Automatic Verification of Floating Point Units

Udo Krautz
krautz@de.ibm.com
IBM Corporation
Boeblingen, DE

Viresh Paruthi
vparuthi@us.ibm.com
IBM Systems and Technology Group
Austin, TX

Anand Arunagiri
aarunagi@in.ibm.com
IBM Corporation
Bangalore, IN

Sujeet Kumar
sujkumak@in.ibm.com
IBM Corporation
Bangalore, IN

Shweta Pujar
shwetpuj@in.ibm.com
IBM Corporation
Bangalore, IN

Tina Babinsky
treichar@de.ibm.com
IBM Corporation
Boeblingen, DE

ABSTRACT

Floating Point Units (FPUs) pose a singular challenge for traditional verification methods, such as coverage driven simulation, given the large and complex data paths and intricate control structures which renders those methods incomplete and error prone. Formal verification (FV) has been successfully leveraged to achieve the high level of quality desired of these critical logics. Typically, FV-based approaches to verify FPUs rely on introducing higher level abstractions to allow reasoning. This however has to be done manually, and quickly becomes tedious for optimized bit level implementations on board high performance microprocessors. Automated formal methods working directly on the bit level and providing a full end-to-end check exist but are limited to single instructions (issued in an empty pipeline), hence lack in checking control aspects related to inter-instruction interactions, or pipeline control. In this paper we present an approach based on equivalence checking to overcome the single instruction limitation for automated bit level proofs in the formal verification of FPUs. The sequential execution of instructions is modeled by two instances of the design-under-test. One of the instances acts as a reference model for the other. This allows for large numbers of internal equivalences to be leveraged by equivalence checking techniques. We show that this method is capable of proving instruction sequences for industrial FPU designs. Together with a proof of correctness of individual instructions it guarantees correctness of the FPU design as a whole. In our experience this is a one of a kind approach to perform automated end-to-end verification of FPUs.

1. INTRODUCTION

Verification is becoming the dominant cost in design flows with the increasing size and complexity of integrated circuits and systems-on-chips. With the plateauing of performance improvements from technology scaling, architectural and micro-architectural features are assuming a central role in pushing the envelope to scale to workloads of the future. These compound the complexity of blocks such as Floating Point Units (FPU) which are hard to verify to start with.

Verification of FPUs is expensive due to large and complex hardware [8] given the inherent difficult nature of the computation, sequentially deep logic, and high degree of pipelining along with associated control. Traditionally, industrial FPUs are validated by simulation, often using specialized test case generators [6] to target specific areas to exercise. While such approaches are efficient at exposing bugs, they are inherently incomplete and cannot achieve full coverage, e.g. evaluation of all operand combinations over all rounding modes and exception states. Even hardware-accelerated simulation and post-Silicon debug which offer much higher state coverage, are rendered insufficient given the sheer size of the state-space. The coverage problem is further exacerbated by shorter time-to-market (hence lesser verification time), complex logic features such as clock-/power-gating for reduced power, and richer micro architectural features requiring intricate control.

An example of such a micro architectural feature is speculative execution of instruction streams which can increase the utilization of a pipeline like an FPU. However, if the speculatively executed stream shouldn't be executed because a branch turned out to be not taken, the logic has to revert back to a previous check point, or prevent effects of already executed instructions of that stream. Together with out-of-order execution this creates a requirement to specifically "kill" instructions in the pipeline. Additionally, some floating point instructions are implemented in hardware but are iteratively executed within the pipeline - e.g. divide, convert and some multiply instructions. These often cannot overlap with other instructions in the pipeline due to resource conflicts and, therefore, issuing of new instructions has to be prohibited. Since this decreases throughput of the pipeline such blocking conditions should be minimal, and together with other requirements such as functional clock-gating require intricate control.

Formal and semi-formal verification techniques constitute increasingly-prevalent methods by which to attempt to close the coverage gap due to simulation. For example, numerous industrial approaches have proposed the use of a combination of automatic methods and manual theorem-proving techniques to yield complete proofs of correctness of FPUs [2, 3, 4]. However, the introduction of higher level abstractions to model bit level optimized implementations requires manual effort and suffer from false fails introduced in the process. Automated bit level approaches on the other hand are limited to single instructions[1].

In this paper we describe an automated and scalable for-

 DAC '14, June 01 - 05 2014, San Francisco, CA, USA 978-1-4503-2730-5/14/06...$15.00.
http://dx.doi.org/10.1145/2593069.2593096

mal verification based methodology to verify FPUs end-to-end exhaustively inclusive of data flow and control. The approach is largely push-button and reusable across different FPU implementations. The method comprises the following: 1) Verification of all data path aspects by way of model checking the design against a reference model. Practically, this is achieved by executing a single random instruction in an empty pipeline which is checked for numerical correctness against the reference implementation. Such proofs are completed automatically without user-guidance in almost all cases. 2) Verification of control aspects leveraging sequential equivalence checking. This is achieved by treating the DUT as its own reference model, and can be assumed to produce the right result as a consequence of the above (data path verification) albeit not accounting for control in the form of inter-instruction interactions, resource conflicts etc. The set-up calls for submitting a single random instruction in one pipeline and comparing the result of this instruction against the same instruction submitted in the other pipeline as part of a sequence of random instructions. The latter pipeline will enable control features to be invoked as the sequence of instruction execute, and if this in any way interferes with the execution of the "followed" instruction it will manifest as a mismatch between the result produced by the two pipelines. This special set-up allows leveraging structural similarity between the reference and the design (since these are just two instances of the same design) to alleviate capacity issues making the automatic check amenable which otherwise would not be. Experimental results bear out the effectiveness and practicality of the approach in an industrial setting to establish FV as the primary verification vehicle.

The rest of the paper is organized as follows. Section 2 provides definitions to help establish a theoretical basis for the work followed by the relevant theoretical foundations in Section 3. Section 4 provides an overview of our verification framework, and Section 5 provides experimental results from representative applications of the proposed method. Section 6 discusses the portability of the ideas to general purpose FPUs., and Section 7 discusses related work. In Section 8, we summarize and conclude the paper.

2. DEFINITIONS

2.1 FPU-Model

First we define $\mathcal{S}$ as set of states of an execution unit, $\mathcal{D}_{in}$ the set of inputs such as $data = (opCode, operands)$, $\mathcal{D}_o$ the set of output data and $\mathcal{T}$ a set of tags. The unit's next state is defined by

$$ns(S_t, data_t, tag_t, valid_t, stall_t) \rightarrow \mathcal{S}$$

with $S_t \in \mathcal{S}$, $data_t \in \mathcal{D}_{in}$, $valid_t$ an indication of a valid instruction at the inputs of the FPU and $stall_t$ a blocking signal of the pipeline at time t. This definition models all states which typically correspond to the logic's registers, and all transitions based on a given state and the input data. Note that the *stall* is implemented as a feedback to a control component. We call $init \in \mathcal{S}$ the initial state, which the design starts in after a reset.

The output result is defined by:

$$d_{out}(S_t, data_t, valid_t, stall_t) \rightarrow \mathcal{D}_o$$

Further we define the output tag function to identify a completing instruction:

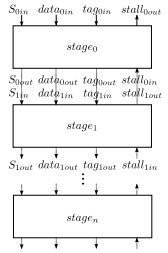

Figure 1: Formalized FPU pipeline

$$t_{out}(S_t, tag_t, valid_t, stall_t) \rightarrow \mathcal{T}$$

The Boolean function we use to compute if the pipeline needs to be blocked when a resource conflict occurs:

$$st_{out}(S_t, stall_t) \rightarrow \mathbb{B}$$

A second Boolean function to compute if a set of outputs is valid:

$$valid_{out}(S_t, valid_t, stall_t) \rightarrow \mathbb{B}$$

For easier reading we will omit these function's parameters when the meaning is obvious. A schematic overview of the definitions is given in fig. 1.

We define $\mathcal{I} := \mathcal{D}_{in} \times \mathcal{T} \times \mathbb{B}$ as the set of all input combinations and $I := (data_0, ..., data_n) \times (tag_0, ..., tag_0) \times (valid_n, ..., valid_0)$ with $I \in \mathcal{I}$ a stream of n instructions.$i_t := (data_t, tag_t, valid_t)$ designates a single instruction of the stream at cycle t. An instruction i_t with $data_t$ is considered issued at time t with tag tag_t when $valid_t = true$ and $stall_t = false$.

Typically FPUs are built as pipelines which allows the execution of several instructions at once. We can reuse the model of an execution unit to formalize each stage within the unit, as in fig. 1. Similar feedback paths can be implemented for $data_t$ with a multiplexer selecting between an input driven by the preceding stage and the stage's own output. These data feedback paths can be used to implement low latency iterative computations.

In general, the number of stages and pipeline depth $\mathfrak{d}$ used to implement $d_{out}(S_t, data_t, valid_t, stall_t) \rightarrow \mathcal{D}_o$ may vary for different designs.

Every instruction implements a function $f : \mathcal{D}_{in} \rightarrow \mathcal{D}_o$ and we call it completed in cycle c when $t_{out} = tag_c$ and the result is $f(data_c) = d_{out}$. The difference of $t - c = l$ is called the latency of the instruction.

Function $L : \mathcal{D}_{in} \rightarrow \mathbb{N}$ defines the latency of an instruction. It is a function of the opcode and operands:

$$L(opCode, operands) = l$$

In general, we can distinguish three sets of instructions:

1. Set $\mathcal{L}_p \subset \mathcal{D}_{in}$ includes all fully pipelined instructions. $\forall\, a, b \in \mathcal{L}_p : L(a) = L(b)$ and typically $L(a) \leqq \mathfrak{d}$. These instructions don't stall the pipeline.

2. Instructions of the set $\mathcal{L}_{mc} \subset \mathcal{D}_{in}$ are called multicycle. Their latency is constant and $\forall\, a \in \mathcal{L}_{mc} : L(a) > \mathfrak{d}$. These instruction stall the pipeline for a fixed number of cycles to avoid resource conflicts.

3. The third set $\mathcal{L}_{var} \subset \mathcal{D}_{in}$ is called variable multicycle. For $a \in \mathcal{L}_{var}$ we distinguish $L(a) > \mathfrak{d}$ and $L(a) \leqq \mathfrak{d}$. The latency of these instructions is not constant for a given *opCode* but depends on the actual *operands*, and the pipeline is stalled for a variable number of cycles depending on it. Examples of such instructions

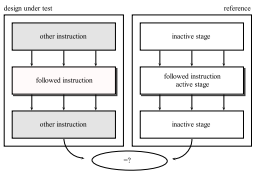

Figure 2: Verification model

are divide algorithms. If a divide instruction detects a "divide by zero" condition then the iterative part of the execution is skipped, otherwise normal computation is performed.

The instructions sets are characterized by the following properties $\mathcal{L}_p \cap \mathcal{L}_{mc} \cap \mathcal{L}_{var} = \{\emptyset\}$ and $\mathcal{L}_p \cup \mathcal{L}_{mc} \cup \mathcal{L}_{var} \equiv \mathcal{D}_{in}$.

2.2 Correctness Criterion

We now define the correctness criterion for sequential execution of instructions. We leverage existing methods to prove functional correctness of a single instruction i executed in an otherwise empty pipeline [1]. The result compare against a high level reference establishes IEEE compliance of the data path of each instruction and covers the instruction end-to-end. It is easily reusable and has been extended over a couple of processor generations. We verify the data path and control signals against this high level reference. In the remainder of this section we therefore assume d_{out}, $valid_{out}$, t_{out} to be proven correct with $S_c = init$ and $valid_x = false$ for every cycle $x \neq t$. The established correctness of the data path under these constraints is represented by a design instance that implements d^{ref}_{out}. The design-under-test is represented by an instance d^{dut}_{out} and all other functions are labeled accordingly.

We call i_t the instruction to be proven, and $I^{ref} := i_t$ and $I^{dut} := i_0, ..., i_t, ..., i_m$ two sequences. The instruction $i_t = (data_t, tag_t, valid_t)$ is called correct iff the design-under-test computes the same result for it in both sequences. For the proof we follow a randomly chosen instruction i_t. The formalized argument for the result output can be stated as:

$$d^{ref}_{out}(init, data_t, valid^{dut}, stall^{dut}_t) = d^{dut}_{out}(S_t, data_t, valid^{dut}_t, stall^{dut}_t)$$

For functions $t^{dut}_{out}, valid^{dut}_{out}, st^{dut}_{out}$ we follow a similar argument. We check that the FPU's state at the time of issue t does not have any affect on the result. Note that the issue into both happens at cycle t, and therefore, *valid* and *stall* also have to be the same to allow i_t to be completed simultaneously, which makes the argument easier and will result in more internal equivalences. With the IEEE compliance of d^{ref}_{out} proven for i_t and the above result, we can transitively conclude that the sequenced execution of i_t implements correct computation as well.

3. MODEL

In this section we describe a model to prove the correctness criteria defined above. Our model consists of two instances of the FPU which is to be verified, F^{dut} and F^{ref}. The first instance F^{dut} we designate as the design-under-test, while the other instance F^{ref} is treated as the reference. We issue a finite instruction sequence I into F^{dut} and a single instruction I^{ref} into F^{ref}. Both instances are initialized to $S = init$.

The sequences I^{dut} and I^{ref} are finite and of minimal length to model a sequence to bound our proof $I^{dut} := i_{irr}, i_t; I^{ref} := i_t$ where i_{irr} is an irritator instruction and i_t is the followed instruction.

The irritator instruction i_{irr} is issued to F^{dut}, and depending on its class it might stall the pipeline. In order to issue according to the correctness criteria defined in 2.2, $st_{out}(S_t, stall_t)$ has to be the same for both instances. Since we assume $st^{ref}_{out}(init, stall_t) = true$ proven for our reference F^{ref}, an issue to it is always possible in cycle t. However F^{dut} might be blocked due to the irritator instruction, i.e. $stall^{dut}_t = st_{out}(S_{t-1}, stall_{t-1})$. We, therefore, reuse $st^{dut}_{out}(S_{t-1}, stall_{t-1})$ for F^{ref}as well; in other words we are delaying the issue of i_t in F^{ref} until i_t can be issued to F^{dut}. After i_t is issued it will pass through the pipeline in the same manner for both pipelines.

It may be noted that we can distinguish two sets of pipeline stages when we compare the two instances. In F^{dut} pipeline stages are active due to additionally active instructions as depicted in fig. 2, while these stages are inactive in F^{ref} since the corresponding irritator instruction wasn't issued to it. However, the stage that is active due to i_t is active in both instances at the same time since we issued it simultaneously and its state should be the same.

Although we would need to consider all possible delays between $i_t$and i_{irr} to achieve completeness for all issue scenarios, it has been our experience that most errors occur when the two instructions are issued as close as possible, resulting in more overlap in the pipeline's control logic. We repeat the same for the reverse sequence $I^{dut} = i_t, i_{irr}$.

3.1 Flush

An important aspect to analyze and establish correctness of instruction sequences in presence of is flush. Pipeline flushes are necessary when processor architectures support speculative code execution. This is a technique in which, for example, both cases of a branch are executed in parallel until the branch condition is evaluated. Then the not taken branch is aborted by a flush. Depending on the granularity of such an abort an architecture might distinguish between flushing a specific instruction or a set of instructions currently in execution in the pipeline. Flushes require careful modeling and targeted scenarios to verify in simulation-based verification. In our approach flushes blend right in into the overall approach and the general correctness criteria remains the same. We simply enable flushes to be effected randomly in the instruction stream, but not the followed instruction. In our model instruction i_{irr} is the target of the flush and instruction i_t is compared between the two pipeline instances to check for same result being computed in spite of the flush. The modeling is easily extended to flushing sets of instructions of the instruction sequence I^{dut}.

3.2 Forwarding

Instructions of $i \in \mathcal{I}$ can further be classified into two classes - independent and dependent. Independent instructions are instructions that operate on different sets of (non-overlaping) registers, and don't require results from another FPU instruction to be issued as intermediate values to arrive at final results. Dependent instructions, on the other hand, reuse at least one register value that has been computed before. Typically, computations which operate on a lot of operands fall into this category as those may be split

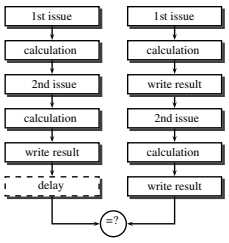

Figure 3: Forwarding model

into several instructions where the later instructions reuse intermediate results computed by earlier instructions.

To speed up dependent instructions, FPUs are designed to support by-passing of results. Instead of writing intermediate results to a register file and reading it upon the issue of the next dependent instruction, the result bus can directly drive the input operand bus. A multiplexer on this bus selects whether the forwarded result or a register content is selected as the input operand for the next instruction. Since writing and reading to the register file adds latency to the execution of an instruction it is beneficial to use the forwarded values from the result bus. However, this adds to the control complexity of the design as such cases are detected by the FPU itself and acted upon subsequently to effect the forwarding.

Sequences of instructions inclusive of forwarding are of special interest as forwarded instructions do not lend themselves directly to a correctness proof in our modeling framework. We need to extend our model to verify these. The foundation remains the same which is to work with duplicate instances of the same pipeline and compare results from the execution in the two. The difference this go round is both the design-under-test and the reference pipelines execute a sequence of instructions.

Whether forwarding can be leveraged by the pipeline or not depends on the issue delay between the dependent instructions. To enable forwarding the second instruction must be issued exactly when the result of the first is available to be reused as an operand. This is typically controlled by the instruction sequencer that can detect such dependencies according to the latency of the different instructions. The execution pipeline will then automatically detect that the 2nd instruction can reuse the result of the 1st and will enable the forwarding. If the issue delay and the latencies don't exactly match, forwarding will not be used and the two instructions are executed as a regular sequence. The model is shown in fig.3

We observe that the forwarding is correct when the computation with forwarding enabled yields the same result as the computation in a regular sequence without forwarding. We formalize the correctness criteria with $I = i_1, i_2$ - an instruction sequence consisting of two dependent instructions $i_1$and i_2. We call the issue cycle of these t_1 and t_2 respectively. The result of i_1 is defined as follows:
$d_{out}(S_{t_1}, data_{t_1}, valid_{t_1}, stall_{t_1}) = data_{t_2}$

Since i_2 is dependent on i_1 we model the result of the first instruction to be the input of the second. We formulate the result function of i_2 in its completion cycle c_2 as: $d_{out}(S_{t_2}, d_{out}(S_{t_1}, data_{t_1}, valid_{t_1}, stall_{t_1}), valid_{t_2}, stall_{t_2}) = data_{c_2}$.

Assuming t_2 to be the earliest cycle in which i_2 can be issued without forwarding, we define $t_2 - g = f_2$ the earliest cycle in which the issue is allowed with forwarding enabled. We call g the gain due to forwarding. Our correctness criteria then follows as:

$$d^{ref}_{out}(S_{t_2}, d_{out}(S_{t_1}, data_{t_1}, valid_{t_1}, stall_{t_1}), valid_{t_2}, stall_{t_2}) = d^{dut}_{out}(S_{f_2}, d_{out}(S_{t_1}, data_{t_1}, valid_{t_1}, stall_{t_1}), valid_{f_2}, stall_{f_2})$$

Obviously, the forwarding should be transparent to the result function. Note that valid and stall signals are not necessarily the same. However it makes sense to consider these to be the same between the two models. Otherwise it would imply that i_1 always causes a pipeline stall at cycle f_2 and forwarding couldn't be enabled at all.

4. (CONDITIONAL) EQUIVALENCE

The above models are effectively a sequential equivalence check of a design against itself. We take advantage of the fact the data flow has been verified in a separate step to aid in verifying the control as it pertains to issuing different instructions with restrictions such as dependencies, or hazards such as flushes and forwarding to simplify an otherwise difficult end-to-end check with model checking. A model set-up as an equivalence check is readily able to leverage internal equivalence points between the two instances to enable conclusive verification of the design as a whole, and an effective bug hunting vehicle which can be gotten off the ground quickly and early to weed out the obvious bugs and to serve as a designer verification platform. This facilitates an automated check of correctness without the need for manual intervention. There are certain quirks to the modeling which require for us to massage the model in certain interesting ways to guide the tool (sequential equivalence checker), which we describe next.

Even though the above is a sequential equivalence check, but the internal equivalences do not hold at all times rather only in specific cycles (due to the special nature of the driving with a single instruction in pipeline versus a sequence in the other) when the computation "reaches" corresponding stages in the two pipelines. Hence it is an instance of conditional equivalence[5]. The pipeline stages in which the followed instruction becomes active should be equivalent while every other pipeline stage at this cycle likely isn't. The bounded model check allows us to unfold the pipeline and it should be rather simple to fig. out which pipeline stages from which cycles are actually relevant to the result. However, depending on the actual type of the instruction this may be harder as additional conditions beyond just the cycle have to be taken into account by the model checker. We found it beneficial to guide the model checker with hints or "lighthouses" to help ascertain pipeline stages which are equivalent, which then makes the analysis of subsequent stages easier. These lighthouses essentially are automatically generated events which check for equivalence of a particular stage across the two models enabled by the stage's active signal. The events are either proven or disproven; the former aids the verification while the latter is without consequence and simply discarded.

We distinguish several "degrees" or notions of freedom, and thus correctness, in the above testbenches to incrementally build-up a full end-to-end proof. First, we can compare an instruction following another one regardless of the state left behind by the first instruction. The result of the second instruction needs to be the same as if the 2nd instruction was

executed in an empty pipeline. One could try to abstract this with a random initial state of the DUT in which the 2nd instruction is issued but often this includes a lot of false fails due to illegal states and figuring out the legal set is non-trivial. Hence, actually issuing the 1st instruction followed by the 2nd instruction is a lot simpler. Second, we start by stressing the model (consisting of the two pipelines) with an interesting mix of instructions from the different classes $\mathcal{L}_p, \mathcal{L}_{mc}, \mathcal{L}_{var}$ outlined above and submitted in a sequence with back-to-back issues as the lowest level of granularity. In doing so we need to especially consider all possible timings between instructions, since it is not known when a residual state might affect the 2nd instruction. Maybe we need to wait an additional couple of cycles before we should issue the 2nd instruction instead of sending them back-2-back (because this scenario might hit the bug). We can expand in this manner to consider larger bounds to cover more windows. That said, there is a practical bound (a fix point) beyond which the effects of such interactions will be masked due to the fact the blocks are clock-gated. Finally, in the most complex and complete case we ought to consider an infinite sequence of instructions and do a full unbounded check to include all possible interactions and window conditions.

5. EXPERIMENTAL RESULTS

In this section we present experimental results gathered on design data from two high performance processors. The results were obtained on a 64bit POWER 6 running at 4.70Ghz running our in-house formal verification toolset *SixthSense* [10]. The initial model for experiments 1 and 2, including driver and the two design instances contains 9744 variables, 3.664.897 AND-gates and 214.046 registers. In table 1 we give the runtime for bounded proofs, the memory consumption and model statistics after initial logic reductions such as constant propagation and structural cone-of-influence analysis.

In experiment 1 the forwarding or bypass logic is tested between instructions of the set $\mathcal{L}_p$. Two instructions are randomly picked from a permissible set. The second instruction which is dependent on the first instruction's result would be issued few cycles before the first instruction completes with the bypass select signals driven appropriately. The second instruction executed in the reference would not be issued before the first instruction completes, which then eventually picks the result of the first instruction through the normal register read with all bypass select signals set to zero. We observe that the model size, run time and memory consumption correlate with the size of the instruction set that the driver can choose from. We can easily create several sets of proofs to allow fast turn around and complete checking of all forwarding cases, enabling quick designer feedback in the process.

In experiment 2 the two instructions of $\mathcal{L}_p$ are interleaved a cycle apart. Both are selected randomly from a permissible set. A flush is targeted to the first instruction at the first execution cycle. The second instruction which is not part of the flush is executed to completion and its result updates the register file, whereas in the reference only the second instruction is issued. Both results are compared upon completed execution. Again, we increased the size of the permissible instruction set through experiments 2a,2b and 2c. Note that the model size of 2a is significantly smaller as the algorithms of the initial logic reduction in the model checker work in a resource limited manner and are able to yield much more reductions within the limits as for the bigger models 2b and 2c. Nevertheless, the runtimes demonstrate that we can easily create a set of proofs covering all possible sequences and flush cases.

We further implemented proofs for sequenced execution of instructions from $\mathcal{L}_{mc}$ and $\mathcal{L}_{var}$. These were carried out in a similar manner but on a different FPU design. The initial model for this design contains 7593 variables, 1.082.359 AND-gates and 119.561 registers. For experiments 3 and 4 we give the runtime for a set of example sequences containing variable latency and multicycle instructions. The runtime for these experiments were measured on a Xeon™ E5-2680 2.7GHz running a 64bit Linux™. We randomly picked 2 instructions from both sets and proved several sequences of combinations of these. In experiment 3 we present results for a 64bit decimal floating point compare instruction that followed a 128bit decimal floating point add in 3a and a 64bit binary floating point convert to decimal in 3b. For experiment 4 we repeated the same but proved a 64bit decimal floating point divide that followed the 128bit add from 3a and the 64bit convert from 3b respectively. The results confirm quick turn around for most scenarios, and given the complexity, reasonable runtimes for the most complex instruction sequences where two variable latency instructions follow each other.

6. PORTABILITY

The overall approach can be easily ported to any FPU environment. There are no architecture specific requirements other than the specifics of flushing and how instructions can be issued as sequences. Furthermore, the method easily integrates into the existing verification environment since it largely reuses drivers for the single instruction proofs. Consequently we can start the sequence verification almost as soon as a driving environment for single instructions is available. This enables designer verification early in the design process starting with single instruction proofs to verify the data path and, subsequently, to verify the control path which typically becomes rather complex when clock-gating and flushing are implemented. Given the push-button nature of the testbench the whole set-up can be given in the hands of the designers which enables them to fix bugs before any HDL changes are committed to the repository. In case a miscompare is found by the formal verification tool, we further found this methodology to be beneficial for debugging. In such a case the generated traces contain the single instruction execution in the reference instance and the sequenced execution in the device-under-test, making it very easy to compare the two behaviors to identify the root cause of the problem.

7. PREVIOUS WORK

Floating-point verification has been the subject of extensive investigation in academia and industry, both to verify correctness of the computation and control aspects as it relates to instruction execution and pipeline control. Data path verification was the focus of earlier efforts which has over time been extended to verify FPU control and allowed for formal verification approaches to assume a central role in verifying these complex logics, and replacing simulation outright in some cases. There is a large body of work from Intel outlining methods using customized tool sets combining

experiment	content (back-to-back)	wall-time	memory consumption	model vars	model registers	model ANDs
1a	forwarding (5 instructions)	00h:05m:56s	1.1GB	1,555	21,509	197,907
1b	forwarding (10 instructions)	00h:17m:29s	4.9GB	1,586	29,467	344,522
2a	flush (5 instructions)	00h:03m:57s	1.1GB	308	306	2,423
2b	flush (10 instructions)	00h:04m:18s	1.1GB	1,563	19,649	169,447
2c	flush (15 instructions)	00h:14m:35s	4.8GB	1,773	33,406	434,127
3a	$\mathcal{L}_{mc}$ followed by $\mathcal{L}_{mc}$	00h:01m:22s	1GB	143	6,256	50,475
3b	$\mathcal{L}_{var}$ followed by $\mathcal{L}_{mc}$	01h:13m:22s	3.6GB	210	7,512	59,917
4a	$\mathcal{L}_{mc}$ followed by $\mathcal{L}_{var}$	00h:24m:37s	1.8GB	141	6,107	96,172
4b	$\mathcal{L}_{var}$ followed by $\mathcal{L}_{var}$	06h:06m:17s	7GB	275	11,980	102,551

Table 1: Experimental results - pipelined instructions

STE and theorem proving to verify FPUs end-to-end. The most recent exposition of these works from CAV 2009 [2] and FMCAD 2012 [3] likely require implementation-specific tedious manual effort and intricate data structures to make those complete, especially as there is a need to reason about intricate sequentially deep state machines as implemented in the RTL of high performance FPU designs. In a similar fashion AMD uses the ACL2 theorem prover to obtain proofs of correctness of FPUs with manually-guided proofs [4]. In contrast our verification method leverages known techniques and combines them in novel and unique ways to enable end-to-end verification of FPUs in a fully automatic manner. Similar self-referential verification methods were proposed earlier[9], in contrast our work concentrates on end-to-end checking of an entire unit, rather then smaller building blocks within an arithmetic circuit with a focus on verifying specific aspects. e.g. data computation. We are not aware of any other efforts with the level of automation and ease of use as ours. Indeed, the effort can be gotten off the ground quickly to serve as a vehicle for early design feedback and bug discovery, and biased towards a proof as the design evolves and stabilizes.

8. CONCLUSION

In this paper we presented an approach to enhance automated formal verification methods to verify FPU data paths to include verification of control aspects of the design to yield an automated end-to-end verification of high performance FPU designs. The method combines known approaches to formally verify data flow of FPUs with executing a single instruction in an empty pipeline against a reference model derived from a standardized spec (e.g. IEEE), and a sequential equivalence check (SEC) of the design implementation against itself with following the computation of a randomly chosen instruction to stress control. Scalability in the former is achieved by implementation agnostic case-splitting derived from the reference model on a per instruction basis [5], and in the latter by way of leveraging concepts of conditional equivalence checking [5] to make the SEC proofs amenable.

In future, our primary focus is to extend this work to include infinite random sequences. Although proofs for these are possible right now for sequences containing only pipelined instructions, once multicycle instructions are included in the set we are not able to get full proofs yet. For these we are currently limited to specific sequences.

9. REFERENCES

[1] Christian Jacobi, Kai Weber, Viresh Paruthi, Jason Baumgartner, "Automatic Formal Verification of Fused-Multiply-Add FPUs", Design Automation and Test in Europe proceedings, pp.1298-1303, 2005

[2] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, Armaghan Naik "Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation", 21st International Conference Computer Aided Verification proceedings, pp.414-429, 2009

[3] M Achutha KiranKumar V and Aarti Gupta and Rajnish Ghughal, "Symbolic Trajectory Evaluation: The Primary Validation Vehicle for Next Gen Intel® Processor Graphics FPU", 12th International Conference Formal Methods in Computer Aided Design proceedings, pp. 149-156, 2012

[4] Peter-Michael Seidel, "Formal Verification of an Iterative Low-Power x86 Floating-Point Multiplier with Redundant Feedback", 10th International Workshop on the ACL2 Theorem Prover and its Applications, pp. 70-83, 2011

[5] Jason Baumgartner, Hari Mony, Michael Case, Jun Sawada, Karen Yorav, "Scalable Conditional Equivalence Checking: An Automated Invariant-Generation Based Approach", 9th International Conference Formal Methods in Computer Aided Design proceedings, pp. 120- 127, 2009

[6] Merav Aharony, Emanuel Gofman, Elena Guralnik, Anatoly Koyfman, "Injecting Floating-Point Testing Knowledge into Test Generators", 7th International Haifa Verification Conference HVC revised selected papers, pp. 234-241, 2011

[7] Jeff Rupley, John King, Eric Quinnell, Frank Galloway, Ken Patton, Peter-Michael Seidel, James Dinh, Hai Bui, Anasua Bhowmik, "The Floating-Point Unit of the Jaguar x86 Core", 21th Symposium on Computer Arithmetic ARITH proceedings, pp. 7-16

[8] David Monniaux, "The pitfalls of verifying floating-point computations", ACM Transactions on Programming Languages and Systems TOPLAS Volume 30 Issue 3

[9] Ying-Tsai Chang, Kwang-Ting Cheng, "Self-Referential Verification for Gate-Level Implementations of Arithmetic Circuits", Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on (Volume:23 , Issue: 7), pp. 1102-1112, 2004

[10] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations, FMCAD, Nov. 2004.