Formal Methods in Computer-Aided Design 2022

Timed Causal Fanin Analysis for Symbolic Circuit Simulation

Roope Kaivola
Core and Client Development Group
Intel Corporation
Portland, OR, USA
roope.k.kaivola@intel.com

Neta Bar Kama
Core and Client Development Group
Intel Corporation
Haifa, Israel
neta.bar.kama@intel.com

***Abstract*—Symbolic circuit simulation has been the main vehicle for formal verification of Intel Core processor execution engines for over twenty years. It extends traditional simulation by allowing symbolic variables in the stimulus, covering the circuit behavior for all possible values simultaneously. A distinguishing feature of symbolic simulation is that it gives the human verifier clear visibility into the progress of the computation during the verification of an individual operation, and fine-grained control over the simulation to focus only on the datapath for that operation while abstracting away the rest of the circuit behavior.**

In this paper we describe an automated simulation complexity reduction method called *timed causal fanin analysis* that can be used to carve out the minimal circuit logic needed for verification of an operation on a cycle-by-cycle basis. The method has been a key component of Intel's large-scale execution engine verification efforts, enabling closed-box verification of most operations in the interface level.

As a specific application, we discuss the formal verification of Intel's new half-precision floating-point FP16 micro-instruction set. Thanks to the ability of the timed causal fanin analysis to separate the half-precision datapaths from full-width ones, we were able to verify all these instructions closed box, including the most complex ones like fused multiply-add and division. This led to early detection of several deep datapath bugs.

***Index Terms*—Formal Verification, Symbolic Simulation, Complexity Reduction**

I. Introduction

Comprehensive formal verification of execution engines has been standard practice in virtually all Intel® Core™ and Intel Atom® processor development projects in the last two decades, and extensive infrastructure has been built to support these efforts. Formal verification of Intel processor execution engines is primarily based on *symbolic circuit simulation*, a technology extending usual digital circuit simulation with symbolic values, representing sets of concrete values in a single simulation [1], [2], [3], [4], [5].

Full correctness of processor execution engines is indispensable for product quality, as errata in basic execution datapaths tend to be both customer visible and un-patchable. Due to the size of the data space and the difficulty of identifying and covering all internal corner cases with either pre-silicon or post-silicon testing, formal verification is the only approach that can ensure sufficient quality, especially for complex floating point datapaths.

Execution engines in industrial processor designs typically combine a set of different pipelined datapaths into a single design component. To minimize circuit size, each individual datapath multiplexes logic for a family of related operations, controlled by operation-specific control signals. The datapaths may support different latencies, with simpler operations executing in fewer pipestages than complex ones. Many datapaths are implemented as straight pipelines, however certain operations may use iterative algorithms with feedback loops. Designs also usually contain bypass networks that route data from the datapath outputs directly back to the inputs, avoiding the delay of going through a register file. The execution engine in a contemporary Intel processor has several million logic gates and hundreds of thousands of flip-flops, and the source code for it consists of hundreds of thousands of lines of code in a hardware description language.

Focusing on the verification of an individual operation implemented in an execution engine, we can conceptually distinguish two different sources of verification complexity:

1) the inherent complexity of the plain datapath for the operation, ignoring all other functionality of the execution engine, and
2) the complexity caused by the presence of the rest of the execution engine, and its possible effects on and interferences with the datapath of the operation.

As an example of the first, any datapath involving multiplication can be expected to pose a verification challenge, irrespective of any surrounding logic. For the second, the isolation of the result of an operation in a shared result bus depends on the control logic of all the datapaths sharing the bus. In a practical verification task, the verification engineer faces these two dimensions simultaneously, and the complexity caused by the surrounding logic may make the verification of even inherently trivial datapaths, such as bitwise OR, challenging or infeasible.

Considering the inherent datapath complexity, without surrounding environment, the large majority of operations implemented on an execution engine can be directly verified

Intel provides these materials as-is, with no express or implied warranties. Intel processors might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Intel, Intel Core, Intel Atom, Pentium and Intel logo are trademarks of Intel Corporation. Other names and brands might be claimed as the property of others.

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_16

This article is licensed under a Creative Commons Attribution 4.0 International License

by symbolic simulation in a closed-box fashion. This is the ideal scenario due to the many advantages of closed-box verification: a well-defined specification, no need of insight into implementation details, and low sensitivity to internal design changes. For the most complex operations, especially complex floating-point arithmetic such as multipliers, fused multiply-adders and dividers, this straightforward approach is computationally infeasible, and verification is done by means of decomposed reference models, requiring time and both design and verification expertise.

If the plain datapath for an individual operation were to be isolated from the surrounding logic, for most operations it would be amenable to verification by a variety of techniques besides symbolic simulation. However, in practice, the datapath is tightly enmeshed with the rest of the execution engine, and there is no straightforward way to isolate it. In this respect, symbolic simulation has a unique advantage over many competing verification approaches, such as formal equivalence verification or traditional model checking: it allows the verification engineer to understand the computational progress of an operation in the circuit in very concrete terms, to carve out a minimal amount of logic that needs to be simulated for the datapath of that specific operation, and to efficiently abstract away the rest. In other words, symbolic simulation provides an effective way to separate the two sources of verification complexity. The main technical ingredients enabling this ability are discussed in Section II.

Nevertheless, as execution engines typically implement thousands of individual operations, and for each operation the datapath controls are wired differently, the cost of the human effort to analyze and isolate each datapath becomes a limiting factor.

In this paper we describe an algorithmic technique called *timed causal fanin analysis* to derive a tight over-approximation of the circuit logic relevant for the simulation of the datapath of an individual operation (Section III). This method effectively automates the human process of determining the minimal circuit logic for a specific datapath. It is based on the use of information from an earlier, more abstract and less accurate symbolic simulation run to reduce the fanin cone of the logic of interest on a cycle-by-cycle basis. The method enables fully automated closed-box verification of most operations in an execution engine, not just for an isolated datapath, but in the context of the full design unit. It is meaningful only in the context of verification by symbolic simulation. The method has been a key technical enabler in Intel's large-scale verification initiatives over the span of many years [3], [6]. However, the current paper is the first detailed exposition of the method in the public domain.

For a recent example illustrating the effects of timed causal fanin analysis, in Section V we discuss the verification of the new FP16 floating-point instruction set on a recent Intel Core processor design. Since the Intel 8087 floating-point co-processor was introduced in 1980, Intel processors have supported single, double, and extended precision floating point formats. The formal verification of complex operations such as multiplication, division, etc., on these formats has always required decomposition, making such verification a time-consuming expert task. Recent Intel Core processor designs have added a new shorter half-precision floating-point format, also known as FP16 [7]. Because of the lower datapath width, the inherent verification complexity of FP16 datapaths is also lower, bringing them closer to the set of designs that one could hope to verify without decompositions.

As a practical result, we found out that *all* FP16 micro-operations could be verified closed box, including the complex multiplication, fused multiply-add, division and square root operations. This led to fast verification convergence and early detection of several high complexity datapath bugs. The timed causal fanin analysis technique was particularly crucial for datapaths shared between FP16 and higher precision operations. It allowed us to avoid simulating the higher-precision logic, the complexity of which would have otherwise made verification impossible.

II. SYMBOLIC CIRCUIT SIMULATION

Symbolic simulation extends traditional digital circuit simulation by allowing the input stimulus to contain *symbolic variables* in addition to the concrete values 0, 1 or X [1]. These symbolic variables are effectively names of values, denoting sets of possible actual concrete values. In the simulation, these symbolic values propagate alongside the concrete values, and in each logic gate, they may be combined with each other or one of the concrete values to result in either a concrete value or a logical expression on the symbolic variables, represented by an expression graph. In this paper, as in most of symbolic circuit simulation verification practice, we use the binary decision diagram (BDD) representation for symbolic expressions [8]. See Figure 1 for an example.

In a bit level symbolic simulator, a single symbolic variable a corresponds to the set of boolean values containing both 0 and 1. If stimulus to a symbolic simulation refers to the variables a, b and c, the internal signals might carry values like $a \wedge b$ or $a \vee (b \wedge \neg c)$. Usual logic rules apply: if the inputs to an AND-gate are a and 1, the output will be a, if the inputs to an AND-gate are a and b, the output is the logical expression $a \wedge b$, and if the input to a NOT-gate is b, the output will be $\neg b$. In symbolic simulation, a specific symbolic variable is associated with a specific signal and time in the stimulus. This does not fix the value, but instead gives a name that can be used to refer to the value.

The special value X is used in symbolic simulation to denote a universal undefined or unknown value, which propagates according to rules such as in Figure 2. The value X denotes lack of information: we do not know whether the value is 0 or 1. The propagation rules reflect this intuition. Symbolic simulation uses X's as an abstraction mechanism: unlike symbolic variables, X's are an over-approximation of Boolean circuit behavior. Both symbolic variables and X's allow us to verify a property over a single symbolic trace, and conclude that it is valid over every possible trace instantiating the X's and the symbolic variables with 0's or 1's. This ability of a

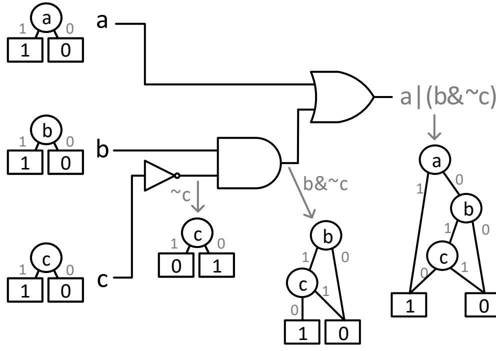

Fig. 1. Symbolic expressions in simulation

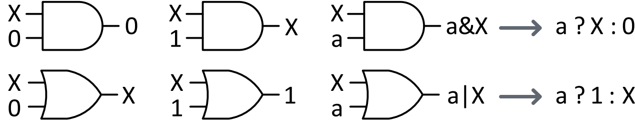

Fig. 2. Logic with the undefined value X

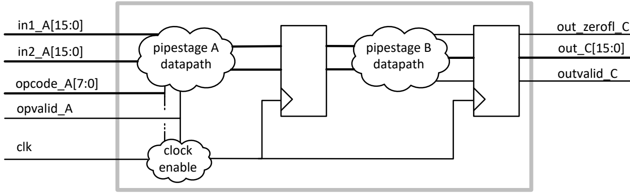

Fig. 3. Simplified ALU

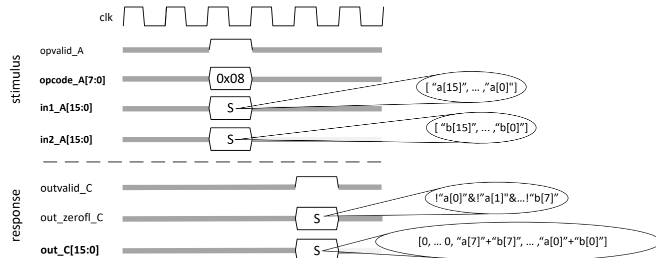

Fig. 4. Symbolic trace

single symbolic trace to cover all behaviors of a circuit allows us to use symbolic simulation as a formal verification method.

Figure 3 depicts a simplified pipelined ALU circuit with a 16-bit wide two-cycle datapath from inputs to outputs, and Figure 4 depicts a typical symbolic trace that might be used in the verification of this ALU, focusing on a single instance of an eight-bit wide bitwise OR operation. In the stimulus, the control signals are driven with concrete values corresponding to the operation, and the input data is driven with symbolic variables $a[15], \ldots, a[0]$ and $b[15], \ldots, b[0]$ in the one cycle in which the operation is issued. In all other cycles these signals have the undefined value X (gray waveform). In the simulation, the values of the output data and zero flag two cycles later are then expressions on the symbolic variables associated with the input data, and in all other cycles they are X's.

The practice of verification by symbolic simulation has similarities to bounded model checking (BMC), however with two important differences. First, BMC considers instances of a property in a time window up to a given bound, whereas symbolic simulation focuses on one fixed instance of a property, and second, BMC starts from a properly initialized state of a system, and symbolic simulation from an unconstrained state. The focus on one fixed instance of a property can be seen as a distinguishing aspect of symbolic simulation.

The size of the symbolic expressions flowing in the signals of the circuit during the simulation is the most crucial complexity metric and the limiting factor determining what can and cannot be computed. We strive to minimize this symbolic complexity in several ways:

1) by choosing the properties to be verified so that they are as narrowly targeted as possible and by restricting the circuit simulation to only those scenarios that are relevant for the property under verification,
2) by limiting the number of symbolic variables and concrete 0/1 values used in the simulation stimulus to maximize the use of the default undefined value X,
3) by limiting the set of signals for which simulation values are computed, the times for which those values are computed, and the values that are computed, and
4) by choosing concise representations for the computed symbolic expressions.

For example, in execution engine verification we (1) focus on one operation instance at a time, (2) drive symbolic values on inputs only when the operation instance under verification samples them, (3) simulate only signals that are needed for the datapath of the operation and only at times relevant to the progression of its pipeline, and (4) use a BDD variable ordering that is a good match for the operation.

Symbolic simulation works best with targeted properties of fixed length pipelines, typically of the transactional form

$$\textit{trigger A at time t is followed by response B at time } t+n$$

To restrict circuit behaviors to cover only cases where the trigger of the property under verification is true, we use the technique of *parametric substitutions* [9], [10]. The basic setup for the parametric substitution algorithm is that we want to verify an implication $C(\bar{v}) \Rightarrow D(\bar{v})$ between two symbolic expressions C and D over a set of symbolic variables $\bar{v}$, and the assumption C in some fashion makes it easier to compute the goal D. The algorithm creates a mapping $\bar{v} \mapsto \bar{p}$ from variables $\bar{v}$ to symbolic expressions $\bar{p}$ such that when the symbolic variables in $\bar{p}$ range over all possible values, the values of the symbolic vector $\bar{p}$ range exactly over the set of assignments to $\bar{v}$ for which the condition C is true. Then, the implication can be verified by checking whether $D(\bar{p})$ holds.

In the context of symbolic simulation, the aim is to check an implication between the trigger and the goal of the property being verified over the traces of the circuit. This is done by computing a parametric substitution from the trigger, carrying

out the symbolic simulation with the parametrized expressions $\bar{p}$ instead of the original variables $\bar{v}$ in the stimulus, and by checking that the verification goal is true in the resulting trace. For a concrete example of parametric substitution for symbolic simulation triggers, please see Section III below, especially Figure 6 and the related discussion.

The techniques for limiting the sets of signals, times or values for which simulation is done are collectively called *weakening*. In weakening the user instructs the simulator to replace a value that would otherwise be computed with the undefined value X. We distinguish three kinds of weakening:

- *Universal weakening*, where the user instructs the simulator to replace the values of certain signals with X's at all times in the simulation. It is equivalent to the concepts of 'free' or 'stop-at' present in many model checkers.
- *Cycle specific weakening*, where the user instructs the simulator to replace the values of certain signals with X's, *but only at specified times*. This technique is unique to symbolic simulation, and the fact that it is even meaningful to talk about signals at specific times in the verification task is directly related to the fact that symbolic simulation focuses on just one fixed instance of the verification goal. Cycle specific weakening is an extremely versatile technique that allows users to apply their intuition about the usage of signals at times relative to the progress of the operation under verification in order to reduce the simulation cost.
- *Dynamic weakening*, where the user instructs the simulator to replace any symbolic value with X, if the size of the expression for the value would exceed a user-given threshold. Dynamic weakening is a robust technique that allows users to quickly resolve many symbolic complexity issues caused by the computation of unnecessary expressions in the simulation without detailed analysis.

Weakening is a safe complexity reduction technique: if we verify a property over a symbolic simulation trace with weakening, the same property also holds over a trace with the same stimulus and no weakening.

The computations in symbolic simulation are conceptually simple and concrete. Further, they can be naturally related to the progress of the operation under verification through its pipeline. This gives the verification engineer fine-grained visibility into the computations on the level of individual signals, enabling precise analysis and mitigation of computational complexity bottlenecks through weakening. In the context of execution engine verification, this visibility allows the verifier to identify the datapath of an individual operation and weaken the surrounding circuit logic. However, when pipelines for different operations are tightly enmeshed in a circuit, it is often time-consuming to determine which signals and simulation times are really needed for a specific operation.

III. Timed Causal Fanin Analysis

As discussed above, the size of the symbolic expressions is the primary capacity barrier in a simulation, and consequently it is very important that we avoid the computation of symbolic values unnecessarily, in contexts where they do not contribute meaningfully to the verification goal. In a forward simulator this is not trivial. When simulating a certain cycle, we do not know yet which signals in that cycle will matter to the verification goal in a later cycle.

One straightforward technique for reducing the set of signals for which simulation needs to compute values is the standard cone of influence (COI) reduction. The validity of a verification goal can only depend on the transitive fanin of signals referenced in it, and therefore signals outside of this set do not need to be simulated. However, for execution engines that contain bypass networks, the circuit forms in practice a nearly strongly connected graph, i.e. almost every signal is in the transitive fanin of almost every other signal, and the cone of influence reduction offers little help.

Another source of reduction comes from the simplifying effect of any global constants in the design. For example, an AND-gate with one input a constant zero does not actually depend on the value of its other input, and that other input can be removed from the fanin of the gate without changing the behavior of the circuit. As designers do not intentionally include dead logic in their designs, such global constants usually reflect circuit functionality, such as test or scan modes, that can be completely disabled for verification purposes. They usually offer only marginal help in reducing simulation scope around the main functionality of a design.

The *timed causal fanin analysis* algorithm is based on the idea of using constants to reduce the fanin cone of interest. However, this is done on a cycle-specific basis, relative to the cycle times in a fixed symbolic simulation, using the concrete 0/1 values present in that cycle only. As with cycle-specific weakening, the fact that we can meaningfully refer to a particular cycle relative to a verification task is specific to symbolic simulation. The three main steps of the method are:

1) Perform a preliminary symbolic simulation to determine *cycle-specific* concrete 0/1 values in the simulation.
2) Compute the transitive cone of influence of nodes and cycles in the verification goal *per cycle*, using the concrete 0/1 values from step 1 to reduce the fanins in each cycle.
3) Compute a cycle-specific weakening list, *per cycle*, that weakens every signal of the circuit except the signals in the transitive cone of influence for that cycle, as computed in step 2.

Step 1 consists of a symbolic simulation run for the circuit with the same stimulus that is used for the main verification run. However, for this initial simulation, the dynamic weakening threshold is low. As described in Section II, this means that any symbolic expressions above the threshold are discarded and replaced with X's in the simulation. The size threshold is specified by the user. All relevant cycles of the resulting stimulation trace are then scoured for all concrete 0/1 values.

It is important to note that this preliminary simulation is much more than just timed constant propagation. First, the trigger of the property has already been factored into the stimulus with parametric substitution, and any concrete 0/1

values implied by the trigger are present in the trace, especially in the pipelined datapath control signals. Second, in addition to the concrete values that the trigger forces directly in the stimulus, also the concrete values that are implied indirectly by circuit logic together with the trigger restrictions are present in the trace, due to the canonicity of the BDD representations and the automatic simplification in BDD operations.

Step 2 consists of a backwards traversal over relevant simulation cycles, starting from the last cycle of interest and proceeding back in time. For each cycle, we compute the causal fanin at that cycle using the concrete 0/1 values present at the cycle to reduce the causal fanin cone.

For a combinational gate s of the circuit, we define the *combinational causal fanin set of s at simulation time t* to be the set of signals s_{in} such that s_{in} is an immediate fanin of s and either

- s_{in} has a concrete 0/1 value in cycle t in the simulation in Step 1, or
- the value of s_{in} may affect the value of s, given all the concrete 0/1 values in the fanins of s in cycle t in the simulation in Step 1.

In short, for each cycle the concrete 0/1 values computed in Step 1 for that cycle are used to reduce the fanin cone of combinational gates. For example, if selectors to a mux have concrete 0/1 values in a certain cycle, only the single mux input that is selected by those selectors is in the timed causal fanin in that cycle.

For a flip-flop (state element) s_{ff} of the circuit, with input s_{in} and clock c, we define the *flip-flop causal fanin set of* s_{ff} *at simulation time t* by the rules:

- If the clock c toggles in cycle t in the simulation in Step 1, then s_{in} belongs to the set.
- If the clock c does not toggle in cycle t in the simulation in Step 1, then s_{ff} belongs to the set.
- If the clock c is X in cycle t in the simulation in Step 1, then both s_{in} and s_{ff} belong to the set.

Conceptually, if we do not know whether the clock toggles or not, both the input and the held value of the flip-flop matter.

For each cycle t, we then define the *timed causal fanin set* $cfan(t)$ as the minimal set of circuit signals satisfying the following rules:

1) If the verification goal directly refers to signal s in cycle t on the simulation, then $s \in cfan(t)$.
2) If signal s is in the flip-flop causal fanin set of a flip-flop s_{ff} at simulation time $(t+1)$, and $s_{ff} \in cfan(t+1)$, then $s \in cfan(t)$.
3) If signal s is in the combinational causal fanin set of a combinational gate s_{out} at time t, and $s_{out} \in cfan(t)$, then $s \in cfan(t)$.

For each cycle t, we compute $cfan(t)$ by starting from the set of signals determined by the rules (1) and (2) and by constructing the transitive closure of the set under rule (3), stopping at the flip-flop boundary.

Step 3 finally constructs a weakening list that for every cycle t replaces the value of every signal not in $cfan(t)$ with X. This weakening list is then used in a full symbolic simulation for the original verification goal. As the computation of the timed causal fanin in Step 2 includes all signals and times that may affect the signal-time references in the property under verification, the weakening list never abstracts with X any values that could contribute to the property. As an optimization, we can alternatively weaken only the barrier of signals whose fanin intersects with $cfan(t)$ but which are not in $cfan(t)$ themselves.

As a point of comparison, consider the same verification task posed as a bounded model checking problem. If we look at just the timed constant propagation aspect of the preliminary simulation, and the concrete 0/1 values directly forced by the trigger, an analogous constant propagation process would take place at an early point inside the SAT call for the BMC problem, resulting in expression simplification similar to the fanin reduction above. As for the concrete 0/1 values indirectly implied by the trigger and the circuit logic, sooner or later they either might or might not be noticed and propagated by the SAT engine, depending on how hard the engine tries to determine constants. However, this whole process is completely hidden from the user, inside a SAT engine. In particular, if a potentially helpful simplification does not happen, either because the engine misses it or because the trigger does not capture the user intent accurately, the issue manifests to the user only through increased run time or the inability of the tool to resolve the verification goal, without actionable feedback that would enable the user to assist the tool.

However, when we use timed causal fanin analysis in the symbolic simulation flow, the results of the preliminary simulation and the concrete 0/1 values that are or are not present are visible and accessible to the user. The values can be queried, viewed as waveforms and root-caused through circuit gates. The user can understand what happens in the simulation and compare that to their intuition and expectations about what should happen. The concept of the timed causal fanin cone itself is based on a clear operational intuition, allowing the user to understand the computation in terms of circuit functionality. A commonly asked debug question is: "why is signal s in cycle t in the timed causal fanin cone of my property, when conceptually it should not matter, for example because it is in a different unit/datapath/pipestage?" This question can be concretely answered by showing a path of dependencies from the given signal and time through fanin relations to some signal and time relevant to the property being verified.

As an example, consider the simplified ALU circuit in Figure 5 with a one-cycle adder unit and a two-cycle multiplier unit. At the interface, the signal *vld* marks a valid operation and *mul* chooses between addition and multiplication. Further, suppose that we are focusing on adder correctness as expressed in the following property, where **N** and **P** are the next-time and previous-time temporal operators, respectively:

$$\underbrace{(vld \wedge \neg mul)}_{\substack{\text{ADD} \\ \text{time } t}} \wedge \mathbf{P} \underbrace{\neg(vld \wedge mul)}_{\substack{\text{NOT MUL} \\ \text{time } (t-1)}} \Rightarrow \mathbf{N} \underbrace{(is_ok(res))}_{\substack{\text{RESULT OK} \\ \text{time } (t+1)}}$$

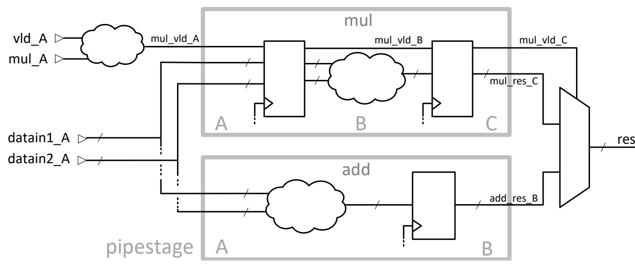

Fig. 5. Simplified ALU with adder and multiplier

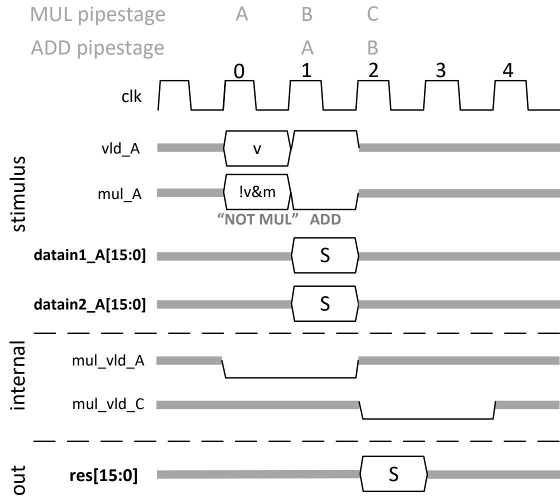

Fig. 6. Stimulus and preliminary simulation trace

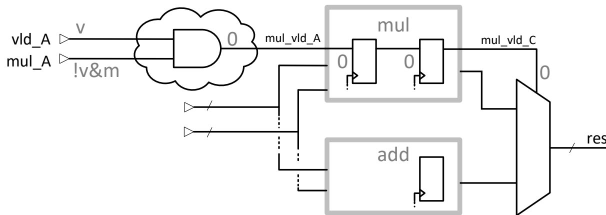

Fig. 7. Internal simplification in control logic

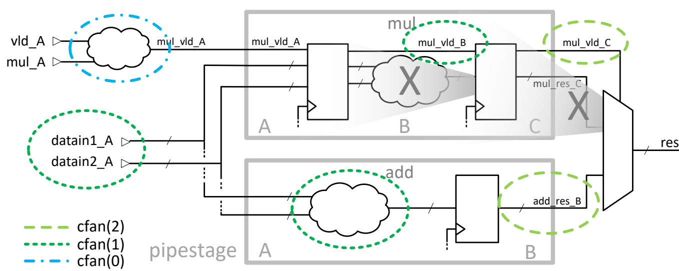

Fig. 8. Timed causal fanin cone computation

Conceptually this property says that if an addition operation is issued, and there is no pipeline hazard from a multiplication operation a cycle ago, then the circuit will produce functionally correct output in the next cycle (where we have omitted the details of 'functionally correct output' and its dependency on the data input signals).

Figure 6 depicts a stimulus and trace for the Step 1 preliminary simulation on the circuit, with an instance of the property above with time $t = 1$, starting in cycle 1 and producing output in cycle 2. The stimulus values for the control signals *vld* and *mul* in cycles 1 and 0 have been generated by parametric substitution from the triggers of the property:

- In cycle 1, the stimulus associates the concrete value 1 with the signal *vld* and the concrete value 0 with the signal *mul*, since this is the only possible assignment satisfying the trigger 'ADD in cycle 1', i.e. $vld \wedge \neg mul$.
- In cycle 0, the stimulus associates a symbolic variable v with the signal *vld* and the symbolic expression $\neg v \wedge m$ with the signal *mul*, reflecting the trigger 'NOT MUL in cycle 0'. Note that the possible values of these two symbolic expressions range exactly over the set of assignments to *vld* and *mul* that make the trigger $\neg(vld \wedge mul)$ true in cycle 0, a guarantee of parametric substitution. Note also that no concrete 0/1 assignment would capture the trigger fully, since there are three possible concrete value pairs satisfying the trigger.

Simplification on internal control signals, as depicted in Figure 7, then leads to the trace of Figure 6. Using the cycle-specific concrete 0/1 values from this trace, Step 2 of the timed causal fanin analysis method proceeds as in Figure 8. In Step 3, all signals and times outside the timed causal fanin of Figure 8 are weakened in the main simulation. Note, in particular, that all multiplier datapath logic is automatically weakened by the timed causal fanin algorithm.

From the perspective of the user applying the timed causal fanin method, the practical workflow can be divided into two stages. First, there is the computation of the causal fanin cone in Steps 1 and 2. In this stage the user may need to adjust a default dynamic weakening threshold for the preliminary simulation in Step 1 or the default depth of the fanin cone traversal in Step 2 to balance two needs. On the one hand, the threshold and the depth of the fanin cone need to be low enough that the steps can be computed in a reasonable time. On the other, the threshold and depth need to be high enough that as many concrete internal values as possible are computed to reduce the causal fanin cone. In this first stage of the work the user also may find out that the verification triggers are not strong enough to guarantee the satisfaction of the verification goals, by simply looking at the causal fanin cone and noticing unexpected causal dependencies. These may either reflect a design bug, or a need to strengthen the triggers to properly capture the intent of the property under verification.

In the second stage of the work the user then applies the weakening list computed in Step 3 in the main simulation, debugs any failures, and repeats the main simulation if necessary. In many instances the main simulation is less resource

intensive than the preliminary one, since although the symbolic expressions that need to be computed are larger, the number of signals for which they actually need to be computed is much lower, thanks to the timed causal fanin weakening list.

The timed causal fanin algorithm is helpful in most symbolic simulation verification tasks, and we use it as a routine step in our verification flow. Already on its own, symbolic simulation is at its strongest for narrowly targeted properties, and the timed causal fanin method accentuates this strength. When comparing the automated weakening provided by the method to manually crafted weakening lists, in our experience the automatically produced weakening is almost always superior, as user time and patience for fine-grained analysis of the design is often limited. As a weak point, the presence of data-qualified clocks in a design tends to reduce the efficacy of the method, as then the timed causal fanin cone will include same combinational logic over multiple cycles.

Two major building blocks underlying the timed causal fanin method are fundamentally BDD-based: first the parametric substitution algorithm, and secondly the automated simplification of symbolic expressions in the internal wires of the circuit, which results in the concrete 0/1 values that are used to contain the fanin cone. If we want to avoid BDD's and simulate with non-canonical expressions and use SAT instead, the same crucial process of identifying simplifying internal concrete 0/1 values could be achieved by speculative SAT queries checking for constants in the preliminary simulation under the trigger assumptions. The sheer number of internal signals in many circuits is a challenge in this approach, though. What works better in practice is a hybrid approach, where the preliminary simulation uses BDD's, with the resulting automated simplification, but the main simulation used for the verification of the goals in carried out with non-canonical expressions and SAT.

IV. Execution Engine Formal Verification

At high level, a single Intel Core consists of a set of major design components called *clusters*. The front-end cluster fetches and decodes architectural instructions and translates them to micro-operations (abbreviated as uops), which the out-of-order cluster then schedules for execution. The execution engine, residing in the EXE cluster, carries out data computations for all micro-operations. The memory cluster handles memory accesses and may contain first level caches. Outside of an individual core is a system-on-chip layer including, for example, a graphics processing unit and a memory controller.

The execution engine for a typical Intel Core processor design implements over 5000 distinct uops in several different units: the integer execution unit (IEU) contains logic for plain integer and miscellaneous other operations, the single instruction multiple data (SIMD) integer unit (SIU) contains logic for packed integer operations, the floating-point unit (FPU) implements plain and packed floating-point operations such as FADD, FMUL, FDIV, etc., the address generation unit (AGU) performs address calculations and access checks for memory accesses, the jump execution unit (JEU) implements jump operations and determines and signals branch mispredictions, and the memory interface unit (MIU) receives load data from and passes store data to memory cluster.

Formal verification of execution datapaths, especially for floating-point and other arithmetic operations has been a focus area at Intel ever since the Pentium® FDIV bug in 1994. The primary vehicle for this work is symbolic simulation, incorporated in Intel's in-house Forte/reFLect verification toolset under the name of Symbolic Trajectory Evaluation (STE) [2]. All Intel Core processor execution engine data-paths since 2005, as well as most Intel Atom processor and Gen Graphics arithmetic engines have been formally verified using symbolic simulation [3], [6].

In formal verification, every uop corresponds to a separate symbolic simulation task. In the verification setup for a single uop the control signals are set to fix the data-path controls to match a single instance of that uop, and symbolic variables on the data are used to exhaustively simulate the data-path instance. The simulation is connected to an abstract functional reference model for the uop through source and write-back mappings, and the output of the design and the reference model compared. These design-dependent mappings extract the intended source and result values for the uop at the relevant times relative to the instance we are verifying.

Formal verification of complex designs would ideally be done by closed-box verification for its many advantages: a well-defined specification, no need of insight into implementation details, and low sensitivity to internal design changes. For a large majority of uops in the execution engine, the data-path can be exhaustively symbolically simulated in one pass at the full cluster level.

However, for complex floating-point arithmetic, such as multipliers, fused multiply-adders and dividers, the computation of symbolic expressions for the datapaths is fundamentally technically infeasible. Instead, the verification of these complex uops is done through a decomposed reference model that splits an operation to several sequential stages, where each stage of the reference model is separately related to a stage of the implementation. With such decomposition cut-points, we reduce symbolic simulation complexity, as each stage on its own produces smaller symbolic expressions than a full input-to-output closed-box simulation. For years, this has been the technique used for all the floating-point types traditionally implemented on Intel designs, i.e., single, double, and extended precision floats.

Decomposed verification is technically much harder than closed-box verification, requiring both special verification expertise and detailed insight into implementation details to map the decomposition stage boundaries to the design. It is also much more sensitive to even small design changes, making the maintenance cost high. Generally, the more stages the decomposition has, the harder the verification task is. The hardest datapath verification tasks on current Intel processor designs are the dividers, which need a series of decomposition stages and advanced complexity management strategies in each individual stage.

V. Half-precision floating-point arithmetic

Floating-point numbers are a binary representation for a subset of real numbers as triples (s,e,m), where the sign s is a single bit, and the exponent e and mantissa m are unsigned bit vectors of some fixed lengths. The IEEE standards on floating-point numbers define several different formats differing on details, as well as special encodings for zeros, infinities, denormal numbers (very small numbers that are below the main range of values representable in a format), and other exceptional values [11]. Since only a subset of the reals is representable as floating-point numbers, not all results of arithmetic operations on floating-point numbers can be expressed precisely as floating-point numbers themselves. Therefore, the IEEE standards define the concept of rounding, determining which sufficiently close representable number should be used, if the accurate result is not representable.

Intel designs have traditionally supported three formats of floating-point numbers: single, double, and extended precision. Recently, as a part of the AVX-512 extension set in the latest Intel Core processor designs, support was added for a new shorter floating-point format, the so-called half-precision or FP16, consisting of one sign bit, five exponent bits and ten mantissa bits [7]. While the new format offers a narrower range and less precision, it allows twice as many values to be packed into a vector than with single-precision floats, doubling the effective performance of vectorized algorithms for applications that do not need higher precision arithmetic.

The architectural and micro-architectural instruction sets of the latest Intel Core processor designs support most common arithmetic half-precision operations natively. Some half-precision uops are implemented in dedicated design units, some others in units shared with higher precision arithmetic. Half-precision division and square root uops are implemented by an iterative design shared with the similar higher precision uops. In contrast to some higher precision operations, denormal input and output values are handled natively for half-precision arithmetic, without microcode assistance.

As the basic datapath for a half-precision uop has only half as many input data bits than the corresponding single-precision uop, we know that the size of symbolic expressions in its simulation is always lower than for single precision. Without experimentation we do not know how much lower, as the symbolic expression sizes can be at best linear and at worst exponential in the number of input bits, depending on the operation. What we do know is that any verification recipes that work for single precision should easily work for half precision. Also, we can realistically hope that the reduction in size might be large enough to obviate the need for decomposition for some of the complex operations, pushing them to the domain of closed-box verification, or at least reduce the decomposition needed. On the negative side, experience shows that native denormal handling tends to materially increase symbolic complexity, as denormals break the separation of exponent and mantissa datapaths. Also, we know that special care will be needed for uops implemented in units shared between half precision and higher precisions to avoid the prohibitive cost of simulating also the higher precision behavior.

From this starting point, we carried out verification of all half-precision arithmetic uops on an Intel Core processor design. The technical learnings from the initiative can be summarized as follows:

- Simple floating-point uops such as comparisons, conversions to and from integers, reciprocals, etc., that allow closed-box verification for higher precisions, were easily verifiable for half precision. As anticipated, floating-point addition (FADD) could also be directly verified, in contrast with higher precisions, where FADD needs an exponent difference-based case split. Timed causal fanin analysis was essential in the separation of the simple uop and FADD datapaths from the complex ones implemented in the same design units.
- As the first result for known high complexity uops, we were able to verify floating-point multiplication (FMUL) directly without a decomposition. This is in marked contrast with higher precisions where decomposition is unavoidable, as the symbolic expression sizes for multiplication are known to be exponential. However, the lower number of mantissa bits for half precision means that we are not too far up the exponential curve yet in the basic datapath for the operation. For FMUL, the datapath is shared with the more complex fused multiply-add (FMA) operation. Timed causal fanin analysis helps FMUL verification by removing FMA-specific parts of the shared datapath, in particular in the rounding logic where FMUL exhibits only a narrow range of possible behaviors compared to FMA.
- Somewhat surprisingly, we were also able to verify half-precision fused multiply-add (FMA) uops without decomposition. This required careful complexity management, and a large case split on addend mantissa values to reduce the symbolic complexity of the basic datapath, with a high total run time. As FMA is the most complex operation on its shared datapath, there is no circuit logic that timed causal fanin analysis could just directly cut out. However, for each case in the case split, the simulation of the basic datapath alone approaches the capacity limits of the tool. How timed causal fanin analysis helps is by removing logic that is on the basic datapath, but is not relevant to the specific case.
- Finally, with heavy use of simplifying case splits and timed causal fanin analysis, we were able to carry out closed-box verification for half-precision division (FDIV) and square root (FSQRT) operations, as well. For division and square root, timed causal fanin analysis was indispensable, as the datapaths are mixed with the higher precision ones, and the long-latency uops have ample potential for uncontrolled symbolic expression growth.

The most complex arithmetic datapath proofs showed that for FP16, verification of all uops can be done closed box. In most

of these tasks and all high complexity ones, the contribution of timed causal fanin analysis cannot be quantified by the computation time or memory usage with the method vs without, since without either automated or manual weakening the closed-box verification tasks are computationally infeasible. In our view, the best metric is the human effort required for the effort.

The largest positive impact was observed on the operations that are traditionally the most complicated and heavy to verify. For FMUL, the first higher-complexity operation, we implemented a new verification strategy that did not include the decomposition that the higher-precision proof requires. Note that FMUL is in fact FMA without an addend, which makes it a lighter task for verification, however any bug we would catch on FMUL, also exists on FMA. We continued with a new verification strategy for the FMA operations: closed-box input-to-output verification with a case-split on addend mantissa value. The effort of FMA verification bring-up was reduced from several quarters for a higher-precision 'big-FMA' in a standard Intel Core processor development project, to a couple of weeks.

For FDIV and FSQRT the effort reduction was also substantial. The proof was dramatically simplified, compared to the traditional multi-stage decomposed higher-precision proof. The FDIV and FSQRT proofs were completed in 6-8 weeks and provided confidence in design quality and arithmetic correctness. Like the FMA, effort for these verification tasks is usually measured in quarters of work.

Comparing then automated vs manual generation of weakening lists, the simple uop and FADD verification likely could have been carried out with manual analysis, as these tasks are not computationally challenging and a coarse analysis would suffice. On the other hand, a manual separation of the FMUL logic from the FMA, or the logic used vs not used by the different FMA cases, and especially the separation of the FP16 FDIV and FSQRT datapaths from the higher precision ones would likely have required an extraordinary human effort focusing on design minutiae.

The main advantages of the closed-box verification that enabled quick results were clear specification, ease of failure reproduction in dynamic validation with concrete source values, and the absence of any need to locate cut-points and define complicated side conditions. The first corner-case datapath bug was found in less than a week of work. Altogether, the FP16 verification initiative caught several extreme complexity bugs in just a few weeks of works at an early stage of the design project. This reduced the design cost of fixing the issues, and most importantly prevented them from escaping to the silicon implementation. Here are two examples:

1) An FMA16 uop multiplies two small positive normal numbers, produces a very small intermediate value, and adds the addend – the smallest normal negative. The mathematically accurate result is tiny, between the smallest normal negative and zero. Since Flush-To-Zero (FTZ) mode was set, the result ought to be zero, but the design returned the smallest normal negative.
2) FMA received three very specific normal numbers as inputs, and FTZ was set. We expected to produce the smallest normal number after rounding to nearest, but the result was flushed to zero. The specific inputs were:
   a: $s = 0$; $e = 00010$; $m = 1.0110000000$
   b: $s = 0$; $e = 01111$; $m = 1.0001011011$
   c: $s = 1$; $e = 00010$; $m = 1.1111111101$
   The intermediate result of the operation after it was normalized was: $s = 1$; $e = 0$; $m = 1.1111111111\mathbf{1}$ – one extra bit after the mantissa length, which is exactly at half-point for rounding, and therefore needs to round up. After rounding and normalizing we got a normal (non-tiny) number: $s = 1$; $e = 1$; $m = 1.0000000000$, that should not have been flushed to zero.

VI. Summary

Empirical experience has consistently shown that the timed causal fanin reduction algorithm is a key complexity reduction technique for practical symbolic simulation. It has also proven to be robust in face of design changes and over different design styles.

Timed causal fanin analysis was the primary enabler allowing us to verify all FP16 uops, including the most complex arithmetic operations, without decompositions. Closed-box verification greatly reduced the development effort of complex proofs, leading to fast detection of deep corner-case bugs in early stages of the project. Avoiding the use of decomposition has lowered the sensitivity to design implementation and made the verification collateral easily reusable for future projects.

References

[1] C. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods Syst. Des.*, vol. 6, no. 2, pp. 147–189, 1995.

[2] C.-J. Seger, R. Jones, J. O'Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.

[3] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), pp. 414–429, Springer, 2009.

[4] T. Melham, "Symbolic trajectory evaluation," in *Handbook of Model Checking* (E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds.), ch. 25, pp. 831–870, Springer International Publishing, 2018.

[5] R. Kaivola and J. O'Leary, "Verification of arithmetic and datapath circuits with symbolic simulation," in *Handbook of Computer Architecture* (A. Chattopadhyay, ed.), Springer, 2022.

[6] A. Gupta, M. V. A. KiranKumar, and R. Ghughal, "Formally verifying graphics FPU," in *FM 2014: Formal Methods* (C. Jones, P. Pihlajasaari, and J. Sun, eds.), pp. 673–687, Springer International Publishing, 2014.

[7] "Intel AVX512-FP16 Architecture Specification, June 2021, Revision 1.0." https://software.intel.com/content/www/us/en/develop/download/intel-avx512-fp16-architecture-specification.html, 2021.

[8] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers*, vol. C-35, pp. 677–691, August 1986.

[9] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *Proc. of 36th ACM/IEEE Design Automation Conference*, pp. 402–407, 1999.

[10] R. B. Jones, *Symbolic Simulation Methods for Industrial Formal Verification*. Springer, 2002.

[11] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, 1985. Note: Standard 754–1985.