Leveraging Accelerated Simulation for Floating-Point Regression

John Paul[1], Elena Guralnik[2], Anatoly Koyfman[2], Amir Nahir[2], and Subrat K Panda[1]

[1] IBM Systems & Technology Group in Bangalore, India, {john.paul,subratpanda}@in.ibm.com
[2] IBM Research in Haifa, Israel, {elenag,anatoly,nahir}@il.ibm.com

Abstract. Accelerated simulation (acceleration) platforms play a pivotal role in the verification of today's complex designs. Currently, acceleration is used with either adapted pre-silicon tools or post-silicon tools. We present a novel acceleration-only tool, which enables a fast and efficient methodology for floating-point regression. We overcome the lack of test-bench in this environment through self-checking.

1 Introduction

Functional verification is widely acknowledged as one of the main challenges of the hardware design cycle [12]. The growing size and complexity of modern hardware systems have turned the functional verification of these systems into a mammoth task [20]. Verifying such systems involves tens or hundreds of person years and requires the compute power of thousands of workstations. But even with all this effort, it is virtually impossible to eliminate all bugs in the design before it tapes-out. Despite advances in formal verification technologies [7], dynamic verification (a.k.a. simulation-based verification) remains the primary vehicle for the functional verification of hardware systems [20]. Today's state of the art verification methodologies include a highly automated process that incorporates stimuli generation, checking, and coverage collection—combined with islands of manual labor [20].

In the past, software simulation was (almost) the exclusive vehicle for executing the verified designs. But, the increasing complexity of designs, combined with shorter time-to-market requirements, raised the need for performing parts of the verification tasks on other platforms. Today, functional verification is performed on a variety of platforms, ranging from transaction-level modeling, via software simulation, acceleration, and emulation, to the silicon itself [18, 4]. In some cases, verification is done in a heterogeneous environment involving a variety of platforms, as in the case of Hardware-Software Co-Simulation [6].

Acceleration and emulation platforms are somewhere in between software simulation and silicon. They are much faster than software simulation, but not as fast as silicon. Similarly, they provide better observability than silicon, but not the free and total observability provided by software simulators. Therefore, verification solutions, and specifically stimuli generators, for such platforms should combine requirements from both worlds.

The acceleration platform is especially attractive because it can be leveraged very early in the process. It can serve to strengthen the pre-silicon verification effort and enable the early detection of bugs. In addition, acceleration products offer a simulation-like interface and are thus relatively easy to use. Some products go as far as offering live, seamless, migration between acceleration and simulation [1].

The high cost of developing unique verification solutions for acceleration platforms causes most of these solutions to be adaptations of existing solutions for software simulation or post-silicon tools. In this paper, we demonstrate how the acceleration platform can be utilized to address one of the more common use-cases in the development life cycle: regressing a change in the design logic.

As part of the logic development process, designers make frequent changes to the logic. These changes may originate from the need to fix a bug, improve timing, or simply implement a change in the specifications. One common concern is that making a change to the logic, as small as it may be, can introduce new bugs. It would greatly improve productivity if such bugs were detected shortly after their introduction. To validate that no new bugs were introduced in the process, the verification engineer, or the logic designer, runs a *regression suite* [8]. The regression suite is a large set of test-cases that provide high confidence regarding the functional correctness of the design. For complex units, running regression in software simulation can take days. [3]

Using our solution, the verification engineer can choose a large set of ready-made test-cases, such as the regression suite mentioned above, and convert them into a single, fully-contained, self-checking, program. Using an accelerator can speed up the execution time for the test case by several orders of magnitude. This not only leads to finding bugs faster but also has a significant effect on the time required to reach coverage closure.

We demonstrate the proposed solution on floating point (FP) data verification. Verifying the hardware implementation of the floating point unit (FPU) is known as an intricate problem. The numerous corner cases of the vast test space, coupled with the complexity of the implementation of floating point operations, turn the FPU verification effort into a unique challenge in the field of processor verification. It is not surprising that the most well known hardware bug is Intel's FDIV bug [2].

We present a tool that takes a large set of FP test-cases (pre-generated by FPgen [5, 11]) and converts them into a single program that is then simulated at the core-level environment. This program is a concatenation of the original test-cases, where each test-case is preceded by a prolog and followed by an epilogue. The prolog mimics the required initializations specified in the original test-case. These initialization are typically handled in software simulation by the environment, which forces the initial values into the specified resources. We convert these initializations into a set of reloading instructions, which bring the required resources to the desired state. The epilog runs in two different modes: *simulator mode* and *hardware mode*. When running in simulator mode, we run the program to collect the expected results. Following that, we run the test-case on the accelerator in hardware mode. In this mode, the epilog is in charge of

[3] while an industrial simulation farm holds thousands of servers, the regression task is so common that it is impractical to expect to be assigned with sufficient machines to complete the regression task quickly

comparing the actual state of the design with the expected values obtained from the reference model, and flagging any discrepancies that may indicate a bug.

We show that using this tool, thousands of test-cases can be compressed into a single test-case and executed on an accelerator in a short amount of time, and report results of a field trial.

The rest of this paper is organized as follows. In Section 2 we provide background about floating point verification and test-generation, as well as on the accelerated simulation platform. Section 3 provides an in-depth review of our solution for floating point regression on the acceleration platform. Our results are described in Section 4. Section 5 concludes this paper.

2 Background

2.1 Acceleration

Accelerated simulation platforms (more commonly known as accelerators) are an important component in today's simulation-based verification [9]. Accelerators are special purpose massively-parallel machines, developed for the sole purpose of accelerating the simulation of hardware models. The accelerator is constructed of a large number of tightly synchronized parallel logic processors. To simulate a hardware model on an accelerator, the model must first be compiled in a process that converts the hardware model to a set of instructions for each of the accelerator's processors, schedules the instructions for the processors, and determines the synchronization points between them [16]. State of the art accelerators run over three orders of magnitude faster than software simulation (i.e., over 1000 times faster).

While accelerators offer much faster simulation, there are several challenges related to their use. First, any interaction between the accelerator and an external computer (termed host) requires stopping the acceleration engine. This means that using a traditional environment in which the test-bench runs on the host and the accelerator runs the hardware model severely under utilizes the accelerator due to the frequent communication. The transaction-based acceleration (TBA) [14] methodology overcomes this problem by having part of the test-bench compiled into the hardware model and reducing the interaction between the host and the accelerator.

However, the TBA approach encounters the second challenge in using accelerators. The speed of the accelerator, as well as the duration of the compilation process, heavily depend on the size of the hardware model. That is, the more logic is added to the hardware model, the slower the accelerator runs. This limits the ability to use techniques such as TBA or checker synthesis [10].

We note that for the case of floating point data path verification presented in this paper, neither TBA nor checker synthesis is applicable. Moving input data operands from the host to the accelerator and output values in the other direction is unreasonable due to the extensive amount of data. The complexity of the floating point algorithms and the required accuracy needed to verify the results prohibit the creation of hardware realizable checkers.

2.2 Floating Point Test Generation

The complexity of the floating point data path implementation and the numerous corner cases that should be addressed not only call for a dedicated test generator, but also demand a comprehensive test plan. The FPgen [5] verification solution provides these two components.

FPgen generation capabilities are primarily based on constraint satisfaction technology [17]. As part of the verification process, FPgen produces a large set of test-cases in the form of input data operands for floating point instructions, targeting the areas outlined by the test plan.

The primary focus of the FPgen generator is to solve data constraints on operands of individual floating point instructions. A data constraint on an operand is defined as the set of values that can be selected for this operand. An individual instruction may have independent data constraints for each one of its operands. Solving all the instruction constraints is equivalent to selecting a value from each given set, such that the instruction semantics are satisfied. FPgen provides engines that solve these constraints within a reasonable amount of time. Moreover, when multiple solutions exist for a constraint, one should be selected at random, with uniform probability where possible. This randomness is important because the constraints only reflect a suspected area. One instance in this area might reveal a problem, while another might not.

Constructing an appropriate set of such constraints is of utmost importance to ensure successful verification. Exhaustive checking implies testing an enormous, practically unbounded, number of different calculation cases; practical computational resources suffice only to simulate a meager fraction of these. We need to choose these cases very carefully in order to obtain a representative sample of the state space. In particular, a proper focus on the corner cases is a crucial factor in providing a sufficiently comprehensive set of test-cases. Continued analysis of the instructions themselves, and of the various bugs appearing in their implementations, has provided us with valuable knowledge, reflected as an integral part of FPgen's test plan template.

3 FP Regression Tool

We apply the concept of converting a set of pre-generated test-cases into a self-executing self-checking program for the floating-point data path regression problem. In this section we describe the tool's execution flow, the structure of the generated program, and the accompanying debug aids.

3.1 Execution Flow

The high level execution flow of the tool is described in Figure 1.

We start with a set of test-cases pre-generated by FPgen[5], as depicted in the leftmost section of the Figure 1. FPgen is a test-generation framework that provides a convenient platform for biasing and generating operand data for floating-point instructions. The verification engineer can choose the set of test-cases so they focus on a specific instruction or event (for example, *sqrt*) or opt for a set that provides broad coverage of

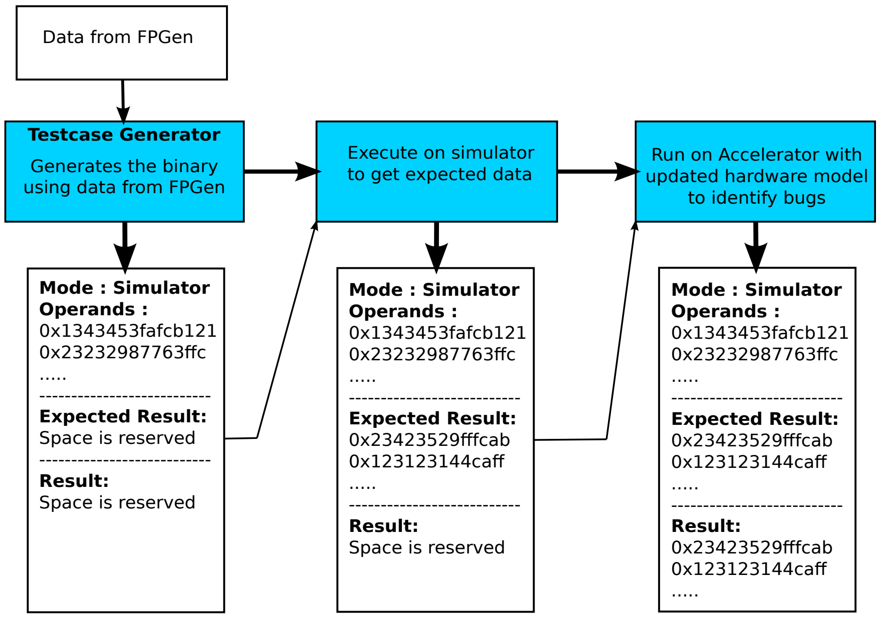


Fig. 1. Execution Flow

the entire floating point spectrum. Our tool outputs a single program that includes all the desired test-cases.

Next, our tool generates the initial program and then executes it on a software reference model that is instruction accurate [4], as depicted in the middle section of the Figure 1.

The purpose of this stage is to collect and store the expected results from the execution of the different test-cases. We assume the software reference model correctly implements the specification and provides accurate results [5]. After every test-case, epilog instructions are executed to store the results into empty arrays. This is done using instructions that are part of the generated program and does not require any involvement of the environment.

Once the program completes execution on the software reference model, our tool dumps the data from the arrays and modifies the program image. The program is modified in two ways: the expected results of the test-case execution are now stored in an array designated for storing expected results. In addition, we modify the execution

[4] An instruction accurate reference model calculates the values that will appear in the registers and memory after each executed instruction, as specified in the architecture book.

[5] In reality, this may not be the case, and errors in the software reference model are often found. This makes the debugging of the failure a little more challenging, but does not significantly change anything that interferes with our solution.

mode to *hardware* mode so the subsequent execution of the program can also check the results.

Finally, we take the modified program and run it on an accelerator with an updated (potentially buggy) design model, as depicted in the rightmost section of the Figure 1. During this run, the program executes the test-cases and compares their results to the expected values. Any data related to results that are incorrect is saved to a debug report at the end of the run.

3.2 Program Structure

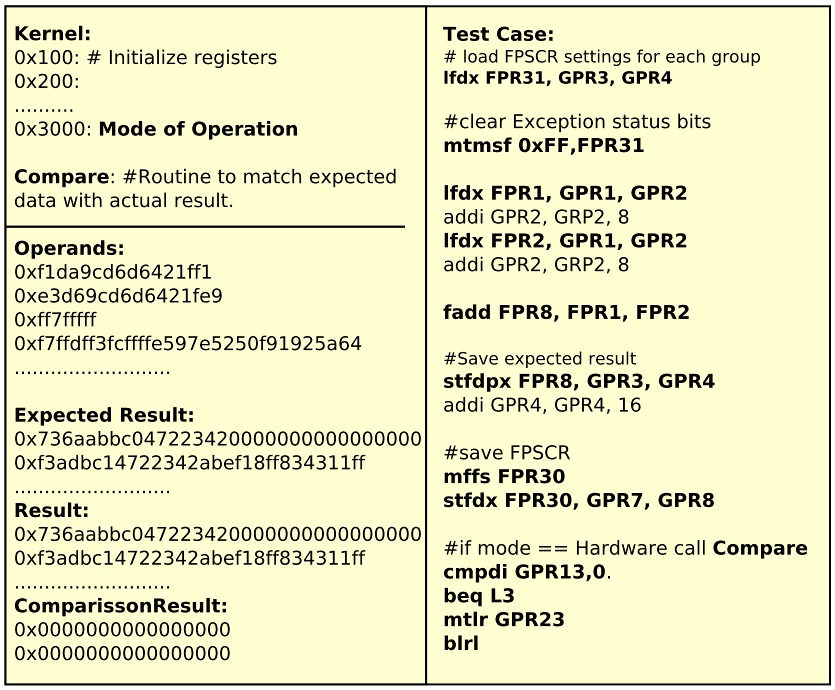
The generated program is constructed of three major parts: kernel, data tables, and the test program itself. Figure 2 depicts this structure.

Kernel The program is designed to run on *bare-metal* [19], that is, we do not rely on an operating system (OS). This is important for two reasons. First, it significantly reduces all OS-related overheads and thus enables us to maximize the utilization of the accelerator (i.e., we spend minimum cycles executing "irrelevant" instructions). In addition, it enables us to have complete control of the system and switch freely between running modes (e.g., from hypervisor to user mode).

Table 1. Program usage of registers

Register	**Usage**
GPR1, GPR2	base and offset pointers to input data table
GPR3, GPR4	base and offset pointers to input state table
GPR5, GPR6	base and offset pointers to actual/expected results data table
GPR7, GPR8	base and offset pointers to actual/expected state table
GPR9, GPR10	base and offset pointers to expected results data table
GPR11, GPR12	base and offset pointers to expected state table
GPR13	mode of operation (simulation/hardware)
GPR14, GPR15, GPR16	used for comparisons
GPR17, GPR18	base and offset pointers to comparison results table

The kernel is in charge of initializing the relevant resources once the execution of the program begins. We demonstrate the concept on a real Power$^{\text{TM}}$ design, and thus, we make extensive use of General Purpose Registers (GPRs). We assign most of the processor's general purpose registers with fixed roles for managing the test program.



Kernel:

```
0x100: # Initialize registers
0x200:
..........
0x3000: Mode of Operation
```

Compare: #Routine to match expected data with actual result.

Operands:

```
0xf1da9cd6d6421ff1
0xe3d69cd6d6421fe9
0xff7fffff
0xf7ffdff3fcffffe597e5250f91925a64
.........................
```

Expected Result:

```
0x736aabbc047223420000000000000000
0xf3adbc14722342abef18ff834311ff
.........................
```

Result:

```
0x736aabbc047223420000000000000000
0xf3adbc14722342abef18ff834311ff
.........................
```

ComparissonResult:

```
0x0000000000000000
0x0000000000000000
```

Test Case:

```
# load FPSCR settings for each group
lfdx FPR31, GPR3, GPR4

#clear Exception status bits
mtmsf 0xFF,FPR31

lfdx FPR1, GPR1, GPR2
addi GPR2, GRP2, 8
lfdx FPR2, GPR1, GPR2
addi GPR2, GRP2, 8

fadd FPR8, FPR1, FPR2

#Save expected result
stfdpx FPR8, GPR3, GPR4
addi GPR4, GPR4, 16

#save FPSCR
mffs FPR30
stfdx FPR30, GPR7, GPR8

#if mode == Hardware call Compare
cmpdi GPR13,0.
beq L3
mtlr GPR23
blrl
```

Fig. 2. Program structure

Table 1 lists these roles. For example, we use GPR9 and GPR10 as base pointer and offset register, respectively, to the expected results table. After values are saved to the expected results table, GPR10 is incremented to point to the next available entry in the table. Note that GPR5, GPR6, GPR7 and GPR8 point to different tables in the two different modes of operation. When the tool runs in simulation mode, these registers point to the *expected* values tables and when it runs in hardware mode, they point to the *actual* values.

We chose to use load/store instructions that rely on two registers, as opposed to a single register and a value-base offset. This guarantees that the program can cope with very large tables. Because our program focuses on floating point verification, there's no harm in assigning program management roles to the GPRs, which are not needed for the test program itself. Furthermore, assigning fixed roles prevents us from having to re-initialize the registers as part of the programs' execution. We only need to increment the offset registers. This further increases the accelerator's utilization.

In addition to register initializations, the kernel also includes interrupt handlers for cases in which we expect the instructions to take exception.

A symbol in the kernel is allocated to hold the value of the mode of operation (simulation or hardware). We place this symbol in a pre-determined place ($0x3000$ in Figure 2) so we can modify its value in the program image without re-compiling.

Data Tables The program includes four types of tables: input data and state tables, expected results data and state tables, actual results data and state tables, and a comparison results table. For simplicity, we chose to place data and state values in different tables.

When the program is first created, we populate its input data and state tables with the data collected from the FPgen pre-generated test-cases. The other tables are empty at this stage. Since we know the number of test-cases included in the program, we can determine the required size for each table.

When the program executes on the software reference model (in simulation mode), the kernel initializes the required registers to point to the expected results tables; the epilog instructions save the values into those tables. Once the program completes execution, we dump these values from the memory of the software reference model into the program image, populating the expected results tables there. At this stage, we also change the value of the mode of operation symbol.

When the program runs on the acceleration platform, the epilog instructions save the actual values into the actual results tables and the compare routine checks whether these values match the expected results. We do not really need to store these values, as we can do the comparison based on the test instruction's target register. However, we chose to store them into a table for later use in building a debug report.

Test Program The test-program is in fact a concatenation of the FPgen pre-generated test-cases. Each FPgen test-case, typically consisting of one or two floating point instructions, is preceded by instructions that load the data inputs into the instruction source registers and set the required state. In addition, the pointer-offset registers are incremented to point to the entries of the next test-case.

Every test-case is followed by a set of instructions that save the target register and the new state to the relevant tables. In simulation mode these are the expected results tables, while in hardware mode these are the actual results tables. In hardware mode, we also branch from this part of the test-program to the compare routine in order to validate the accuracy of the results and mark any discrepancies in the comparison table.

At first glance, it may seem like our program has a significant overhead. For every test-case we have about 20 instructions required to set the input, save the output, and compare the results. However, this is not the case. First, not all instructions "are born equal." Although the *addi* (add immediate) instruction used to increment the offset pointer requires one cycle for execution, the actual floating point instruction requires a much longer execution time.

Furthermore, the placement of the input tables in memory, along with the test program's deterministic access pattern to these tables, enables the processor to activate its prefetching mechanisms, reducing the time required to reload the source registers.

Finally, we order the test-cases within the test program according to their required input state. This reduces the rate of state changes within the test program, increasing the test program's effectiveness.

3.3 Debugging

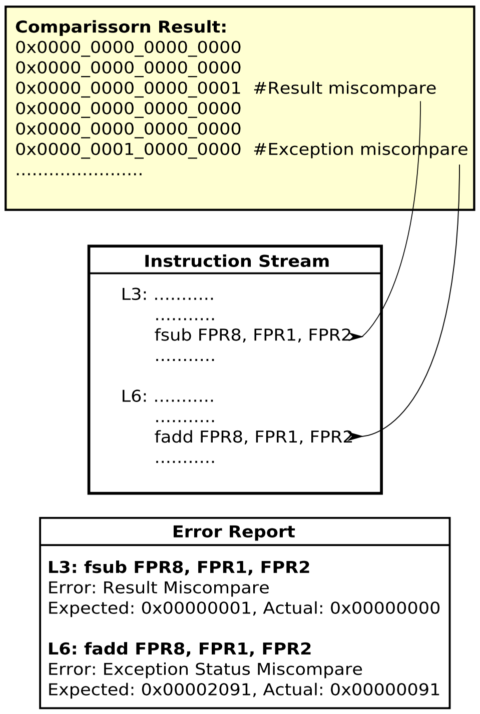
As stated above, when running in hardware mode on the accelerator, we compare the actual results with the expected ones. When the results of the test-case do not match the expected values, we store an error code into the comparison table. We use different error codes to designate different types of mismatches. Following that, the execution of the test program continues, allowing the detection of multiple errors in a single run.

After the program completes execution, we analyze the comparison table. By providing the location of the error code in the table, we are able to cross-reference it with the expected and actual results tables, in order to provide a detailed report pinpointing the failure. This process is depicted in Figure 3. The bottom of the figure displays a snippet of the report. As can be seen, the report holds the ID of the failed instruction within the test program, the type of error, as well as the expected result and actual result.

In some cases, where debugging the failure with the accelerator proves difficult, the generated report is sufficient to find the original FPgen test-case and run it in simulation, where the environment eases the debugging work (either due to the better observability, or because of the presence of better checkers).

4 Results

The Power$^{\text{TM}}$ architecture [15] supports various types of floating point data and instructions – binary floating point, decimal floating point and vector computation. In addition, the architecture also supports single/double precision values, normalized/denormalized values and various kinds of rounding modes, all making the input space huge. Furthermore, a wide variety of exceptions such as overflow, underflow, and zero-divide are supported. In the Power7 Processor core [13] the decimal floating-point (DFP) facility shares the 32 floating-point registers (FPRs) and status registers with the floating point



```
Comparissorn Result:
0x0000_0000_0000_0000
0x0000_0000_0000_0000
0x0000_0000_0000_0001  #Result miscompare
0x0000_0000_0000_0000
0x0000_0000_0000_0000
0x0000_0001_0000_0000  #Exception miscompare
......................
```

```
Instruction Stream
  L3: ...........
      ...........
      fsub FPR8, FPR1, FPR2
      ...........

  L6: ...........
      ...........
      fadd FPR8, FPR1, FPR2
      ...........
```

```
Error Report
L3: fsub FPR8, FPR1, FPR2
Error: Result Miscompare
Expected: 0x00000001, Actual: 0x00000000

L6: fadd FPR8, FPR1, FPR2
Error: Exception Status Miscompare
Expected: 0x00002091, Actual: 0x00000091
```

Fig. 3. Debug report

Table 2. Experimental setups

Experiment		**Content**			**Program Size** (KB)	
Config-uration	**Instr-uction**	**Instr. per-model**	**Models**	**Total instr.**	**Binary Size**	**Operand Array**
FADD50	fadd	50	56	2085	312	25
FADD100	fadd	100	56	3921	572	47
FADD500	fadd	500	56	16893	2400	210
FADD1000	fadd	1000	56	27663	3916	340
FADD2000	fadd	2000	56	47035	6648	581
FADD3000	fadd	3000	56	65889	9328	834
MIX10	All	10	134	1080	184	21
MIX50	All	50	134	5330	824	108
MIX100	All	100	134	10364	1520	209
MIX200	All	200	134	20648	3128	418

units; the vector unit supports data with 128 bits. FPgen can generate input operand values for all instructions that execute on these units given any constraints on the input, output, and intermediate values, as explained in Section 2.2.

To validate the value of our tool, we've conducted a wide set of experimental results. In this section we report these results, as well as results of a field trial of this tool.

4.1 Experimental Results

For the purpose of experimentation and verifying the capability and usability of the tool, we performed experiments on a variety of cases. We divide our experiments into two extreme types: *single instruction* and *instruction mix*, where the former adheres to a case where a designer makes a localized change (relevant to a single instruction – *fadd* in our case, both in single-precision and double-precision forms), and is seeking to validate this fix. The latter type is of relevance when the designer makes a broader change, and thus must validate a large set of instructions. In reality, there is a wide range of cases in between these two extremes. We further refine our experiments to consider different numbers of instructions required to validate the change.

Table 2 and Table 3 describe the results of our experiments. Each row in these tables describes one setup.

Table 2 has two sections of columns. The leftmost part, titled **Content**, describes the contents put into the program as part of the experiments. It is combined of three values: the number of desired generation solutions requested of FPgen for each model, the number of models, and the total number of the generated floating point instructions. FPgen sometimes fails to find a solution, and so the total number of instructions is always less than the product of the former two fields. A model is a set of constraints used for FPgen's activation. For example, one model may call for generating *fadd* such that the output triggers an *overflow*, while a different model may constrain both of *fadd*'s operands to be denormalized numbers.

The second column section, titled **Program Size**, provides details about the size of the generated program. We distinguish between the size of the generated program

Table 3. Experimental results

Experiment		**Run Time**			
Config-uration	**Instr-uction**	**FPGen Run (min)**	**Binary Gen (sec)**	**Ref Gen (sec)**	**Accel Run (sec)**
FADD50	fadd	12	0.4	0.35	1150
FADD100	fadd	18	0.7	0.4	1122
FADD500	fadd	50	2.8	0.8	1233
FADD1000	fadd	93	4.4	1.2	1211
FADD2000	fadd	93	7.4	1.8	1284
FADD3000	fadd	248	10.2	2.4	1352
MIX10	All	29	0.4	0.5	1112
MIX50	All	122	1.8	0.5	1115
MIX100	All	258	3.5	0.6	1176
MIX200	All	493	6.8	1	1206

(Binary Size in the table) and the size of the input operands table. The total size of the program (the sum of these two values) is important because it impacts the time required to load the program into the memory of the accelerator. Note that we have a very efficient loader that is capable of loading data into the memory of the accelerator at a rate of over $100MB$ per minute.

Table 3 holds the data regarding the time required to run each of the phases of the tool's execution. Note that we provide the time required to run FPgen to generate the test-cases, while in reality it is very common to store these test-cases, so that in future executions, this phase is redundant. The columns show, from left to right, the time required by FPgen to generate the test-cases, the time required to parse the resulting test-cases and covert them to the initial program, the time required to executed the program on the software reference model to gather expected results, and the time required to run the program on the accelerator.

Our results indicate that, as expected, the program size grows linearly with the number of floating point instructions put into the program.

Interestingly, the accelerator run time is barely affected by the number of floating point instructions in the program. This is because this time is governed by overheads - the time required to reset the accelerator, upload the hardware model, and write the program to memory. This indicates that in order to properly utilize the accelerator, the verification engineer should strive to run as big a regression as possible.

Overall, our tool enables the verification engineer to run over $65,000$ test-cases, in under half an hour.

Fault Injection and Debugging In order to demonstrate the tool's ability to discover bugs, we introduced random faults to different places in the program and observed whether they were detected by the tool and, if detected, whether the final report generated by our tool pointed to the source of the fault/error. We introduced the faults into the program after its execution on the software reference model (i.e., just before we ran it on the accelerator). We distinguish between three types of faults: input, output, and exception.

We introduced input faults by modifying the value of one of the input operands. This may represent a bug in one of the reloading instructions. In some rare cases, our tool fails to detects such faults. One example of this is the case of *division by zero*. In this case, a faulty value in the numerator may go undetected, since the data of the target FP register is not affected. We still consider this to be a problem, as this may cause intermediate events in the computation of the result to remain out of reach. Fortunately, our accelerators supported the collection of coverage data [4] and we were able to validate that the required events were indeed hit.

Output faults represent problems in computing the output values. We injected these by modifying the value of the expected results. Exception faults represent wrongful behavior, such as taking an exception when it should not have been taken or vice versa. For both of these types, our tool invariably detected all faults and was able to pin-point the problematic instruction.

Software bugs are an important issue that is a major concern in validation. Software bugs may trigger false positives and false negatives, resulting in a lot of menace in comparison to the real bugs. Our tool's framework has a certain degree of robustness against software bugs because of two important reasons: *(i)* We are able to run the program, when in hardware mode, on a software reference model to verify it and *(ii)* The program is rather generic, and the main changes between different runs are the data tables. In other words, once we verify that the reloading sequence is fully functional, it works regardless of the subsequent floating point instruction; same goes for the compare routine.

4.2 Field Trial

As part of the signoff process for one of the next IBM Power designs the need arised to run roughly 85 million test-cases in simulation. The verification team decided to, in order to meet the deadline, make use of our tool to run roughly 35 million of these test-cases on a single accelerator that was allocated to them (the same type of accelerator we used to gather the experimental results).

The test-cases were divided to 700 sets of $50,000$ test-cases. Each set was converted to a single program using our tool, and then simulated using the acceleraor. The simulation time of each set was roughly 20 minutes. Overall, three weeks work were required to simulated all 35 million test-cases (we note that the net time required for this is 10 days, but the floating-point verification team shared the accelerator with other teams as well).

The verification team estimates that running the same set of test-cases in simulation would have required two and a half months[6]. Thus we clearly see the benefits of the suggested approach in the field.

[6] Since all design units have to go through the signoff process, allocating more simulation resources to this team was not an option.

5 Conclusions and Future Work

We introduced a method that enables the verification engineer to convert a large set of pre-generated test-cases into a self-contained self-checking program. Running this program on an accelerator provides the ability to quickly verify that modifications made to the hardware logic did not introduce new bugs. We demonstrated this technique on floating point data path verification.

Our solution focuses on the data path. We intend to augment it with irritator threads [3] to increase the quality of the test-cases.

References

1. Incisive simulation acceleration deployment. http://www.cadence.com/rl/Resources/application_notes/CDN_Incisive_Simulation_Acceleration_Deployment.pdf.
2. FDIV replacement program (statistical analysis of floating point flaw). Technical report, 1994. `http://www.intel.com/support/processors/pentium/sb/CS-013007.htm`.
3. A. Adir et al. Advances in simultaneous multithreading testcase generation methods. In *Proceedings of the 6th Haifa Verification Conference*, LNCS 6504, pages 146–160. Springer-Verlag, 2010.
4. A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *Proceedings of the 6th Haifa Verification Conference*, 2010.
5. M. Aharoni, S. Asaf, L. Fournier, A. Koyfman, and R. Nagel. FPgen - a deep-knowledge test generator for floating point verification. In *Proceedings of the 8th High-Level Design Validation and Test Workshop*, pages 17–22, 2003.
6. S.-H. Chen et al. Hardware/software co-designed accelerator for vector graphics applications. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 108 –114, june 2011.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT-Press, 1999.
8. S. Copty, S. Fine, S. Ur, E. Yom-Tov, and A. Ziv. A probabilistic alternative to regression suites. *Theor. Comput. Sci.*, 404(3):219–234, 2008.
9. J. Darringer, E. Davidson, D. Hathaway, B. Koenemann, M. Lavin, J. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan. EDA in IBM: past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1476–1497, December 2000.
10. S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of system verilog assertions. In *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*, DATE '06, pages 70–75, 3001 Leuven, Belgium, 2006. European Design and Automation Association.
11. E. Guralnik, M. Aharoni, A. J. Birnbaum, and A. Koyfman. Simulation-based verification of floating-point division. *IEEE Trans. Computers*, 60(2):176–188, 2011.
12. International technology roadmap for semiconductors 2009 edition - design. Website. `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf`.
13. R. Kalla and B. Sinharoy. POWER7: IBM's next generation balanced POWER server chip. In *Hot Chips 21*, 2009.
14. S. Matalon et al. Building transaction-based acceleration regression environment using plan-driven verification approach. http://www.cdnusers.org/community/incisive/Vtp_dvcon2007_tbaregression.pdf.

15. C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture*. Morgan Kaufmann, 1994.
16. M. D. Moffitt and G. E. Günther. Scalable scheduling for hardware-accelerated functional verification. In *ICAPS*, 2011.
17. Y. Naveh et al. Constraint-based random stimuli generation for hardware verification. In *AAAI*, 2006.
18. E. Singerman et al. Transaction based pre-to-post silicon validation. In *DAC*, pages 564–568, 2011.
19. J. Storm. Random test generators for microprocessor design validation, 2006. http://www.inf.ufrgs.br/emicro.
20. B. Wile, J. C. Goss, and W. Roesner. *Comprehensive Functional Verification - The Complete Industry Cycle*. Elsevier, 2005.