techniques [VCM94], or a combination of both [KL93]. The first approach has the disadvantage that it requires extremely detailed, bit-level specifications that are difficult to formulate correctly. Moreover, the bit-level specifications of operations like multiplication can be exponentially complex. The latter two approaches are relatively laborious and require users with substantial special training to guide the proof. However, they allow specifications to be written cleanly, in terms of the usual arithmetic operations upon arbitrary-precision integers. Our work demonstrates that word-level model checking algorithms combine the best of both worlds, admitting a high degree of automation while allowing very abstract specifications.

The remainder of this paper is organized as follows. Section 2 contains a brief introduction to the word-level model checking techniques we used in verifying the FPU. Section 3 describes the functional units in the Weitek FPU and the techniques that can be used to verify them. In particular, we will discuss property decomposition, property-specific model extraction, latch removal, and verification by invariants. Section 4 presents results obtained in applying these techniques to a floating-point unit from an Intel microprocessor.

2 Word-Level Model Checking

Symbolic model checking [McM93] is a very efficient technique for verifying the correctness of sequential circuits. It is based on binary decision diagrams (BDDs) and has been very successful in verifying the control logic of industrial circuits. However, BDDs are sometimes unable to represent the data path of circuits efficiently (e.g. multipliers and shifters), preventing their wide-spread use in the verification of arithmetic circuits. Recently, data structures that allow such an efficient representation have been derived from BDDs, such as binary moment diagrams (BMD) [BC95] and multi-terminal BDDs (MTBDD) [CMZ+93]. These representations have further been combined to form hybrid decision diagrams (HDDs) [CFZ95].

2.1 Hybrid Decision Diagrams

A BDD is a directed acyclic graph with a total order on the occurrence of variables from root to leaf. Multi-terminal BDDs have a similar structure. However, BDDs have Boolean leaves, while MTBDDs have integer leaves and therefore represent functions from Booleans to integers. Functions from integers to integers can also be represented when the input is encoded in binary form. Efficient algorithms exist that compute common arithmetic operations when operands are given in this form. A BMD is another representation for functions that map Boolean vectors to integers. This representation is more compact for some useful arithmetic functions which have exponential size if represented using MTBDDs.

Both BMDs and MTBDDs have been integrated into the verification system at Intel through the use of hybrid decision diagrams (HDDs). In particular, for state variables in the circuit corresponding to data bits, this hybrid representation behaves like a BMD; while for the state variables corresponding to control signals, it behaves like a MTBDD. By using HDDs in this manner, this system is able to handle complex circuits containing both complex control logic and wide data paths. The reader is referred to [BC95, CMZ$^+$93, CFZ95] for detailed descriptions of BDDs, BMDs and HDDs.

2.2 Specifying Word-Level Properties

The HDD-based verification system allows the expression of properties involving relationships among data variables. Unlike a BDD-based system where properties can only reason about state variables, the HDD-based system allows properties involving relations between the values of data variables called words. A word is an array or bit vector of state variables. The value of the word is the value of the unsigned integer represented by that bit vector. An arithmetic expression can be constructed from words in the circuit, constants and arithmetic operations on words. In our word-level extended CTL, any Boolean combination of strict or nonstrict inequalities of integer expressions with arithmetic operations such as addition, subtraction, multiplication can be specified. For example, the property

$$\mathbf{AG}((p < 0) \rightarrow \mathbf{AX}((p = (-2 \cdot b + 3 \cdot r)) \wedge p \geq 0))$$

specifies that it is always the case that if p is negative, then in the next clock phase the value of p is equal to and p becomes non-negative.

Extended CTL allows a wide range of abstract specifications on data variables, which are not expressible in a system using a Boolean representation.

2.3 Model Checking with Word-Level Properties

Model checking is a technique that, given a state-transition graph and a temporal logic formula, determines which states satisfy the formula. In symbolic model checking systems [McM93], BDDs are used to represent the transition relations and sets of states. The model checking process is performed iteratively on these BDDs. Symbolic model checking has dramatically increased the size of circuits that can be formally verified. However, model checking algorithms cannot be used directly for verifying arithmetic circuits. Expressions that involve variables with integer values cannot be handled in a clean and efficient manner. Word-level model checking overcomes this problem by extending the original algorithms to evaluate arithmetic expressions using hybrid decision diagrams [CKZ96].

In word-level model checking, the transition relation and formulas not involving words are implemented using BDDs as in the original algorithm. HDDs are used only to compute word-level expressions. The BDD representing the set of variable assignments that make an algebraic relation true is built using the HDD representations of the expressions within the algebraic relation. After the BDDs for atomic formulas have been computed, the BDDs for temporal formulas are computed in the same way as in standard model checking. The iterative computations are exactly the same in both cases. The power of this extended system will be evident from the verification results presented in this paper.

3 Verifying a Floating-Point Arithmetic Unit

3.1 Weitek Floating-Point Coprocessor

In this section we present a typical FPU, the Weitek WTL3170/3171 Sparc floating-point coprocessor [BSC+90], and the techniques we have found useful to verify each type of circuit found in the coprocessor. The verification methodology we use for floating-point arithmetic circuits is very general and our techniques are applicable to other arithmetic circuits as well. In section 4, we shall report verification results specific to an Intel FPU design. We will focus on the mantissa computations, as they pose the most interesting verification challenges. The exponent unit., which is shared among several of the FPU's functions, is relatively simple to verify.

[hbt]

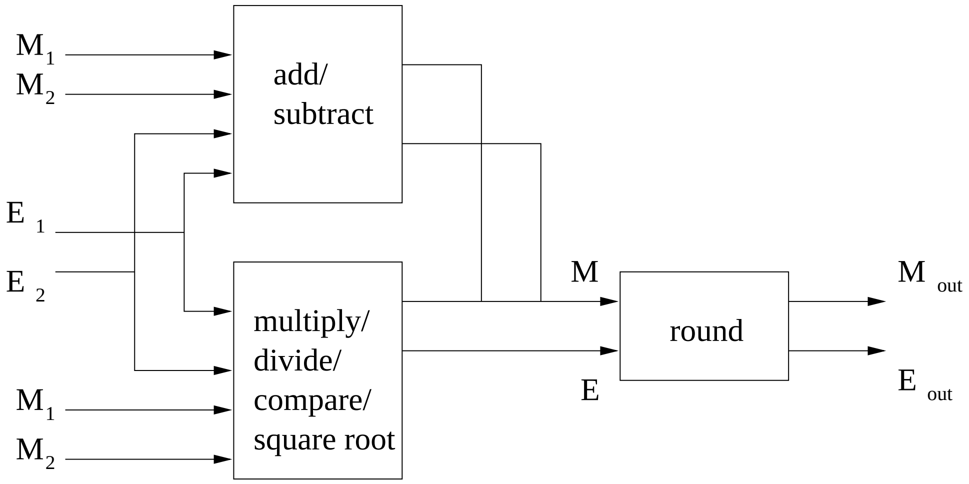

Fig. 1. Block Diagram for WEITEK Co-processor

Figure1 shows a block diagram of the Weitek FPU. It consists of circuits performing addition/ subtraction, multiplication, square root, division and rounding. The verification methodology presented here does not follow this structural decomposition, but rather verifies one arithmetic operation at a time. The exponent operation for all units but the

ALU are verified separately. The rounding operation and exponent adjustment during rounding is quite difficult to specify as part of the preceding arithmetic operation and is also verified separately. The FPU takes two floating-point operands (M1, E1) and (M2, E2) and generates a result floating-point number (Mout, Eout). In the following, we assume the mantissas to be bits wide, where is the internal precision of the machine. The verification of arithmetic operations is described in detail in the sequel. However, all the verification tasks follow a general methodology which is described here.

One general technique to tackle complex verification tasks is property decomposition, which is applicable at different levels of description. The task of verifying the correctness of the FPU is decomposed into properties asserting that individual FP sub-circuits compute the correct arithmetic functions. At a lower level, the property of the circuit is often further decomposed into arithmetic properties relating the mantissas and other properties of the exponents.

An FPU can be divided into a number of sub-circuits, each computing a particular arithmetic function on the operands. Depending on the machine instruction or micro-operation, different sub-circuits may be activated while others irrelevant to the operation are not used. Thus the model to be verified can be simplified with respect to the property being verified. For example, to verify the division circuit, only the division and the exponent sub-circuits need to be included into the model. In addition, for each sub-circuit and for each property to be verified with respect to the sub-circuit, a property-specific model extraction can be performed on the sub-circuit to simplify away the parts of the sub-circuit that is irrelevant to the property. The property-specific model extraction is done by an automatic tool that was independently developed at Intel. The tool seems to be very similar to the per-function reduction in [BBDEL96].

Several of the arithmetic algorithms used in the FPU are inherently iterative. For example, the square root and division computations begin by generating an initial result consisting of a few high order bits, then refining the initial result in successive iterations by computing lower-order bits until the desired precision is achieved. Iterative algorithms in general can be verified by proving an invariant, a technique borrowed from software verification. An invariant is a property relating the registers used in the computation during each iteration. The proof of an invariant has two parts: a proof that the initialization phase of the circuit causes the invariant to hold, and a proof that if the invariant holds before an iteration of the algorithm, then it continues to hold after the iteration. From these two results we can conclude that the invariant always holds, subject to the precision limitations of the physical registers.

To improve performance, floating-point circuits are often heavily pipelined. Verification of sequential hardware is in general more complicated than that for combinational ones. We can verify pipelined designs in two steps: combinational verification of their functionality and sequential verification of their pipeline control. For the former, we can remove the pipeline latches and treat the whole circuit as a single combinational block. Latch removal results in simpler specifications as well as more efficient verification. When we verify the pipeline control, we can abstract away the datapath as we are interested only in the sequential behavior of the control signals. For each pipeline latch, latch removal takes away the latch and its clock signal, and then reexpresses the latch output as a logic function of its inputs and its enable and reset signals. Note that latch removal can result in incorrect verification if the arithmetic circuit is inherently sequential or iterative in nature, as are division and square root.

Dynamic variable reordering is sometimes useful in verifying arithmetic circuits, especially for iterative circuits that contain a non-trivial control part. We have generalized Rudell's dynamic variable reordering algorithm [Rud93] to work on HDDs and incorporated it into our verification system.

A significant portion of the verification effort involved the user familiarizing himself/herself with the actual circuit description so as to be able to state properties correctly in terms of appropriate circuit signals, be able to ignore irrelevant parts of the circuit and be able to formulate the properties in a manner so as to best manage verification complexity.

3.2 Square Root

This section describes the verification of the mantissa computation for the square root operation. The Weitek paper states that the square root and division operations share a common datapath, but it leaves the details of the square root operation - for example, radix, unspecified [BSC+90]. For illustration, we consider here a non-restoring, radix-2 algorithm that is commonly found in the literature [BV85, OLHA95], and is implemented on a separate datapath.

The algorithm proceeds iteratively, as follows. The partial square root (proot) contains all the root mantissa bits computed thus far, and is used in each iteration to guess what the next partial root should be; the guess is always twice the partial square root plus the guess bit. The partial remainder (prem) contains the difference between the radicand mantissa and the previous guess squared. If the partial remainder is positive in a given cycle, then the square of the previous guess was less than the radicand, hence the most recently guessed bit was correctly presumed to be 1. If the partial remainder is negative, then the square of the previous guess was greater than the radicand, and so the most recently guessed bit was incorrectly assumed to be 1, and the partial root must be corrected accordingly. The guess bit (b) is initially and is shifted right two bits every cycle.

Because the square root algorithm is iterative in nature, we verify it by proving a loop invariant. In particular, we prove that the partially constructed root (proot), the partial remainder (prem), and the guess bit (b) have the following relationship at each iteration i:

$$prem_i < 0 \rightarrow -2 \cdot proot_i + b_i \leq prem_i$$
$$prem_i \geq 0 \rightarrow 2 \cdot proot_i + b_i > prem_i$$

We denote the conjunction of the above properties by INV_i. It can be proved mathematically that if the loop invariant is true at each iteration, when the algorithm terminates, the result is correct [OLHA95].

We prove the invariant by induction on the number k_M of iterations. According to the algorithm, in the 0'th iteration the registers are initialized as follows.

$$prem_0 = radicand$$

$$proot_0 = 0$$
$$b_0 = 2^{2 \cdot k_M}$$

The radicand is positive and less than $2^{2 \cdot k_M}$, so INV_0 should hold.

In subsequent iterations the algorithm updates the registers as follows.

$$prem_{i+1} = \begin{cases} prem_i - proot_i - b_i, & 0 \leq prem_i \\ prem_i + proot_i - b_i, & prem_i > 0 \end{cases}$$
$$prem_{i+1} = \begin{cases} prem_i - proot_i - b_i, & 0 \leq prem_i \\ prem_i + proot_i - b_i, & prem_i > 0 \end{cases}$$
$$b_{i+1} = b_i/4$$

For these later iterations we have to verify that if INV_i is true in a given clock cycle, and the registers are updated as above, then INV_{i+1} will be true in the subsequent clock cycle. That is, we use the word-level model checker to verify the following assertion.

$$INV_i \Rightarrow INV_{i+1}$$

These two results (the invariant holds initially, and is preserved by the register updates) prove that the invariant always holds.

Because the extended CTL used in the model-checking tool supports Boolean combinations of integer inequalities with subtractions, additions and multiplications, the invariant INV_i can be formulated "as is" for the model checker. To verify the properties, we must first automatically obtain a property-specific extraction of the model. In addition, to relate the values of variables in iteration i and $i+1$, we can introduce a set of "history" variables into the abstracted model that store the previous values of prem, proot, and b.

3.3 Division

In this section, we discuss the verification of floating-point division. Weitek WTL3170/3171 uses a radix-4 SRT division algorithm. The similar algorithms can also be found in [Fri61, Atk68]. Since the SRT division algorithm is also iterative, the loop invariant verification technique introduced in the previous section also applies here. Several published papers [CKZ96, BC95] have also shown how to verify radix-4 SRT division circuits using model checking techniques and thus our description here is brief. Given the mantissa d (from M1 in Figure 1) of the dividend and the mantissa b (from M2 in Figure 1), the radix-4 SRT division algorithm iteratively computes a partial remainder and a quotient digit . The partial remainder is initialized to and the quotient digit is initialized to zero. Each iteration the algorithm gets the quotient digit from a lookup table and subtracts from the partial remainder that has been shifted left by 2 bits. In other words, . The algorithm terminates when enough quotient bits have been computed. Suppose that the quotient digits are within the range for some positive . Then a radix-4 SRT division algorithm is guaranteed to be correct if both of the following properties are true in each division loop [Atk68]:

The loop invariant that we want to verify with our verifier is the conjunction of the two properties above. We want to verify that the invariant is true initially and also .

Since the quotient digits that WTL3170/3171 uses are -3, -2, -1, 0, 1, 2, 3, the second property simply becomes .

Again, can be expressed in the extended CTL very easily. Although the extended CTL does not support the division operator, we can easily transform some inequalities with divisions to inequalities with only multiplications. For example, the second property of (for any constant n) can be specified as follows.

The property-specific model extraction and variable ordering techniques discussed previously are also useful here to verify the loop invariant.

3.4 Multiplier

A floating-point multiplier consists of two parts, an integer multiplier and an exponent unit. This section describes the integer multiplier as shown in Figure 2. Depending on the precision, Weitek FP multiplier operates actually in two modes. We first describe the more common multiplier configuration for single precision. For double precision, multiplication is accomplished in two passes using the carry-save adder array. The verification of the double precision case is a straightforward extension and will not be described here.

As shown in Figure 2, the multiplier input M1 is first encoded through the Booth encoder. The value 3*M2 is produced by the 3X generator. Second, the multiplier selects which multiple (1X or 3X) of the multiplicand is used for each partial product. This step is called partial product selection. Third, the carry-save adder array adds all partial products to produce two numbers. Finally, an adder produces the final product. Note that for single precision, the multiplexer is controlled so that the output of the CSA array is not fed back as one of its inputs.

The overall property to verify the mantissa part of the multiplier is M = M1 * M2. To reduce the size of the HDDs required for this word-level model checking, a few decomposition techniques must be used. The Weitek multiplier is pipelined, so the pipeline latches must first be removed so that the multiplier circuit can be considered as a combinational function.

Even with word-level model checking, one has to be careful during the construction of BMDs to make sure that no intermediate computation results in an exponential representation. A simple-minded way of constructing the output BMD function M starts from the inputs and progressively builds the intermediate Boolean functions as BDDs. Once the BDD functions for each output bit are obtained, they are collected and composed into a word BMD function. This method unfortunately will not work on wide multipliers as there is no compact BDD representation of the output bits of a multiplier. This problem can be overcome by the following approach.

Hamaguchi et. al in [HMY95] proposed a backward construction method for obtaining a *BMD function. A cut is first defined across the output(s) and is swept towards the inputs by iteratively moving one gate across the cut at a time. A *BMD representing the function from the cut to the output is always maintained during the backward substitution. We improve on this method by not evaluating a new BMD function for each gate in the circuit. Instead we introduce the notion of auxiliary variables to mark multiple cuts on the circuit. Small intermediate BDDs are built to represent functions separated by auxiliary variables. We obtain the BMD representation of the output word M in terms of the auxiliary variables that are the immediate inputs of M. Then we backward substitute

[hbt]

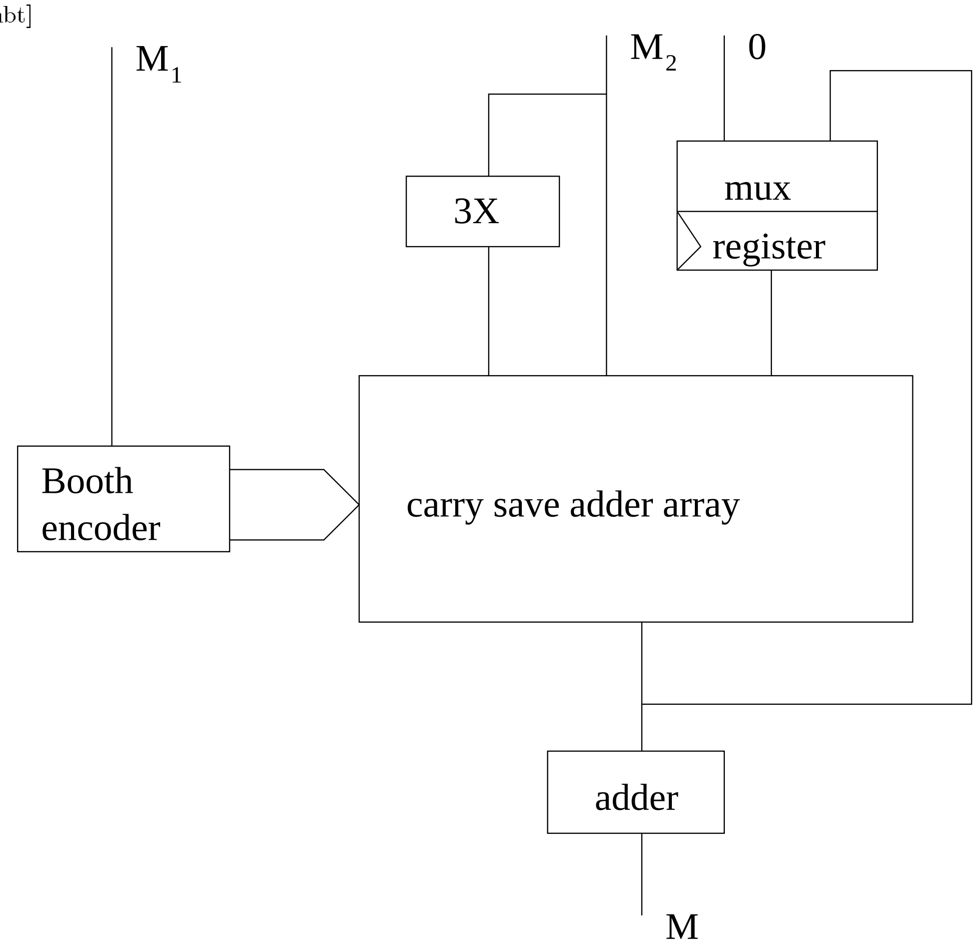

Fig. 2. Block Diagram of FP multiplier

the next intermediate function into the output BMD function, and continue until the latter is formulated in terms of primary inputs M1 and M2. This process is fully automatic. The simple M = M1 * M2 property can be verified directly on the circuit after unlatching and specification of auxiliary variables.

3.5 Adder / Subtracter

The block diagram in Figure3 shows the FP add/subtract circuit from the Weitek FP coprocessor. Note that it is more complicated than its integer counterpart. Given two floating-point numbers, they must first be aligned before their mantissas can be added/subtracted. This is done by comparing the relative magnitude of the two exponents and swapping (M1,E1) and (M2,E2) if E1 ¡ E2. It then shifts the mantissa with a smaller exponent —E1 - E2— places to the right. The larger exponent will become the exponent of the result. The result of mantissa addition will be within the bound [2, 4). If the sum is greater than two, the overflow mantissa must be shifted to the right. This is accomplished by the rounder circuit. On the other hand, after subtraction, it is possible that the resulting mantissa has leading zeros. Normalization is accomplished in such cases by multiple left shifting the mantissa. Let L be the amount of left shifting performed for normalization. The resulting exponent then must also be decreased by L.This is done in the add/subtract unit. Thus, the output of the adder/ subtracter is in the range [1,4).

[hbt]

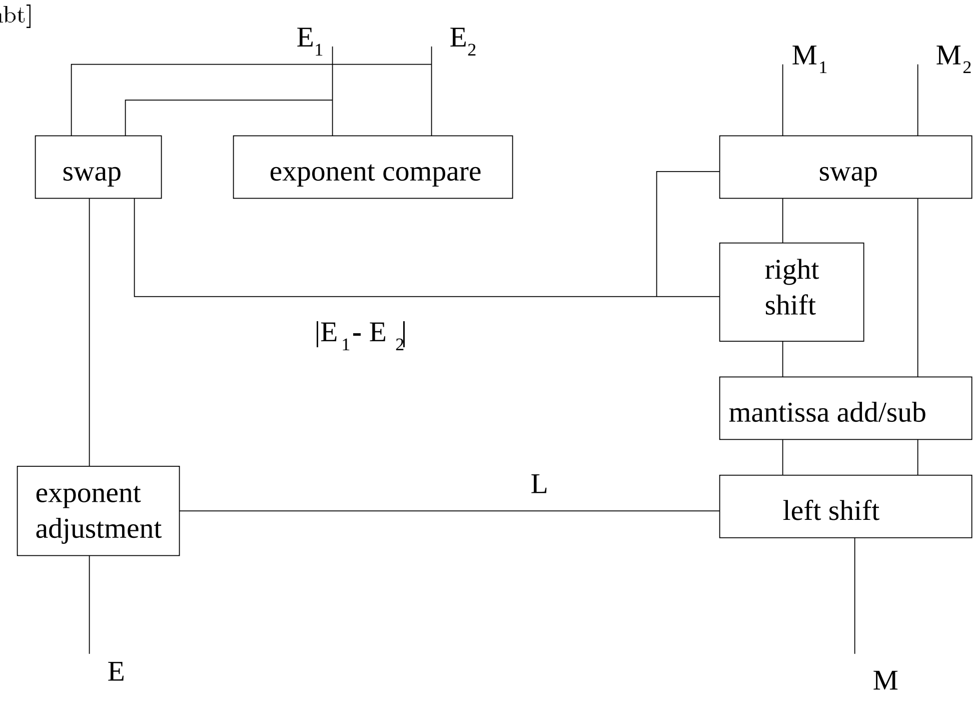

Fig. 3. Block diagram of FP adder/subtracter

The FP addition and subtraction properties can be decomposed and verified by case analysis. First, four combinations of signs of the two inputs are grouped into two cases: true addition and true subtraction. True addition and subtraction are the actual operations performed by the circuit. True addition includes addition of two numbers of the same sign, and subtraction between two numbers of different signs. Similarly, true subtraction refers to subtraction of two numbers of the same sign, and addition of two numbers of different signs. Furthermore, we decomposed each case into sub-properties, which are verified, according to the difference in the exponents.

3.6 Rounding unit

The result from a floating-point operation is finally fed to the rounding unit to be rounded so that it can be represented by a floating-point number of the required precision. The Weitek paper does not give the implementation of the rounding logic. However, it is the specification methodology that is of interest here and we describe it in this section.

In simple terms, we wish to verify that the output of the rounder is within one bit of the input. Thus, one approach is to specify the rounding operation as a relation between the input and output mantissa and exponents as follows (shown for single precision):

This specification has the advantage of being very general and independent of the specific rounding mode that is used. However, we have to split this specification into several cases to make the verification tractable. This splitting is most easily done on the basis of the rounding modes.

Most floating-point systems support four rounding modes for each precision, as specified in the IEEE floating-point standard:

- round to zero
- round to positive infinity
- round to negative infinity
- round to nearest/even.

The standard specifies that the result of an arithmetic operation should be the same as it would be if it were computed with infinite precision and then rounded using one of the specified rounding modes. To simulate the effect of infinite precision in the implementation, the rounding unit extracts a few extra bits from its input besides the fraction bits and the leading 1 bit (L). These extra bits are called the round bit (R) and the sticky bit (S). In a normalized input, the R bit is simply the bit to the right of the least significant bit (M0) of the mantissa and the S bit is the OR of all the bits to the right of the R bit. If the input to the rounder is not normalized, i.e., it is in the range [2,4), it is right shifted by one to bring it in the range [1,2) and the exponent is incremented before the rounding operation is carried out. In the following, we assume that the input mantissa is normalized. The final data format for normalized mantissa in single precision is shown in Figure 4.

[hbt]

Fig. 4. Input mantissa format for single presion

To specify the relation between the input and output of the rounding unit for each rounding mode, we compute an increment bit (I) for each rounding mode from the R and S bits. The relation between the input and output mantissa is then expressed in terms of the I bit. This enables a natural decomposition of the basic specification given earlier on the basis of the rounding modes since the computation of the I bit depends on the rounding mode. The desired behavior of the rounding unit can then be specified as follows: it adds the I bit to the M0 bit of the input mantissa. If this addition causes an overflow, the mantissa is shifted right and the exponent is incremented. The data is then chopped to the desired precision to give the final result mantissa.

The specifications for each mode then take the following form (shown for single precision), where shift is 1 if the addition of the I bit causes the result mantissa to be greater than or equal to 2, and 0 otherwise:

The computation of the I bit differs for each rounding mode and also depends on the sign of the input. Lack of space precludes a more detailed description of the specifications (see page A-24 in [HP96] for an example of I bit calculation). It is possible that the implementation also computes a similar I bit to help in the rounding. It is important that the computation of this I bit in the specification not mimic the logic for computation of the I bit in the actual implementation. This ensures that the specification is at a higher level of abstraction than the implementation.

The above specifications apply only to normal numbers where the exponent value falls in the acceptable range. In case of overflow or underflow, the setting of appropriate exception flags is verified. The actual data output by the rounder in such cases depends on the implementation. In any event, the above specifications can be modified for the special cases to appropriately reflect the desired behavior of the rounding hardware.

4 Experimental Results on an Intel Microprocessor

We applied all techniques discussed above to the FPU of an Intel microprocessor. The microprocessor performs all the floating-point operations mentioned in the previous sections in 64-bit extended precision. We are able to verify the entire floating-point unit of the microprocessor using our word-level symbolic model checking system. The work shows that our techniques do apply to arithmetic circuits found in actual industrial microprocessors. The table below summarizes the figures from the experiments.

The experiments were done on an HP 9000 workstation with 256MB RAM. REM is a partial remainder circuit that is verified using loop invariant techniques similar to those used in the verification of the division circuit. EXP is the exponent unit which produces the exponent result for the multiply, divide and square root operations. The second column shows the number of state variables in the property-specific extractions of the original designs. The automatic property-specific extraction of the design drastically reduced the number of state variables in several models which enabled the verification to succeed. The third column shows the number of properties verified for each macro-instruction. There are eighty properties verified for the rounder because several different cases have to be considered for different modes and precisions and verification of exceptions. The fourth column shows the maximum memory required for a verification run. The experiments show that all the verifications can be done on a machine with about 100MB of memory. The fifth column shows the maximum number of BDD nodes allocated by the symbolic model checker during the verification of all properties for the

Macro-Instruction	No. Of var. in extracted model	No. of properties verifies	Memory required	BDD nodes allocated	CPU time
DIV	287	4	18.8M	756K	194s
SQRT	415	16	18.5M	445K	239s
REM	369	8	9.8M	246K	1538s
MUL	1961	2	3.9M	1923K	508s
ADD	1251	2	22.1M	838K	660s
SUB	1247	9	96.0M	3947K	38525s
RND	295	80	23.6M	692K	2034s
EXP	65	4	8.3M	157K	26s

Table 1. Verification results on an FPU from an Intel microprocessor

macro-instruction.

The last column shows the CPU time spent on the verification of all properties for the macro-instruction. All the experiments except the verification of the subtracter can be done in less than an hour. The subtracter takes more time in comparison to the adder primarily because there are more cases to be considered.

5 Conclusions

In this paper, we have presented a methodology for the formal verification of a complete floating-point arithmetic unit and have shown the results of this methodology applied to a recent Intel microprocessor. In particular, we have verified the correct implementation of floating-point addition, subtraction, multiplication, division, remainder, square root and rounding operations in a fairly efficient and automated fashion. To the best of our knowledge, this is the first comprehensive effort of this magnitude in the verification of complex floating-point circuits in a state-of-the-art FPU design. The results presented here are important evidence of the capability of an automated model checking system.

Our verification uses the technique of word-level model checking. The experimental results show that it was highly effective in our difficult verification tasks. Its compact representation and the efficient manipulation of arithmetic functions is made possible by the word-level HDD representation. From our experience, word-level model checking can be performed fairly automatically and the specification is at an appropriate level of abstraction.

Different verification techniques were also discussed, all of which are crucial to the successful verification of the circuits covered. In this regard, the paper contributed a practical verification methodology for efficient verification of complex arithmetic circuits.

References

[Atk68] D. E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, C-17(10):925–934, October 1968.

[BBDEL96] R.E. I. Beer, S. Ben-David, C. Eisner, and Avner Landver. Rulebase: an industry-oriented formal verification tool. In *Proceedings of the 33rd Design Automation Conference*. IEEE Computer Society Press, June 1996.

[BC95] R. E. Bryant and Y. A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 535–541. IEEE Computer Society Press, June 1995.

[Bry91] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[Bry95] R.E. Bryant. Bit-level analysis of an srt divider circuit. Technical report, Carnegie Mellon University, 1995.

[BSC+90] M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes. Developing the wtl3170/3171 sparc floating-point coprocessors. *IEEE Micro*, pages 55–64, February 1990.

[BV85] J. Bannur and A. Varma. The vlsi implementation of a square root algorithm. In *Proceedings of the 7th Symposium on Computer Arithmetic*, pages 159–165. IEEE Computer Society Press, 1985.

[CFZ95] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams – overcoming the limitations of mtbdds and bmds. In *Proceedings of the 1995 Proceedings of the IEEE International Conference on Computer Aided Design*, pages 159–163. IEEE Computer Society Press, November 1995.

[CKZ96] E. M. Clarke, M. Khaira, and X. Zhao. Word level symbolic model checking – a new approach for verifying arithmetic circuits. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1996.

[CMZ+93] E. M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 54–60. IEEE Computer Society Press, June 1993.

[Coe95] T. Coe. Inside the pentium fdiv bug. *Dr. Dobbs Journal*, 20(4):129–135, April 1995.

[Fri61] C. V. Frieman. Statistical analysis of certain arithmetic binary division algorithms. *IRE Transaction*, pages 91–103, January 1961.

[HMY95] K. Hamaguchi, A. Morita, and S. Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proceedings of the 1995 IEEE International Conference on Computer Aided Design*, pages 78–82, November 1995.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[KL93] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proceedings of the Fifth Workshop on Computer-Aided Verification*, June/July 1993.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[OLHA95] J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: a case study in design by principled optimization. In *Proceedings of the Theorem Provers in Circuit Design '94*, volume 901 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Intl. Conf. on Computer Aided Design*, Santa Clara, Ca., November 1993.

[SB94] H. P. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the pentium processor(1994). Technical report, Intel Corporation, November 1994.

[VCM94] D. Verkest, L. Claesen, and H. De Man. A proof of the nonrestoring division algorithm and its implementation on an alu. *Formal Methods in System Design*, 4:5–31,

January 1994.