FPgen – A Test Generation Framework for Datapath Floating-Point Verification

Merav Aharoni, Sigal Asaf, Laurent Fournier, Anatoly Koifman, Raviv Nagel
IBM Haifa Research Labs
sigalas@il.ibm.com

Abstract

FPgen is a new test generation framework targeted toward the verification of the floating-point (FP) datapath, through the generation of test cases. This framework provides the capacity to define virtually any architectural FP coverage model, consisting of verification tasks. The tool supplies strong constraint solving capabilities, allowing the generation of random tests that target these tasks. We present an overview of FPgen's functionality, describe the results of its use for the verification of several FP units, and compare its efficiency with existing test generators.

1. Introduction

Traditionally, achieving IEEE compliance for FP hardware [1] in microprocessors has been a challenging task. In particular, verifying the correctness of a single FP instruction presents special difficulties. These difficulties stem from the complexity of the test space, which is enormous and consists of numerous corner cases that are very difficult to pinpoint. FP bugs are often uncovered in late stages of the design life, resulting in costly changes, or in listing as errata.

Mainstream test generation tools [2,5] provide some means for verifying FP implementations. However, their lack of focus and internal knowledge related to the FP domain, render them inadequate for providing a practical, let alone complete, solution for FP verification. Even existing test generators that focus on FP verification [8,11,13] have limited power, since they each focus on one type of task. The IEEE test suites [16] are also widely used in the industry. These test suites provide good coverage of IEEE corner cases. However, they target only the IEEE standard and do not target the implementation. In addition, these tests are static, and contain no random elements.

Formal verification (FV) for FP has evolved significantly over the last decade [4,6,9]. FV techniques can be used to prove instruction correctness, but they require a significant investment of machine and manual work time. They are typically not adequate for the debugging stage, but can be very effective in providing assurances of final correctness[17]. Therefore, an optimal combination of the two techniques may be to first use simulation techniques to reach a reasonable level of confidence, and then provide a formal proof via FV.

This paper presents FPgen, a new test generation framework targeted toward the verification of the FP datapath. FPgen is a generic tool; it primarily targets architectures that comply with the IEEE Standard 754 [1], but it can also be used for architectures that deviate from the standard. The goal of this paper is to convey a basic understanding of what this test generation framework provides, and how this functionality promotes the verification process.

When dealing with FP verification by simulation, the main challenge is to generate a set of tests that comprises a *representative sample* of the entire space, taking into account the many corner cases. The framework we present in order to deal with this challenge consists of two components. The first is to define sets of representative coverage models [3], each of which consists of a set of related tasks. The second part is to generate, for each such task, tests that fulfill this task. FPgen provides a solution for the second component.

We illustrate this process by an important example: the "**All Types**" model enumerates over all major IEEE FP types simultaneously for all operands of all FP instructions. These types may include the following: +/-zero, +/-min denorm, +/-denorm, +/-max denorm, +/-min norm, +/-norm, +/-max norm, +/-infinity, +/-qnan, and +/-snan. For a given instruction with three operands, this potentially yields 8000 (20^3) cases that must be covered, assuming 20 major FP types. Obviously, not all cases are possible (e.g., the addition of two positive numbers can't be negative), so the actual number of cases is, in fact, lower. While this model is very simple to define, it is far from trivial to fulfill.

A coverage model, or the set of all coverage models, is an attempt to partition the set of all the calculation cases. FPgen provides *coverage by generation*, where it takes a definition of a coverage model as input, and outputs a set of tests that cover all the achievable tasks of the model.

FPgen offers a convenient platform that consists of a language for the definition of the verification requirements, and powerful solving engines that generate random, different cases for each given task. The context of randomness is important, because bug locations are mostly unpredictable.

FPgen is concerned with biasing and generating ***operand data*** for FP instructions. A bias (or constraint) on an operand data is a set of values to which the operand data is constrained. Resolving biases on input operands is usually straightforward, though it is sometimes hard to obtain uniformity among all the solutions. In contrast, resolving constraints on the data for the *intermediate result(s)* and for the *output* of instructions, adds a layer of complexity due to the instruction semantics involved.

This paper is organized as follows. Section 2 provides a high-level description of the FPgen framework. In Section 3, we describe the solving engines used by FPgen to solve the constraints. Section 4 presents our results, both in the verification of FP Units and in experimental results. Conclusions are presented in Section 5.

2. FPgen High Level Description

FPgen is a random test generator that receives a request for FP tasks as input and outputs a random test, or a set of tests, fulfilling the request.

The primary focus of FPgen is to solve data constraints on operands of individual FP instructions. A data constraint on an FP operand is defined as a set of all values that can be selected for this operand. An individual instruction can have data constraints for each one of its operands. When constraints are requested on all the operands (inputs and outputs) of an instruction, we say there is a *full constraint* on the instruction. Solving all the instruction constraints is equivalent to selecting a value from each given set, such that the instruction semantics are satisfied. This should be done with uniform probability, so that each solution has the same probability of being selected. The randomness is important because the constraints only reflect a suspected area. One instance in this area may reveal a problem, while another will not.

FPgen's main challenge is to provide solving engines that solve these constraints within a reasonable amount of time. Because it is clear that many of the constraints yield NP-complete problems, it cannot be expected that all problems will be solved in polynomial time. However, FPgen's solving engines attempt to ensure that only a small fraction of the problems require long periods of time to be resolved.

We introduce the different types of constraints for a single task in Section 2.1, and present the generalization to coverage models in Section 2.2.

2.1 Single Task

The general structure of a single task is:

FPinst, Op1 in Set1, Op2 in Set2, IntRes in Set3, Result in Set4, Operand Relation

where FPinst is an FP instruction with two input operands (Op1 and Op2), one intermediate result (IntRes), and one result, and where the operand relation holds constraints involving several operands. Input and result operands are defined by the architecture, while the intermediate result depends on the implementation (see below). The case of two operands and a single intermediate result is presented here for the sake of simplicity; generalization to any number of such parameters is straightforward. For example, we can define the following constraint on the add instruction: Op1 is a normalized number, Op2 is a negative number, the intermediate result is unconstrained, the final result is a denormalized number, and the exponent of the final result is smaller than the exponents of either Op1 or Op2 by at least 10.

FPgen provides multiple means to define sets of FP numbers:

1. **Range of FP numbers** – for example [+min denorm, +max denorm]. This is a convenient way to define the basic FP types (such as denorm), and to target the vicinity of critical FP values.
2. **Mask** – each bit can be specified as 0, 1, or X, where X signifies "don't care". For example, the set +-Zero is defined by X00…00. This provides a means of targeting the neighborhood (in terms of Hamming distance) of critical FP values. An important use of masks is for constraining the lower bits of the intermediate result to test correct rounding.
3. **Number of 1s/0s** – constrains an FP number to contain between min and max bits equal to 1 (or 0). A large number of 1s or 0s may trigger a special behavior in the micro-architecture.
4. **Sequence of 1s/0s** – this is similar to the previous constraint, but focuses on sequences.

In addition, the language includes set operations (i.e., union, intersect, and complement). The set language definition serves to conveniently express constraints that emanate from typical tasks in a verification plan.

An expression language allows the definition of different relations between the operands' fields (sign, exponent, and fraction). For example, using this language, the following constraint can be defined:

Result.exponent=max(Op1.exponent,Op2.exponent)-20

In this particular situation, the result exponent is much smaller than the input exponents. This checks a corner case for add and subtract instructions, where a renormalization operation is applied to the intermediate result.

The intermediate result is a special operand added to the operand list of the instruction as follows: The IEEE Standard requires that each operation is performed in an intermediate result as if it had infinite precision and unbounded range. This intermediate result should then be corrected to fit into the destination's format. In practice, implementations define an internal FP number that has a wider exponent and fraction. This enables the calculation of the output, as though the intermediate has infinite precision. FPgen deals with this intermediate result as one of the instruction operands.

The size of the mantissa is a parameter that can be controlled by the user. The size of the exponent is defined such that neither underflow nor overflow will occur. This way, all the defined constraints can be applied to the intermediate result as well, such as a request for an exact result or a result exactly in the middle of two FP numbers. Such control is important as some bugs are often the result of faulty handling of specific intermediate results. Analytic algorithms that solve these types of constraints for selected patterns and instructions are described in [8,11,13].

2.2 Coverage Model

A coverage model is a set of related verification tasks, such as the "All Types" model described in Section 1. For the sake of comprehensiveness and efficiency, test plans usually require that full coverage models are fulfilled, rather than fulfilling a list of disparate tasks. FPgen allows the user to directly request the fulfillment of coverage models, so that the model can be given as a single input request.

A coverage model for a two-input instruction is defined in FPgen as follows:

FPinst, Op1 list, Op2 list, IntRes list, Result list, Relation list

where each operand list consists of a list of FP sets, and the relation list consists of a list of constraints among the operands.

The tasks yielded by this model come from the Cartesian product of the different lists, i.e., all possible combinations of selections from the different lists. The number of tasks is calculated based on the multiplication of the number of sets and relations for each participating list. Since Cartesian products often include illegal cases (as in the "All Types" model), FPgen allows the definition of restrictions on the model. These restrictions represent the set of cases for which FPgen is not requested to find solving instances. In addition, FPgen provides the ability to control the number of instances required for each task.

3. Solving the Constraints

The heart of FPgen lies in its solving engines. In this section, we explain how FPgen solves various data constraints and we present the overall picture of the different types of engines that FPgen employs to deal with a variety of constraints.

Resolving constraints over FP numbers involves a different type of mathematics from the one over real numbers. Table 1 illustrates a simple example, assuming FP numbers with a 4-bit binary fraction.

Table 1: Unsatisfiable Constraint in FP Arithmetic

Inst	Op1	Op2	Output
Subtract	[$1.0001*2^{10}$, $1.0010*2^{10}$]	[$1.0001*2^{10}$, $1.0010*2^{10}$]	[$1.0000*2^{2}$, $1.0000*2^{5}$]

Over the FP domain, the input ranges include only two numbers ($1.0001*2^{10}$ and $1.0010*2^{10}$), yielding three possible outputs of the subtract operation: 0, $1.0000*2^{6}$, and $-1.0000*2^{6}$. Clearly, there are solutions over real numbers (for example, $1.0010*2^{10} - 1.00011111*2^{10} = 1.0000*2^{2}$), but no solutions over the FP numbers. While over the real numbers, the number of digits in the fraction is unlimited, in an FP number, the number of digits is limited to the precision, which is, in this case, four. Therefore, any possible FP result satisfying this constraint must have an exponent of at least 6.

Constraints can be given on input operands, output operands, or on both simultaneously. Solving constraints on output operands, as opposed to input operands, involves the instruction semantics and adds significantly to the complexity of the problem. Constraint restrictions can become intractable when simultaneous constraints are requested on both input and output operands. Many instruction constraints can be shown to yield NP-complete problems.

FPgen uses multiple engines in the solving phase. After first analyzing the constraint, FPgen directs the task to the appropriate engine. It uses two major types of engines:

1. **Analytic engines (A-engines) –** guaranteed to find a solution within a reasonable amount of time (ranging from 1 to 3000 tasks per minute), if it exists. The output of A-engines is either a random solution or a message that none exists. These engines handle constraints on inputs or output separately. They also handle relation constraints between the exponents of the input and the output over add and multiply-add instructions. Finally, a few special cases of full constraints are also solved by A-engines.

Most of these engines are based on mathematical algorithms and properties of FP numbers. Rules that hold true for real numbers are not necessarily true for FP numbers. For example, for real numbers, a+b=c implies that c-b=a. This is not necessarily true for FP numbers. This occurs because of the rounding operation. This fact is taken into account in all the algorithms used to solve constraints involving the result of an instruction. The algorithms employed for solving constraints are based on a wide variety of mathematical theories and approaches. One example is the use of group theory in order to solve constraints on the lower half of the bits of the intermediate result of the multiply instruction. Another example is the engine that solves constraints on the intermediate result of the divide instruction, which is based on continued fraction theory. Two of the constraint solving algorithms are discussed in detail in [7, 14].

2. **Search engines (S-engines) –** based on search methods such as SAT [10] and Hill Climbing [12] techniques. These engines are used on constraints for which no A-engine exists. S-engines are dedicated to solving full constraints, since they are the only ones for which A-engines are not always available. Typically, S-engines have heuristic search solutions, but their success, within a reasonable amount of time, is not guaranteed. Therefore, they may return with a “quit” message, indicating that no solution has been found, although one may exist.

We summarize the types of constraints and the main existing solvers that currently exist in FPgen:

1. Constraints on the inputs: analytic engine.
2. Constraints on the output only: different analytic engines exist for different instructions, and for final and intermediate results.

3. A relation constraint between the exponents of the input and the output over add and multiply-add instructions: different analytic engines exist for add and for multiply-add.
4. Full range or mask constraints: described in [7,14].
5. Additional types of full constraints are handled by search engines. The most important of these are: (a) SAT solver: the constraint is translated to a CNF formula, which is then solved by a SAT solver (e.g., Zchaff [10]) modified to find a random solution. (b) Hill Climbing: this engine uses a space search similar to Hill-Climbing or simulated annealing [12]. This is an iterative process, in which we start from a random FP number and then successively flip one bit at a time, according to an instruction-dependent heuristic function.

Depending on the instruction and the type of constraints, FPgen will opt for either A-engines or S-engines. When FPgen selects the A-engine path, it chooses a specific engine. The scheme differs slightly when FPgen takes the S-engine path. First, a time limit must be fixed. FPgen opts to the search engine that offers the best fit to the specific constraint. If the chosen engine doesn't find a solution after the time limitation expires, FPgen may try another search engine. In the future, we will add a mechanism to transfer the problem to *all* the appropriate S-engines, *in parallel*. The first engine to hit a solution or prove that no solution exists will terminate the process. Fig. 1 describes the scheme.

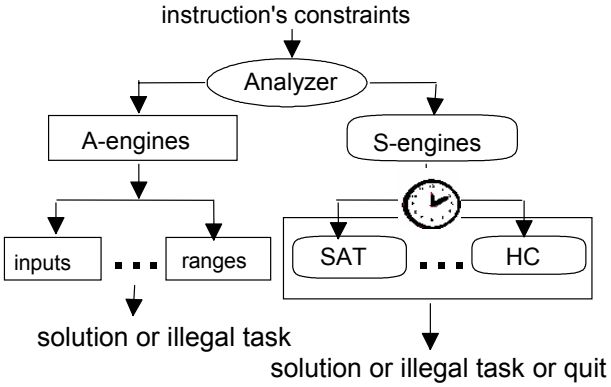

Figure 1: FPgen Engine Scheme

4. Results

This section presents empirical evidence about FPgen's performance. We describe the results of its use in the verification of several FP units and compare its efficiency with existing means of test generation.

4.1 Verification of FP Units

FPgen has been used for the FPU verification of six different FP units. Its use is driven by a comprehensive test plan composed of over 50 coverage models, which comprise several million tasks. FPgen offers the only way to directly target these models. Other solutions require a significantly longer amount of time and often do not attain full coverage (see 4.2).

In the course of its use, FPgen uncovered a large number of bugs at all stages of verification. However, the full strength of FPgen is manifested in uncovering bugs that were not detected by other means, sometimes at very mature stages of the design.

An interesting example of such a bug was discovered during the verification of a design whose implementation was based on two previous designs. Upon testing, we discovered that these designs also contained the bug, although their verification had already been completed, and they already had functional silicon.

The bug was in single precision multiply-add and involved an extreme case of cancellation of bits. The exponent of the result was smaller than the maximum between the exponents of the addend and the product by exactly 47, which is the maximum possible difference between these exponents. In this particular case, the design returned an incorrect result. FPgen found this bug by covering a model that targets all such possible exponent differences.

Another example of a bug found by FPgen is a divide bug, in which the normalized intermediate result is exact, the LSB is one, and the unbiased exponent is -1023 (max-denorm exponent). In this case, the inexact bit was supposed to be set but it wasn't. FPgen detected this case by covering a model that enumerated over the LSBs of the intermediate result, combined with an enumeration over the basic types of FP.

4.2 Experimental Results

The following experiment demonstrates the efficiency of FPgen for a typical coverage model. We recorded and compared the relative efficiency of different generation tools in covering the "All Types" model described in section 1 for the instructions: add, multiply, and divide. The model has 1586 plausible tasks. For comparison, we used the test generator Genesys Pro (Gpro), which is the successor of Genesys [2,5]. We applied Gpro in three different contexts:

1) a completely random manner (Rand), showing the model's relative distribution;

2) a mode where the inputs are biased toward the types defined in the model (IB);

3) using Gpro's specific Testing Knowledge (TK), biased toward the outputs required in the model.

We also show the combined results of all three approaches.

Any test generator can cover a significant part of the model by simply ignoring the output, and covering all 20x20 input combinations. Because the output must belong to one of the FP types, each such test covers a task defined in the model. In this way, 400 tasks are immediately covered for each instruction.

We ran each mode until we reached a steady state where the rate of covering new tasks approached zero. Table 2 and Fig. 2 summarize the results. The graph in Fig. 2 shows the progress for the first 20000 tests for each method. The summarized information can be seen in the table.

Table 2: Experimental Results

Mode	Time (hours)	Number of Tests	Covered Tasks	Coverage (%)
Rand	70	330000	135	8.5
IB	10	20000	1325	83.5
TK	185	1100000	1283	80.9
All three	265	1450000	1435	90.5
FPgen	0.1	1586	1586	100

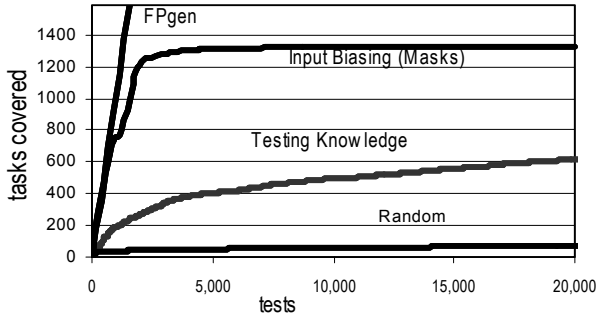

Figure 2: Experimental Results

FPgen finished covering the model in six minutes, while the other methods covered only about 90 percent of the model, even after running for several days. Not only does FPgen provide solutions for the legal tasks, it also identifies the other tasks as unsolvable. In this way, FPgen serves as a coverage tool, in addition to its generation capabilities.

While FPgen reached a completely homogeneous coverage (each task was hit exactly once: 1/1586=0.063%), the coverage distribution reached by the different Gpro modes is highly non-homogeneous: some of the tasks were hit many times, while others were not hit at all.

5. Conclusion and Future Work

FPgen is a test generation framework for implementing simulation-based FP datapath verification. It has been used to verify several FP-Units and has assisted in uncovering many interesting bugs. Our experimental results emphasize the advantages of using FPgen in FPU verification. FPgen easily covered tasks that were previously very hard to hit.

The FPgen framework was developed to support IEEE standard compliant architecture as a first goal. However, the underlying ideas governing the approach can be applied to any architecture. In fact, the FPgen framework already supports a large array of non-IEEE features, such as the multiply-add instructions and common IEEE standard deviations such as the absence of denormal numbers.

FPgen has been successfully integrated into existing verification processes by providing a means to easily fulfill current test plans. The next step is to observe that, given the power of test generation offered by FPgen, test plans themselves can and should be extended. For this purpose, we are establishing a comprehensive test plan for IEEE Standard 754.

Additional directions for our work include support for decimal FP [15], and the continuous extension of the range of problems solvable by FPgen. We will research the mathematical properties of FP numbers to create additional A-engines and apply new search algorithms to create improved S-engines.

6. References

[1] *IEEE standard for binary FP arithmetic.* An American National Standard, ANSI/IEEEE Std. 754-1985.

[2] A. Aharon et al. "Test Program Generation for Functional Verification of PowerPC Processors in IBM". In DAC, 1995, pp. 279-285.

[3] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. "User defined coverage - a tool supported methodology for design verification". In DAC, 1998, pp. 158-163.

[4] Edmund M. Clarke, Steven M. German and Xudong Zhao. "Verifying the SRT Division Algorithm Using Theorem Proving Techniques". In Formal Methods in System Design, 14(1), 1999, pp 7-77.

[5] L. Fournier, Y. Arbetman, and M. Levinger. "Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator. Application to the x86 Microprocessors Family". In DATE99, 1999, pp. 434-441.

[6] R. Kaivola and N. Narasimhan. "Formal Verification of the Pentium 4 FP Multiplier". In DATE02, 2002, pp. 20-27.

[7] L. Fournier, A. Ziv. "Solving the Generalized Mask Constraint for Test Generation of Binary FP Add Operation". Theoretical Computer Science, Special Issue: Real Numbers and Computers, Vol. 291/2, 2003, pp. 183-201.

[8] M. Parks. "Number-theoretic Test Generation for Directed Rounding". In Proc. Computer Arithmetic, 1999, pp. 241 - 248.

[9] K. L. Mcmillan. "Fitting Formal Methods into the Design Cycle". In CAV, 1994, pp. 314-319.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an Efficient SAT Solver". In DAC, 2001, pp. 530-535.

[11] W. Kahan. A Test for Correctly Rounded SQRT. //www.cs.berkeley.edu/~wkahan/SQRTest.ps , 1996.

[12] S. Russel, P. Norving. *Artificial Intelligence: A Modern Approach.* pp 111-115.

[13] L. McFearin and D. Matula. "Generation and Analysis of Hard to Round Cases for Binary FP Division". In Proc. Computer Arithmetic, 2001, pp. 119-126

[14] A. Ziv, M. Aharoni, and S. Asaf. "Solving Range Constraints for Binary FP Instructions", In proc. ARITH16, 2003, pp 158-164.
[15] M. Cowlishaw, E. Schwarz, R. Smith, and C. Webb. "A Decimal FP Specification", In Proc. Computer Arithmetic, 2001, pp. 147-154.
[16] B. Verdonk, A. Cuyt, and D. Verschaeren, "A Precision and Range Independent Tool for Testing FP Arithmetic: basic operations, square root and remainder". ACM TOMS, Vol. 27, Number 1, 2001, pp. 92-118.
[17] R. Goering, "Formal tools won't replace simulation, experts say", http://www.eetimes.com/story/OEG20030606S0017, EE Times.