H-0140 October 10, 2002
Computer Science

IBM Research Report

FPgen - A Deep-Knowledge Test-Generator for Floating Point Verification

Laurent Fournier, Sigal Asaf
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

FPgen – A Deep-Knowledge Test-Generator for Floating Point Verification

Laurent Fournier and Sigal Asaf
IBM Research Laboratory in Haifa
laurent@il.ibm.com, sigalas@il.ibm.com

Abstract

This paper describes FPgen, a test-program generator for datapath verification of floating-point units in microprocessors. FPgen is a convenient and powerful platform for generating interesting data combinations for floating-point instruction operands. Its main purpose is to provide a means to fulfill floating-point test-plans which typically include a myriad of tasks that stem from both the architecture and the micro-architecture. FPgen focuses on the IEEE standard definitions and therefore supports architectures that comply with these definitions.

1. Introduction

Traditionally, achieving IEEE compliance for floating-point hardware in microprocessors has been a challenging task. Many escape bugs, including the infamous Pentium bug [16], belong to the floating-point unit and reveal that the verification process in this area is still far from optimal. The growing demand for performance, quality, and faster time-to-market causes the verification work to become increasingly difficult. Problems in the floating-point unit implementation have many sources. These problems range from data operations on individual instructions, to the correct handling of sequences of instructions in which back-to-back events challenge superscalar implementations. The complexity stems both from the richness of the specification (architecture) and from the peculiarities of the implementation (micro-architecture).

Verification has traditionally been handled by simulation of test-programs [3,5]. Recently, the area of formal methods has evolved significantly, especially for the floating-point datapath [9,10,11,13]; however, it is nevertheless far from providing a comprehensive answer to the problem.

In most environments, the simulation of test cases is still a cornerstone of the verification process. This paper presents FPgen, a new test generator developed in IBM, targeted toward the functional verification of the datapath of floating point units in microprocessors. FPgen is designed to yield a quasi-optimal framework for the generation of test cases in this area.

When dealing with floating point verification by simulation, there are an enormous, practically unbounded number of different calculation cases which need to be tested. In practice, simulation can be done on only a very small portion of the existing space. The rationale behind verification by simulation is that one acquires confidence in the design correctness by running a set of test cases that exercise a sufficiently large number of *different* cases, which in some sense are assumed to be a *representative sample* of the entire space. It is inferred that the correct handling of the tested cases is a testimony to the design correctness of all the cases. The difficult question is: "How should such a representative set of test cases be built?" Since both the architecture specification and the micro-architecture implementation yield a myriad of special cases, pure (uniform) random generation of test cases would be largely inefficient. As described below, FPgen offers full control to bias test generation so it reflects the underlying distribution of interesting cases.

How does one know that a certain set of tests is sufficient? This question is related to the notion of coverage, which defines the comprehensiveness of the set related to the verification target [6,7,8]. Usually, *coverage models*, which are sets of related tasks, are defined and the set of tests should fulfill all the defined tasks. For example, a common - albeit far from trivial to fulfill - coverage model is one that requires enumeration of all major IEEE floating point types, simultaneously, for all operands of all floating-point instructions (the "All *IEEE Types" model*). For a given instruction with three operands, this potentially yields a thousand (10**3) cases that must be covered, assuming 10 major floating-point types (+/- NaNs, +/- Infinity, +/- Zero, +/- Denorm, +/- Norm). Table 1 illustrates this model, with a few common floating point instructions. This model can be further refined by including additional floating-point types, such as minimum and maximum Denorm, etc. Obviously, not all cases are possible (e.g., the addition of two positive denorm numbers cannot reach infinity), so that the actual number of cases is in fact lower than the size of the full Cartesian product.

Instruction	Op1	Op2	Output
Fadd	+/- NaN	+/- NaN	+/- NaN
Fsub	+/- Infinity	+/- Infinity	+/- Infinity
Fmul	+/-0	+/-0	+/-0
Fdiv	+/-Denorm	+/-Denorm	+/-Denorm
Fsqrt	+/-Norm	+/-Norm	+/-Norm
etc.			

Table 1: The "All IEEE Types" Coverage Model

A coverage model, or the set of all coverage models, is often an attempt to partition the set of all the calculation cases in such a way that the probability distribution is uniform over all subsets. As explained below, FPgen provides *coverage by generation*, i.e., it takes the request of a coverage model (such as the one in Table 1) as input, and outputs a set of tests that cover all the achievable tasks of the model.

The verification process is commonly defined and monitored through a verification plan, which leads to multiple verification tasks. FPgen's main purpose is to provide a convenient platform for performing all these tasks, both *qualitatively* and *quantitatively*. On the one hand, it should enable the most complex verification requirements, involving intricate sets of constraints, to be fulfilled. On the other hand, it should enable the generation of a practically unlimited number of different cases for each given set of constraints. FPgen provides these two properties in a context of randomness. This is meaningful since bug locations are mostly unpredictable. Thus, FPgen provides a comprehensive, quasi-optimal simulation-based solution for datapath verification of floating point implementations. While its scope is not limited to a certain architecture or design, its first target is the IEEE Standard for Binary Floating Point Arithmetic [1], and architectures that comply with this standard.

FPgen provides a convenient platform for biasing and generating ***operand data*** for floating point instructions. Simply put, 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 relatively straightforward, even though uniformity among all the solutions is sometimes extremely hard to obtain. In contrast, resolving constraints on the data, for both the *intermediate result(s)* and the *output* of instructions, adds a layer of complexity since it involves instruction semantics. FPgen's scope, however, goes beyond the single instruction domain, and includes the generation of ***sequences*** of instructions possibly driven by the definition of ***coverage models***.

FPgen belongs to a family of *Deep-Knowledge Test Generators*. These generators, developed in IBM, focus on specific areas not sufficiently stimulated by other means. This family includes test generators for micro-architecture flow and for Memory Management (Address Translation) Unit verification.

1.1. Outline

This paper is organized as follows: Section 2 surveys the current state-of-the-art and Section 3 provides a high-level description of the tool. Section 4 describes the solving engines, Section 5 introduces a series of additional features, and Section 6 discusses the concept of coverage by generation. Section 7 concludes the paper.

2. Motivation

Traditionally, floating-point verification has been undertaken through simulation of test programs. Recently, formal verification is being applied to this domain. While these two complementary, yet competing technologies are applied to various extents in the industry, floating-point verification is still achieved primarily by running test programs. Mainstream test generation tools [3,5] do provide some means for verifying floating-point implementations. However, their lack of focus and internal knowledge related to the floating-point domain, render them inadequate for providing a practical, let alone-complete, solution to the floating-point verification problem. Even existing test generators that focus on floating points [17,24] are of limited power, especially for the control of output operands. The common drawback of existing test generators is that new procedures must be written each time an additional event is deemed important and needs to be generated with reasonable probability. Typically, such procedures are difficult to write, and require an in-depth knowledge of the floating-point domain.

Other sources of tests widely used in the industry are the IEEE test suites. These test suites are an important quality threshold, and assist in reaching confidence in the design correctness. The tests provide good coverage of IEEE corner cases. However, the bug-free running of these suites is necessary, but far from sufficient, since their scope is confined to the IEEE standard and the implementation itself, with its host of specific cases, is not targeted.

The second technology, Formal Verification (FV), has evolved significantly during the last five to ten years. While FV has traditionally been more focused on the verification of control paths, it has lately started to target floating-point datapath verification as well. Verification of control paths [20] [21], is improving, but still collides with the BDD size explosion problem relatively fast as the control block size increases. Two new approaches, both relying on theorem proving [18] technology, have emerged for datapaths. The first approach translates the data-path circuit into the Theorem Prover's language. The proof of the circuit's correctness is done entirely with the Theorem Prover, by proving a succession of manually proposed and deducted lemmas (e.g., the work in [9,10] with ACL2). Because of the circuit's complexity and the Theorem Prover's limitations, parts of the proof are performed manually [9,10]. The second approach combines traditional FV tools with the Theorem Prover. The circuit is split into blocks and each block is verified by traditional FV tools. The Theorem Prover then combines the blocks into one unit [11]. Both approaches require a significant amount of manual work by experts in mathematics, floating-point, formal methods, and FP micro-architecture. While FV solutions are very attractive, they still suffer from three significant drawbacks:

- Expensive in terms of time, especially expert time.
- Involve many manual steps. This explains the presence of bugs even after these proofs are performed [11].
- Not applicable to all FP instructions or for verifying a sequence of FP instructions.

Therefore, at the very least, simulation must still be performed in parallel to any FV effort.

Due to the incompleteness of existing technologies, additional ad-hoc tools and scripts are often sporadically used to cover the verification scope. Clearly, the situation is far from optimal. In parallel, the complexity of floating-point units continues to increase at a rapid pace, to meet demands for increased performance and aggressive targets on FP benchmarks. Given this state of affairs, it is not surprising that the floating point area remains a significant source of escape bugs in microprocessor designs.

3. FPgen High Level Description

This section provides a general overview of FPgen's functionality. FPgen is a random test generator which receives a request for a certain floating-point task, or tasks, as input, and outputs a test, or a suite of tests, that fulfills the request. The test format consists of three sections: (1) initial state of resources, (2) sequence of instructions, (3) expected results of resources. Expected results are computed via a behavioral simulator [3].

Although the overall scope of FPgen includes, among other things, instruction sequences and coverage models, its primary focus is the solving of data constraints on operands of individual floating-point instructions. An individual data constraint on a floating-point instruction operand is defined as a set of values that can be selected for this operand. It must be a subset (not necessarily proper) of the set of all possible values. Restricting the set of values of an input operand to the denormal numbers is an example of a data constraint on the operand. An individual instruction can have as many data constraints as it has operands. When constraints are requested on all operands (inputs and output) of an instruction, we say there is a *full constraint* on the instruction. Solving all the instruction constraints is reduced to selecting a value from each given set in such a way that the instruction semantic is respected. This should be done with randomly uniform probability, where each solution should have the same probability of being selected.

FPgen's main challenge is to provide solving engines that can solve these constraints, even the more intricate ones, within a reasonable amount of time. Since it is clear that many of the full constraints yield NP-Complete problems, it cannot be expected that *all* such problems will be solved in polynomial time. However, FPgen's heuristics will attempt to ensure that only a small fraction of the problems require long time periods to be resolved. For more on the algorithms and different approaches used, see Section 4 - Solving the Constraints.

The rest of the functionality is a relatively straightforward extension to this basic power. Hence, we first present data constraints for *individual* instruction operand, and then introduce the extensions that complement the full description of the tool's scope.

The general outlook of a single instruction constraint is of the form:

FPinst (Op1 in Set1) (Op2 in Set2) (IntRes in Set3) (Output in Set4)

where FPinst is a floating-point instruction with two input operands (Op1 and Op2), one intermediate result (IntRes), and one output. Input and output operands are defined by the architecture, while the intermediate result is a construct that belongs to the implementation domain (see Section 5 Additional Properties). The case of two operands and a single intermediate result is used here for simplicity of notation, but generalization to any number of such parameters is straightforward.

FPgen provides multiple means to define sets of floating-point numbers and sets of floating point fields (sign, exponent, and fraction). The different types of sets, shown in Table 2, serve to conveniently translate constraints that emanate from typical tasks of the verification plan. They therefore constitute a critical component of the tool.

Set Name	**Definition**	**Example**	**Motivation**
Range	A range of numbers	[min-denorm, max-denorm]	-Definition of all the FP basic types (such as denorm) - Target neighborhood of critical FP values
Mask	A number where some bits are don't cares (X), while the others are regular 0's and 1's	+-Zero = X000...000 -Zero = 1000...000	- Target Hamilton neighborhood of critical FP numbers - Check correct rounding: only some LSBs of the intermediate results are relevant, while the others can be random (i.e., don't cares)
Number of ones/zeros	Specify the number of bits equal to 1 or 0. Min and max are given	At least 1 bit set in bits 5-8	Numbers that exhibit extraordinary sequences of 1's and 0's are often handled in a specific way (to gain performance) by the micro-architecture
Sequence of ones/zeros	Specify the length of a continuous stream of 1's or 0's. Min and max are given	A stream of at least 45 1's in the mantissa	Similar to previous entry, but focuses on continuous sequences. For example, a number with a very long sequence of 1's [16]
Symbol-reference	Usage of symbol to relate between operands	Exp(op1) - Exp(op2) <= 2	For addition instructions: relate between exponents of input operands (when exponents are too far apart—which accounts for the vast majority of cases—the addition reduces to a trivial case). It is also important to relate between input and output
Set operations	Union, intersection, and complement of basic set types	op1 is norm and Exp(op1) is Exp(op2)+5	Further pinpoints the targeted areas, thereby providing the language with unlimited power

Table 2: Supported Set Constraints

3.1. Coverage Model

A coverage model is a set of related verification tasks, such as the All Types model from Table 1. For the sake of comprehensiveness and efficiency, test plans usually require that full coverage models be fulfilled, rather than fulfilling a list of dispersed tasks. These models attempt to partition the test verification space in such a way that each partition receives a more or less equal focus. FPgen allows the user to directly request the fulfillment of coverage models.

A coverage model is defined by specifying a set of different constraints to be fulfilled, where each constraint corresponds to a particular task targeted by the coverage model. More precisely, a single instruction coverage model will have the following form (again, for an instruction with two inputs):

FPinst (Op1 in Pattern1) (Op2 in Pattern2) (IntRes in Pattern3) (Res in Pattern4)

A Pattern is a construct that represents a logical OR among sets of floating-point numbers. Patterns have the following general form:

Pattern = Set1 OR Set2 OR...OR SetN...

where each Set is a set of floating-point numbers, as defined above. Each task of the coverage model corresponds to a specific selection of a Set for each Pattern. The number of different tasks in such a model is the multiplication of the number of Sets for each participating Pattern (Cartesian product).

3.2. Sequences

Although constraints are defined for single instructions, the input language allows a sequence of instructions to be specified as an individual coverage task. In general, this is a straightforward extension to the single instruction scope. However, for relatively long sequences, specific care should be taken that constraints are fulfilled, since all resources (i.e., FP registers) tend to be used [25].

Coverage models can also be defined for a sequence of instructions. Covering such models is reduced to covering the Cartesian product of the tasks engendered by each single instruction coverage model.

The importance of sequences stems from the complexity of the underlying, typically pipelined, micro-architecture. FPgen allows to combine interesting data-related events with error-prone micro-architecture scenarios, such as back-to-back exception situations. The area of floating-point sequence verification is not within the scope of FV, or at most weakly tackled, rendering the corresponding FPgen functionality critical even to FV-based verification processes.

4. Solving the Constraints

This section provides an overview of the major FPgen module which is responsible for solving the constraints. While a detailed description of the underlying algorithms is beyond the scope of this paper (see [15]), we present the overall solving scheme and give some insight into the inherent complexity of the individual algorithms.

It should be clear that resolving constraints over the floating point numbers involves a different type of mathematics than the standard one over the Real numbers [27]. Table 3 below illustrates this observation with a simple example, assuming floating point numbers with a 4-bit fraction.

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

Table 3: A Constraint with no Solution over the FP Numbers

Over the floating point domain, the input ranges include only two numbers, yielding three possible outputs of the subtract operation: 0, $1.0000*2^{6}$, and $-1.0000*2^{6}$. There are clearly solutions over the Reals (where the ranges shown include an infinity of numbers), but no solutions over the floating points.

FPgen solves constraints that emanate from set restrictions on instruction operands. Given a restriction, FPgen is tuned to search for a random instance that will solve it and is uniformly distributed among the set of all solutions. For some complex cases, the uniformity target is dropped and, at the very least, FPgen ensures that each solution has a reasonable probability of being selected. (See Section 5.1 - Instance Selection.)

As described above, constraints can be given on input operands, output operands, or on both types simultaneously. Clearly, solving constraints on output operands, as opposed to input operands, leads to a significant leap in the complexity of the problem, as it involves the semantic of the instructions. Constraint restrictions can become intractable when simultaneous constraints are requested on both input and output operands. For example, it is unclear how a random solution can be found for the task appearing in Table 4. It was taken from a coverage-model defined in a test-plan of an IBM microprocessor: The instruction is *fp-multiply-add* (output = round(op1*op2+op3)) and the format is double-precision.

Op1.fraction	Op2.fraction	Op3	Exponent relations
Has between 1 to 10 ones	Has between 1 to 10 zeros	denorm	1.Exp(Product)-Exp(Op3)=0 2.Exp(Op3)-Exp(output)=52

Table 4: An FPgen Example

Many similar cases can be shown to be NP-Complete.

Multiple engines were used to implement the solving phase (Figure 1). After first analyzing the constraint type, FPgen directs the task to the appropriate engine or group of engines. Two major types of engines are used:

1. A-engine – engines that are guaranteed to find a solution of the constraint within a reasonable time, if it exists. Therefore, their output is either a random solving instance, or a message that no solution exists. These analytic engines are based on mathematical algorithms for tractable, albeit often complex, problems.

2. S-engine – engines used on constraints for which no A-engine is known. S-engines are dedicated to 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, their output can also include a “quit” message, indicating that no solution has been found, although one may exist.

Depending on the instruction and the type of constraints, FPgen will opt for either A-engines or S-engines. Table 5 describes the existing engines (the development of additional engines is underway). When the A-engine path is selected, a specific A-engine is chosen to either resolve the problem or inform that no solution exists. The scheme differs slightly when the S-engine path is taken. First, a time limit must be fixed. Second, the problem will not be transferred to a unique engine, but 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. Currently, each S-engine works independently, but in the future, we envision a "cooperative" mode where information is transferred between S-engines to speed up the process, increasing the odds of success.

4.1. A-engines and S-engines: a High-Level View

In this section, we provide some insight into the individual solving engines and describe the engine partition over the existing problems. Table 5 summarizes the situation. In short, A-engines handle constraints on input or output only, and a few special cases of full constraints. All other types of full constraints are directed towards S-engines. Some engines are *generic* in the sense that they can be applied to any problem, while others are valid only for a *specific* instruction, or even for only a specific type of constraint on a given instruction.

For example, input constraints are resolved by a single A-engine, while a different A-engine is used for each output-only constraint. The S-engines are of a more generic nature, but individual heuristics can be added for each instruction separately. For instance, a hill-climbing search for full constraints (see description in Table 5) on an add instruction should know that there are two major types of addition; one involves numbers of comparable dimension and the other does not. Thus, in the latter case, the output is equal to the largest term, except perhaps for the LSBs due to rounding.

Most of the A-engines are based on mathematical algorithms. For example, the engine for constraints that works only on the intermediate-result of the fp-multiply instruction is based on factorization. The algorithm for mask constraints on all operands for the fp-add instruction is based on mathematical features of the add instruction and the mask set [15].

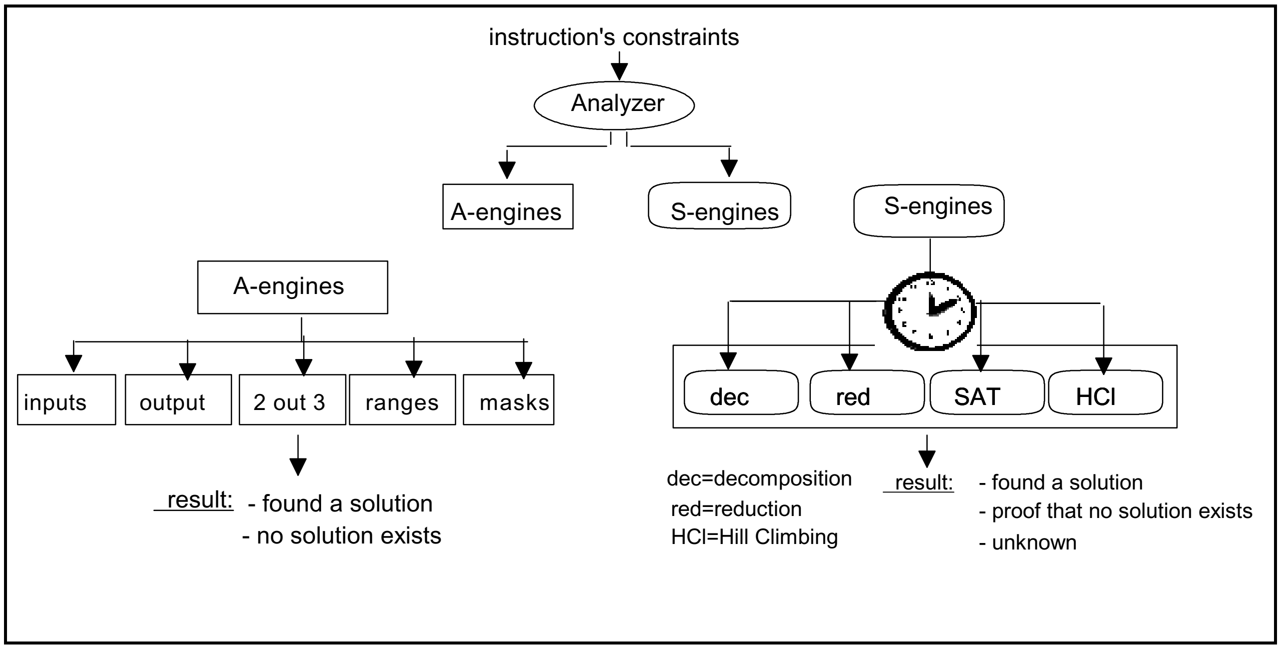

Figure 1: FPgen Engines Scheme

Constraint Type	Engine Type	Type of instruction	Remarks
Inputs only	"A" Generic	All	Uniformity of the solution is usually achieved, except for complex constraints involving set operations
Output only	"A" Specific	All	In many cases, a separate engine is provided for output constraint and for intermediate result constraint
2 out of 3 Input, Output	"A" Specific	All (2 inputs)	Engines for solving constraints that include an output constraint and a single input constraint on instructions with two inputs. The remaining input is free
Full ranges	"A" Specific	All	An important family of constraints. For example, the All Types model presented above belongs to this family. Much less straightforward to solve than anticipated. Algorithms will be published in a separate paper.
Full Masks	"A" Specific	add/sub type only	Algorithm to be published [15].
Decompositi on	"S" Generic	All	The full constraint is decomposed into two constraints: inputs-only and output-only (or output and single input if relevant). Each one is solved separately with a

			corresponding A-engine, until the solution matches the additional constraint or the time limit is exhausted.
Reduction	"S" Generic	All	The full constraint is reduced to one being solved by an A-engine, (e.g., full-ranges). Since the reduction may remove or add solutions, this is done repeatedly until a valid solution is found or the time limit is exhausted.
SAT	"S" Generic [a]	All	The problem is translated to a CNF formula where each of the bits of the FP numbers represents one Boolean variable. A solution to the CNF formula is found through a SAT solver (e.g., GRASP [22,23] , modified in order to get a random solution).
Hill Climbing	"S" Generic [a]		This engine uses a space-search similar to Hill-Climbing or simultaneous annealing [26]. A random FP number is chosen for each of the operands. During each iteration, one bit is flipped. The flipped bit is chosen according to an instruction-dependent heuristic function.

In the above table, A stands for A-Engine, S for S-Engine, and [a] to indicate that the engine is of generic nature, but individual heuristics/translations are added for each instruction separately.

Table 5: FPgen Engines

5. Additional Properties

This section introduces additional FPgen features.

5.1. Instance Selection

When FPgen selects an element in a Set, the idea is to give each element the same probability (i.e., select with a uniformly random distribution). We opt for natural distribution since there is no reason to prefer one solution over another. However, it is recognized that in some cases, the price to reach this goal may be high in terms of time complexity. Therefore, this requirement is often relaxed; it is sufficient to have a relatively equal probability for each element. The minimal requirement is that each element has at least a *reasonable* probability of occurring. This is usually simple for individual constraints on input operands, (although it can become deceitfully complex for intricate sets of values), becomes more difficult when the data constraint is on the output operand, and yields largely untractable problems for most full constraints.

5.2. Intermediate Result Framework

The IEEE 754 standard requires that each operation be 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 floating-point number with a wider exponent and a wider fraction, which enables the calculation of outputs as though the intermediate format was unbounded.

FPgen relates to such an intermediate result as one of the instruction operands. The field's sizes (exponent and mantissa) are parameters that can be controlled by the user. This way, all the defined constraints can be applied to the intermediate result as well. Consider for example, a constraint on the intermediate result, where the constraint asks for an inexact result or a result with the maximum intermediate exponent. Analytic algorithms have been described to solve this type of constraint for some specific patterns and instructions [17].

Standing apart from other instructions, the fp-multiply-add instruction has two intermediate results. One intermediate result represents the result of (op1*op2) and the other (op1*op2+op3).

5.3. Knowledge Library

FPgen provides a means to accumulate and encapsulate knowledge previously acquired.

1. **Data Type**. Data types enable to weight different constraints on a single operand. For example, one can ask for an operand to be 30% denormal and 70% zero. It is also a convenient way to define a regular constraint (like denormal), for reuse by name (as a macro).

2. **Event**. An Event gathers *several* individual instruction constraints and allows them to be *weighted* separately. This yields a means to describe the different ways a certain Event may happen, by referring to all of them using a single name, and weighting them appropriately. For example, the Event Add_Overflow could be defined as appearing in Table 6.

Weight	Op1's Set	Op2's Set	Intermediate Set
50%	huge	huge	exp>max-exp
30%	max norm	norm	exp>max-exp
20%	max norm	denorm	exp>max-exp

Table 6: Event Add_Overflow

In the above table, exp is exponent, huge is a set of high floating-point numbers (range), and max-exp is the maximal exponent value. Given such definitions, one can request (assuming ADD_Underflow is defined in an analogous manner):

Add_Overflow OR Add_Underflow

Of course, to get an overflow event, the following request is sufficient:

Instruction	Set for intermediate-result
fp-add	Exponent > max-exp

Table 7: Overflow Constraint

However, every solving instance will be then given the same probability, while the Add_Overflow Event can induce a more adequate partitioning distribution, verification-wise.

The capability of enumeration for each case of the Event is also given. When enumerated, an Event defines several constraints that must be fulfilled for the coverage model. In this case, the weights are redundant.

3. **Library**. FPgen provides a repository for preserving coverage models, data-types, and events. This way, the accumulated knowledge is captured for future use and reference.

5.4. Coverage Model Restrictions

For some coverage models, as in the All IEEE Types case, there are individual tasks that are not feasible (i.e., there is no possible solution). Although FPgen is able to discover these cases on its own and will issue appropriate messages, it is good practice performance-wise, to inform FPgen about these restrictions. FPgen makes it possible to include this optional restriction information when defining a model.

5.5. Exhaustive Enumeration

An additional way of defining a coverage model involves requesting *all* the elements of a given set. For example, one can ask to have the fp-add instruction with all the possible different values of exponent on the output. Obviously, the size of the set should be manageable.

5.6. Time Complexity

Most of the constraints are solved in a relatively short amount of time. Some extremely complex constraints, where the number of solutions is small, may require a very time-consuming solving process. FPgen users may control the maximal time allowed for searching for solutions. For example, a few tasks may remain uncovered for a desired coverage model, and it is not even clear whether they are feasible. Similarly, to optimize its design, a designer may want to know whether certain combinations of data are possible on inputs and output. Clearly, these types of usage should allow longer time periods for FPgen's solving mechanism. Upon failing to find a solution, FPgen indicates whether there are no solutions or the allocated time period was insufficient to find a potential solution.

5.7. Integer Constraints

The IEEE standard demands support for conversion between floating-point and integer types. FPgen supplies the set constraints Range and Mask for integer biasing.

5.8. Coverage Density

Within a coverage model, the number of instances required for each task can be controlled using the *coverage density* parameter. This enables FPgen to generate many different instances that fulfill the same underlying purpose. This is important since a task usually reflects a suspected area, and hitting this area on multiple points yields a significant added-value.

6. Coverage by Generation

FPgen implements the concept of *coverage by generation*. This means the input of FPgen is a coverage model request, and the generation process is focused on fulfilling each of its tasks, one at a time. This is different from *coverage by feedback* [*19*], where generation is partially random and coverage analysis of the generated tests is fed back into the generation process to bias it towards uncovered tasks. The coverage by generation approach has several advantages:

- No expensive expert time is spent on coverage analysis and on re-tuning the generation
- Tasks are covered in a homogeneous manner (i.e., each task is hit the same number of times)
- Faster generation of the covering set of tests
- Impossible tasks can be reported by the tool. No expert time is spent analyzing such tasks

Figure 2 illustrates the typical coverage progress of the two approaches.

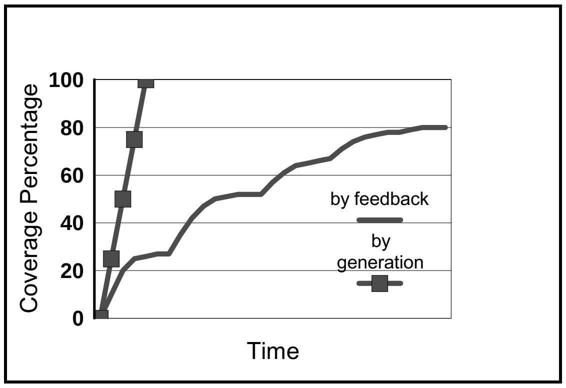

Figure 2: Coverage by Generation vs. Coverage by Feedback

The straight line of the coverage by generation approach is expected, as each task is directly targeted and covered. The wave-like form of the coverage by feedback approach stems from its inherent lack of precise knowledge and its iterative feedback. More specifically, each wave represents the covered targets until additional feedback is provided to the generation process, time at which an additional wave, typically of smaller amplitude, starts. The flattening of each wave reflects the need for additional knowledge. The coverage percentages of the successive waves vary with the complexity of the tasks present in the coverage model. Those given here are only for purpose of illustration.

7. Conclusion

We developed FPgen, a Deep-Knowledge Test-Generator for floating-point datapath verification, whose purpose is to give a quasi-optimal test generation framework for the simulation domain. FPgen is already being used to verify several IBM processors, and has already assisted in uncovering many interesting bugs. The analysis of these bugs serves as a basis for further development in FPgen. In addition, FPgen easily covers tasks that were previously very hard to hit. Moreover, for each of the covered tasks, a practically unbounded number of cases can be generated.

While FPgen is a generic tool that supports IEEE architectures, adaptation for different architectures is possible. For example, FPgen already supports multiply-add (a non-IEEE instruction).

In parallel with FPgen development, we are setting-up a comprehensive test plan for the IEEE standard. FPgen's library will include coverage models to fulfill this test-plan, both qualitatively and quantitatively. Since most architectures comply with the IEEE standard, this will allow most floating point design verification efforts to focus almost exclusively on the microarchitecture part of the verification, an area where FPgen functionality is also highly applicable.

Bibliography

1. IEEE standard for binary floating point arithmetic. An American National Standard, ANSI/IEEEE Std 754-1985.
2. B. Beizer: “Software Testing Techniques”. Van Nostrand Reinhold, 1990.
3. Y.Lichtenstein, Y.Malka and A. Aharon, “Model-Based Test Generation for Processor Design Verification”. Innovative Applications of Artificial Intelligence (IAAI), AAAI Pres, 1994.
4. L. Fournier, D. Lewin, M. Levinger, E. Roytman and Gil Shurek: “Constraint Satisfaction for Test Program Generation”. Int. Phoenix Conference on Computers and Communications, March 1995.
5. A. Aharon et al. “Test Program Generation for Functional Verification of PowerPC Processors in IBM”. DAC95, San Francisco, June 1995, pp 279-285.
6. B. Marick: “The Craft of Software Testing, Subsystem Testing Including Object-Based and Object-Oriented Testing”. Prentice-Hall, 1995.
7. C. Kaner, “Software negligence and testing coverage”. In proceedings of STAR 96: the Fifth International Conference, Software Testing, Analysis and Review, pp 299-327, June 1996.
8. R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv: “User defined coverage - a tool supported methodology for design verification”. DAC98, pp 158-163, June 1998.
9. Edmund M.Clarke, Steven M.German and Xudong Zhao: “Verifying the SRT Division Algorithm Using Theorem Proving Techniques”. Formal Methods in System Design, volume 14, number 1, January 1999, pp 7-77.
10. David M.Russinoff: “A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode”. Formal Methods in System Design, volume 14, number 1, January 1999, pp 75-125.
11. J. o’Leary, X.Zhao, R.Gerth and C.-J.H Seger: “Formally verifying IEEE compliance of floating-point hardware”. Intel Tech. Journal, Vol 1999-Q1, pp1-14. http://developer.intel.com/technology/itj/q11999/articles/art_5.htm
12. L. Fournier, Y. Arbetman, M. Levinger: “Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator. Application to the x86 Microprocessors Family.” DATE99, Munchen, 1999.
13. Mark D. Aagaard et al.: “Formal Verification of Iterative Algorithms in Microprocessors”. DAC 2000.
14. Silvia M. Mueller and Wolfgang J.Paul: “Computer Architecture, Complexity and Correctness”, Springer-Verlag Berlin Heidelberg 2000.
15. L. Fournier, A. Ziv: “Solving the Generalized Mask Constraint for Test Generation of Binary Floating Point Add Operation” . To appear in Theoretical Computer Science, Special Issue: Real Numbers and Computers.
16. Coe, T. “Inside the Pentium Fdiv bug”. Dr. Dobbs Journal (April 1996), pp.129-135.
17. M.Parks, Number-theoretic Test Generation for Directed Rounding, Comput. Arithmetic, 1999.
18. K.L. Mcmillan. “Fitting Formal Methods into the Design Cycle”.
19. G. Nativ, S. Mittermaier, S. Ur, A. Ziv. “Cost Evaluation of Coverage Directed Test Generation for the IBM Mainframe”. International Test Conference 2001.
20. I. Beer et al. “RuleBase: Model Checking at IBM”. CAV97.
21. K. McMillan. “Symbolic Model Checking”. Kluwer Academic Publishers, 1993.
22. J.P.M Silva and K.A. Sakallah. “GRASP - a new search algorithm for satisfiability”. Technical Report TR-CSE-292996, University of Michigan, 1996.
23. J.P.M Silva and K.A. Sakallah. “GRASP: A search algorithm for prepositional satisfiability”. IEEE Transactions on Computers, 48:506-516, 1999.
24. W. Kahan. “A Test for Correctly Rounded SQRT”, 1996, //www.cs.berkeley.edu/~wkahan/SQRTest.ps
25. A. Adir, E. Marcus, M. Rimon, A. Voskoboynik. “Improving Test Quality Through Resource Reallocation”. HLDVT 2001.
26. S. Russel, P. Norving. “Artificial Intelligence: A Modern Approach”. pp 111-115.
27. Claude Michel, Michel Rueher, Yahia Lebbah. “Solving Constraint Over Floating-Point Numbers”. CP2001