Seqver : A Sequential Equivalence Verifier for Hardware Designs

Daher Kaiss
Formal Technologies Group
Intel Corporation
daher.kaiss@intel.com

Silvian Goldenberg
Digital Enterprise Group
Intel Corporation
silvian.goldenberg@intel.com

Ziyad Hanna
Formal Technologies Group
Intel Corporation
ziyad.hanna@intel.com

Zurab Khasidashvili
Formal Technologies Group
Intel Corporation
zurab.khasidashvili@intel.com

Abstract— **This paper addresses the problem of formal equivalence verification of hardware designs. Traditional methods and tools which perform equivalence verification are commonly based on *combinational* equivalence verification (CEV) methods. We however present a novel method and tool (Seqver) for performing *sequential* equivalence verification (SEV). The theory behind Seqver is based on the alignability theory, however in this paper we present a refinement to that theory: strong alignability, which introduces a concept of automatic model synchronization to the verification process. Automatic synchronization (reset) of sequential synchronous circuits is considered as one of the most challenging tasks in the domain of sequential equivalence verification. Earlier attempts were based on BDDs or classical reachability analysis, which by nature suffer from capacity limitations. Seqver is empowered with hybrid verification engines which combine state of the art SAT and BDD based engines for performing synchronization and verification. Seqver is widely used today in Intel for formally verifying leading next generation CPU designs.**

I. Introduction

Formal Equivalence Verification (FEV) is an essential CAD capability used to ensure correct implementation of VLSI designs, by verifying the equivalence between the implementation (for example schematics) and the specification (for example RTL) of the designs. This capability is also widely used when logical changes are done on a design, and the functional equivalence between the original and modified circuits must be guaranteed. We refer to the original circuit as the specification (spec) model and to the modified circuit as the implementation (imp) model. CEV technique requires complete correspondence between the storage elements of the models under comparison. Once this is guaranteed, a tautology-checking procedure (e.g. BDD [1] or SAT [2] based solver), can be employed to detect functional equivalence. In this case, the two compared designs are normally decomposed into combinational sub-circuits.

While most of the FEV tools that are available today in the EDA market are based on CEV, performing combinational verification has its drawbacks, which causes a vast impact on the CPU design process. First, the requirement to map every sequential in the spec model to a corresponding one in the imp model forces the designers to write detailed specifications. This low level of abstraction in the specification model has a negative impact on the quality of the specification: more lines of code traditionally form a trigger to error-prone design. In addition, detailed specification normally takes longer to validate since more simulation cycles are needed. Second, and not less important, designing a chip with the state matching requirement in mind has major impact on design convergence. It should be clear that during the chip design, RTL design involves a validation process, which ensures the correct functionality of the RTL. It is a time consuming process and thus designers try to reduce reiterating this process as much as possible. It is a fact that once a functional change in the implementation involves a sequential replacement, the RTL must be changed as well in order to meet the state matching requirement. Every change in the RTL triggers the validation process on the modified RTL. This process is found to be time consuming and may sometimes impact the schedule of the project. Third, state matching designs require mapping between the sequentials in both designs. Despite the fact that some methods for automatic mapping exit, most of this effort in custom designs is manual and thus time consuming. With the advent of Seqver, there is no need to map all the sequentials. The requirement now is to map as many signals as possible (not necessarily sequential elements). Sequential verification has drastically reduced the effort of mapping during FEV.

SEV overcomes the above limitations by enabling verification of two hardware designs which are at different or same abstraction levels, e.g. it enables verification of two designs with a different number or different placement of sequentials. This enables the RTL designers to write a smaller, more readable and easier to validate RTL, without compromising formal equivalence verification with the implementation. An example of an abstract implementation of a memory design vs. a detailed implementation is illustrated in Figure 1. In this design, the specification represents the memory with a flop based modeling, however to satisfy design restrictions, the circuit is implemented using a latch based model. In addition, notice that due to design restrictions, the designer had to stage the pre-decoding logic. In state matching designs, the specification should have matched the implementation in terms of sequentials, i.e., the same latches that appear in the implementation should appear in the specification. In this case, modifying the specification to mimic the schematic implementation has negative consequences in terms of RTL abstraction.

1-4244-9707-X/06/$20.00 ©2006 IEEE

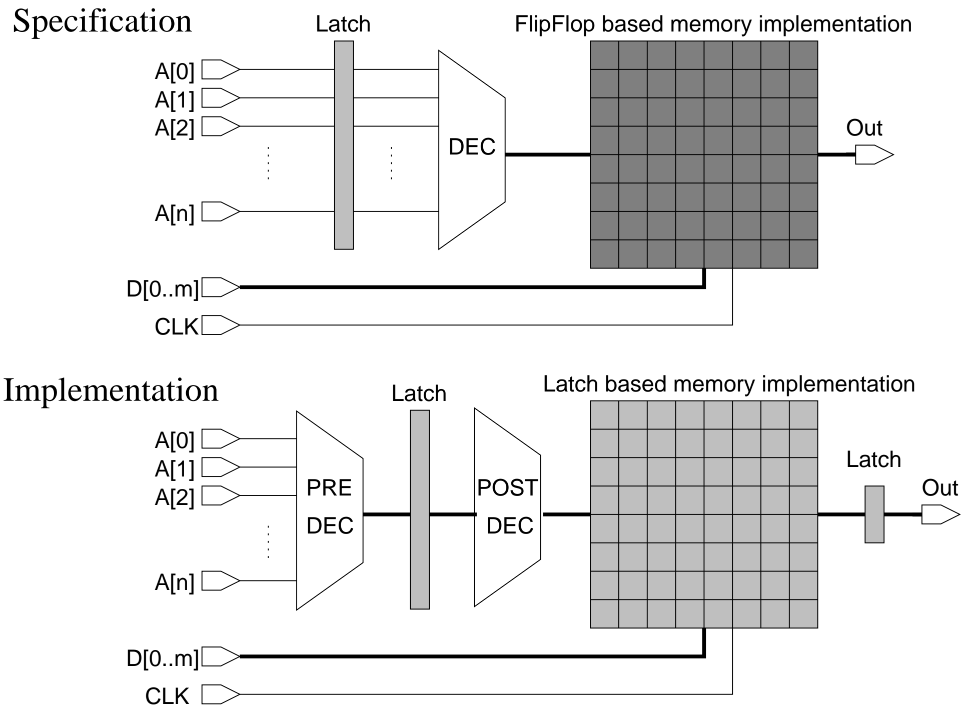

Fig. 1. Example of abstract vs. detailed memory implementation

The rest of this paper is organized as follows. In the next section, we present a framework for a precise, and at the same time flexible, representation of the circuit network. Section III gives a theoretical background to sequential verification and alignability verification. Section IV describes briefly and in high level the way we perform strong alignability verification. In section V, we describe why sequential verification is a step function over combinational verification in terms of accuracy and improvement of design productivity. Experimental results are reported in Section VI. Future work is discussed in section VII. We conclude in section VIII.

II. Preliminaries

A circuit design is modeled at the gate level in terms of combinational elements and storage elements. For simplicity we will assume that the storage element that we have is a device which transports its input to its output when the clock signal is high, and holds the output value when the clock signal is low. A *state* s of a circuit C is any one of the 2^n possible assignments of boolean values to the n storage elements of C. It should be assumed that the power-up, or starting state of a machine representing a circuit design cannot be predicted. Without restricting generality, we will assume that any circuit C has exactly one output, o, and a set of n storage elements $L_1 \cdots L_n$. We denote by C_s and C_i our specification and implementation circuits with outputs o_s, o_i, and with n and m storage elements (or latches) $L_1^s \cdots L_n^s$ and $L_1^i \cdots L_m^i$, respectively. We assume that both circuits have the same set of inputs. We denote by $C_{\Leftrightarrow}$ the combined circuit of C_s and C_i (the *product machine* [11]) with shared inputs and equivalence of $o = (o_s \Leftrightarrow o_i)$ as the output.

We consider *ternary* modeling of circuit node values. A value could be one of the *binary* values, T or F, or an *undefined* value, $\perp$ (elsewhere also denoted by X). Given a binary input vector sequence π, $n(s, \pi)$ will denote the value of node n in a circuit C after 3-valued simulation of C with π, starting at state s. Similarly, $C(s, \pi)$ denotes the (ternary) state into which π brings C, from state s. The *unknown state* of C is the state in which all storage elements have the undefined value X. A *binary state* of a circuit C is a state in which all state elements have binary values.

A circuit C can be represented by a collection of *next-state functions* (NSFs) of the latches as well as of the output, where a NSF is a function of current and next-state values of inputs and latches. For example, consider the circuit C which is illustrated in figure 2. It consists of four inputs a, b, c and clk, one latch l, and and OR gate (o) which is the output of the circuit. We annotate the current state value of a variable "v" using v and the next state value of the same variable using v'. This way, the next state function of the output o is $l' \vee c'$, while the NSF of the latch l is $(clk' \wedge a' \wedge b') \vee (\neg clk' \wedge l)$. Available convenient representations for next state functions can be BDDs or boolean expressions (simple graph data structures for representing propositional logic, where nodes of the graph represent binary operation $\wedge, \vee$, with an annotation whether a variable is negated or not, and variables appear as leafs). We adopted boolean expressions in our work since uniqueness of BDDs in not needed.

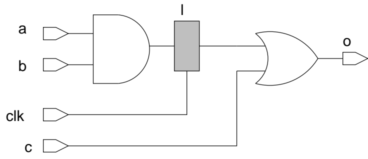

Fig. 2. Example of latch and output functions

III. SEV AND ALIGNABILITY THEORY

Moving from combinational to sequential verification required major theoretical and methodological changes in the FEV tools. Combinational verification was considered an easier task. The two compared designs were decomposed into slices via cutting the designs on the outputs of the sequentials which were mapped (traditionally by the designers). Equivalence verification was performed separately on each of the slices by assuming that the mapped sequentials are equivalent in the current state, and verifying that they are also equivalent in the next state. The task of performing sequential verification is much more complex, since it is not obvious from which states the verification should start. This problem of sequential hardware synchronization (reset) is considered as one of the most challenging tasks in the domain of formal sequential equivalence verification. To read more about comparison between CEV and SEV, please refer to [15].

Some methods for hardware equivalence verification are based on the theory of finite state machines [5], where two finite automata are defined to be equivalent if, starting from a pre-defined initial state, they accept the same input sequence, i.e. the sequence leads to a set of predefined final accepting states. In this case, the two compared designs are modeled using two automata or two FSMs and the equivalence of the designs is shown by proving equivalence of the corresponding FSMs achieved via the equivalence of their corresponding automata. Classic BDD-based model checking and verification algorithms require a reset state [6]. The same is true for well known SAT-based model checking algorithms such as BMC [7], [8] or the induction method [9]. However, in practice, the starting state of the compared circuits is not always available as the power-up state of a hardware design cannot be predicted or controlled. Therefore, a technique for automatically computing the reset state is needed in order to perform equivalence verification. Alignability equivalence [10] is a convenient concept of hardware equivalence verification. In this section, we recall concepts related to it.

Definition 3.1: An *initializing sequence* of C is a sequence of binary inputs which, when applied to the unknown state of C, brings C into a binary state.

Definition 3.2: State (s_1, s_2) of the combined circuit $C_\Leftrightarrow$ is an equal (similarly differ) state if $o_1 = o_2$ (respectively $o_1 \neq o_2$) at (s_1, s_2).

Definition 3.3: State (s_1, s_2) of the combined circuit $C_\Leftrightarrow$ is an equivalent state (denoted by $s_1 \simeq s_2$) if for any input sequence π, $o_1(s_1, \pi) = o_2(s_2, \pi)$. States s_1 and s_2 are then called equivalent states of C_1 and C_2. [1]

Definition 3.4: ([10])

1) A binary input sequence π is an *aligning* sequence for a combined state (s_1, s_2) of $C_\Leftrightarrow$ if it brings $C_\Leftrightarrow$ from state (s_1, s_2) into an equivalent state.
2) Circuits C_1 and C_2 are *alignable*, written $C_1 \cong_{aln} C_2$, if every state of $C_\Leftrightarrow$ has an aligning sequence

It is shown in [10] that $C_1 \cong_{aln} C_2$ iff there is a sequence, called a *universal aligning sequence*, that aligns any state of $C_\Leftrightarrow$. When the two circuits coincide $C_1 = C_2 = C$, then following [4] we speak of self-alignability of C. It is easy to see that C is self-alignable iff it is *weakly synchronizable*, as defined in [13]:

Definition 3.5: A weak synchronizing sequence (ws-sequence for short) of a circuit C is an input vector sequence that brings C from any binary state to a subset of equivalent states $\{s_1, \cdots, s_m\}$, called ws-states of C.

Theorem 3.6: Alignment Theorem: [10] Circuits C_1 and C_2 are alignable if and only if each circuit is weakly synchronizable and there is an equivalent pair $s_1 \simeq s_2$ of states in C_1 and C_2. The concatenation of ws-sequences of C_1 and C_2 is a ws-sequence for both of them and it weakly synchronizes C_1 and C_2 into equivalent ws-states (when C_1 and C_2 are alignable).

We are now going to define a more restrictive version of alignability, which is called strong-alignability. In this definition, we are restricting the weakly-synchronizing sequence to be a synchronization sequence [12]:

Definition 3.7: A reset or synchronization sequence π_r brings C from *any binary* state to a unique state s_r, called a *reset* or *synchronization* state. A circuit C is synchronizable if it has a synchronization sequence.

Note that the set of ws-states and the set of synchronization states are closed under state transition. It is clear that every synchronization sequence is also a weak synchronization sequence. Therefore we can define strong-alignability as follows:

Definition 3.8: Strong-Alignability: Circuits C_1 and C_2 are strongly alignable if each circuit is synchronizable and there is an equivalent pair (s_1, s_2) of states in C_1 and C_2.

The concatenation of synchronizing sequences of C_1 and C_2 is a synchronizing sequence for both of them and it synchronizes C_1 and C_2 into equivalent states when C_1 and C_2 are strongly alignable. Let circuit $C_\Leftrightarrow$ be with output o, and assume that $\mathcal{S}$ is the set of states of $C_\Leftrightarrow$; then:

$$C_1 \cong_{aln} C_2 \Leftrightarrow \exists \pi . \forall s \in \mathcal{S}. \begin{pmatrix} s_r = C_\Leftrightarrow(s, \pi) \ \wedge \\ \forall \gamma . o(s_r, \gamma) = \mathrm{T} \end{pmatrix} \quad (1)$$

Due to the fact that most successful SAT solvers are propositional logic solvers, there is a need to separate the computation of the initial state(s) from the equivalence verification. Equivalence verification of two circuits is done in two stages: first we compute a sequence which guarantees that when applied to a circuit $C_\Leftrightarrow$, it brings the circuit to an *equal*

[1] The concepts of equal-states and differ-states should not be mixed with equivalent and inequivalent states. In this definition, all states are binary.

state (to differentiate from an *equivalent state* since at this stage we don't know whether the equal state is an equivalent one or not). Second, we perform equivalence verification from the state(s) of the first stage. If the verification passes, then the models are declared alignable. The reasons for choosing strong alignability instead of the regular alignability theory for performing equivalence verification were driven by practical reasons. Assume that the second stage fails with a counter example. This failure can stem from two possible reasons: (1) there is no sequence that aligns the two circuits, (2) this is a real counter example that needs to be debugged. Since it is not practical to require from designers to debug counter examples which later on will be root caused to a problematic initialization, we decided to be consistent in the outputs of our verification tool. This method requires the designs to be resettable and in case they are not, we do not proceed to the second stage. However, when the method succeeds in the first stage, it is guaranteed that any counter example which is found in the second stage (equivalence verification) is a real one.

IV. Performing Strong Alignability

It goes without saying that without decomposition, performing alignability verification on full-chip designs is beyond the capacity of current verification tools. In [14], a compositional alignability verification framework was proposed. The main idea is that provided the circuits C_s, C_i are both weakly synchronizable, their alignability can be proved by decomposing them into a stable decomposition with small subcircuits, and proving alignability of each corresponding sub-circuit pair of $C_{\Leftrightarrow}$. For this work, it is enough to know that input constraints are imposed on each corresponding subcircuit pair, say (D_s, D_i); these constraints model the environments of D_s and D_i in C_s and C_i, respectively; and for compositional equivalence verification of C_s and C_i, it is necessary that the synchronizing sequence of D_s and D_i must satisfy these constraints. In practice, these are normally constraints relating circuit signal values in the same time frame.

Given two circuits C_s and C_i, SAT-based strong alignability verification is performed in two stages: (1) computation of reset sequence for both circuits (which by itself produces the reset state) and (2) performing SAT-based equivalence verification from the computed reset state.

During the synchronization stage, we assume that all state elements L are uninitialized (with X value), and using a SAT solver we look for a sequence that brings the circuit to a state where all state elements have binary values. More formally, for every circuit, we try to find a sequence that satisfies the following:

$$\prod_{L \in \mathcal{L}} (L[k] = T) \vee (L[k] = F) \tag{2}$$

where k is the length of the sequence, and $L[k]$ represents that value of the sequential element L after simulating the sequence. Seqver's synchronization algorithm is in the process of patent filing [16].

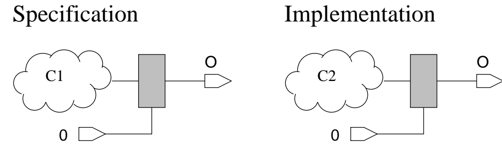

Fig. 3. Non resettable designs

Once the reset state is computed, we mainly use commonly used model checking algorithms (SAT [9] or BDDs) in order to perform the equivalence verification.

V. Seqver as a Step Function over CEV

In this section, we give an overview of major advantages in terms of improving design productivity that sequential verification provides. We will discuss the following aspects: (1) impact of introducing the reset sequence as part of the verification, (2) impact on the number of needed FEV properties and (3) impact on property verification.

A. Synchronization as a contributor to FEV accuracy

As mentioned earlier, combinational verification was all about verifying whether the combinational slices outputs are equal in the next state, assuming they are equal in the current state. One drawback of this method is that combinational verification doesn't check whether the combinational cones are synchronizable at all. If such a problem exists, revealing it is pushed to full chip validation which is not a complete method, and traditionally happens only at late stages of the project. Consider figure 3 for an example.

Assume that the combinational clouds $C1$ and $C2$ are not equivalent. In combinational verification, FEV would confirm that these designs are equivalent, as the assumption is that they are equivalent in the current state and since the clock in both designs is constant 0, the value of the latches in the next state will be the retained value. In the above example, SEV would detect the unresettability of these designs before proceeding to the equivalence verification. For such designs, stuck-at-false phenomena are detected early in the design.

In figure 4 we show another example where sequential verification gives accurate results compared to combinational verification. Again, assume that the combinational clouds $C1$ and $C2$ are not equivalent, and none of them is equivalent to 0. In the combinational case, the verification would fail as the counter example would be that the values of the sequentials in the current state are 0's while in the next state, $C1$ and $C2$ will be inverse. This enables the 1 at the data entry of the sequential to pass in one model, while in the other model, the 0 will be retained . This counter example is not really correct as designers would assume that some reset is happening in the design that will propagate the data (1 in both designs). Notice that sequential verification would confirm that the designs are equivalent.

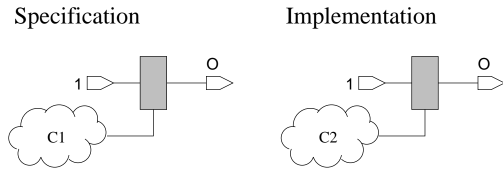

Fig. 4. Equivalent models with different clock scheme designs

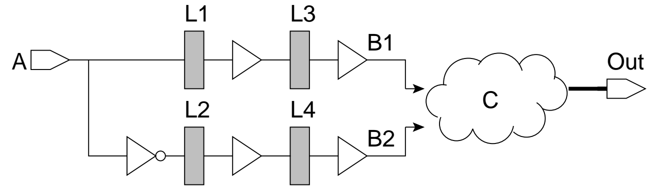

Fig. 5. Example for need of functions

B. Impact on the number of needed FEV properties

Designing state matching designs is not that easy also in terms of handling properties. Recall that in combinational verification, one needs to map all the sequentials in both designs. This mapping introduces a one-to-one correspondence between the sequentials in both designs. These map points define boundaries of the verification slices as every sequential turns to be a 'cut point'. These cut points may have negative impact on the accuracy of FEV as they can generate many false negatives. To clarify this, consider the circuit in figure 5. If one would compare this design to another state matching design, mapping of the sequentials $L1, L2, L3, L4$ is needed. This mapping will introduce 5 verification slices at $L1, L2, L3, L4$ and Out. Consider the verification of the output Out. Its slice is bounded by the sequentials $L3$ and $L4$, however notice that during the verification of Out, there is no way to know that $L3$ and $L4$ are inverse. This information is available only from the logic that drives $L1$ and $L2$. In this case, equivalence verification requires adding an inverse *property* (or an *assumption*) in the RTL between $L3$ and $L4$. For completeness, such properties should be validated in order to avoid any possible verification holes. Notice that in combinational verification, the inverse property might be needed on $L1$ and $L2$ in order to verify $L3$ and $L4$ as well.

Seqver helps reduce this overhead by reducing the number of needed properties. With the advent of sequential verification, the cone of Out can be expanded up to the inputs (A), and thus, there is no need to add properties at all in this example. However, in case a complexity issue is encountered (e.g., due to a complex logic at cloud C, or a deep sequential cone up to the primary inputs), properties could be added, but this is always much less than the needed number of properties in state matching designs. For example, one of the functional blocks that was with state matching encoding and included hundreds of properties was rewritten in an abstract way, while the number of the needed FEV properties that were added was reduced from hundreds to only 50!

C. Impact on the completeness of the property verification process

Recall that FEV properties are added to the design in order to 'compensate' for missing information that was lost as part of the mapping process. Seqver helps reduce the need to map all the sequentials, and thus reduces the number of added properties. This does not mean we do not need FEV properties anymore, however.

Recall the same example of figure 5. Assume that the combinational cloud C is too complex and we have to cut now at signals $B1$ and $B2$. Recall also that an inverse property between $B1$ and $B2$ needs to be added to the design in order to make the verification pass. We'll call it $P1$. In combinational verification, verifying the property $P1$ was limited to combinational logic only and thus the verification of $P1$ could not be performed without adding an extra inverse property between $L3$ and $L4$. Assume that mistakenly, instead of adding an inverse property between $L3$ and $L4$, the designer added an inverse property between $B1$ and $L4$. We'll call it $P2$. In this case, combinational verification of $P1$ and $P2$ passes as each property will pass depending on the other (as the slice for both of them will be bounded by $L3$ and $L4$). This behavior might lead to a verification hole when the logic driving $L3$ and $L4$ doesn't satisfy the property. The above problem is also known as the problem of 'cyclic dependency' between properties when a closure of properties cause one property to pass based on the other properties.

Methods for identifying cyclic dependencies between properties are time-consuming due to the fact that we have lots of properties in the designs. The purpose of Seqver is to solve the above issues. The fact that Seqver can perform verification beyond the combinational cloud enables verifying FEV properties while considering *only* relevant properties specified on the first mapped layer of sequentials. In the above example, the slice of $P1$ will be bounded by $L1$ and $L2$, in case they are mapped, and by A if not. Notice that during this verification, $P2$ won't be used as it is not specified on the boundary of the slice (in both cases). This method enables breaking the above cyclic dependency between the properties.

VI. Results

Table I illustrates the difference between circuits in terms of abstraction (measured by a different number of sequential elements). It also shows an overall mapping effort that was saved when moving to non state-matching designs. This table should be read as follows: the second column represents the number of the outputs in the compared circuits. Columns 3 and 6 represent the number of the sequential elements (Lats) in the specification and implementation models. Columns 4 and 7 represent the number of mapped sequential elements in the specification and implementation. Notice that if a state-matching specification was written, the designers should have mapped all the sequential elements which appear in column 6. Column 7 comes to show exactly the mapping effort which is

Ckt.	Outs	Specification Lats	Mapped	%	Implementation Lats	Mapped	%	Abstraction	CPU (Sec.)
$C1$	29	125	59	47%	195	59	30%	64%	85
$C2$	58	635	634	100%	1212	642	53%	52%	62
$C3$	65	271	129	48%	438	133	30%	62%	642
$C4$	84	2635	2635	100%	4185	3507	84%	63%	817
$C5$	114	2902	2787	96%	5454	2813	52%	53%	1273
$C6$	151	232	232	100%	323	240	74%	72%	21
$C7$	196	235	235	100%	464	244	53%	51%	274
$C8$	205	1129	1105	98%	1354	1338	99%	83%	35
$C9$	208	820	698	85%	1020	429	42%	80%	739
$C10$	221	32	0	0%	64	0	0%	50%	31
$C11$	232	441	95	22%	554	95	17%	80%	1418
$C12$	259	675	669	99%	1167	689	59%	58%	263
$C13$	399	67	57	85%	126	57	45%	53%	54
$C14$	848	1370	1085	79%	1808	1085	60%	76%	4471
$C15$	1040	1627	964	59%	2055	963	47%	79%	600

TABLE I
ABSTRACTION AND MAPPING SAVINGS

saved. For example, for circuit $C1$, only 30% of the sequential elements were mapped in the implementation (59 out of 195), saving 70% of the mapping effort. Column 9 (Abstraction), represents the ratio between the number of the sequential elements in the specification compared to those in the implementation. For example, for circuit $C7$, the number of the state elements in the implementation is almost twice the number in the specification. This idicates that the implementation is much more detailed than the specification. Column 10 gives quantitive numbers about the overall runtime of the tool on these designs. Our experiments were performed on 64bit Intel machines with 8G memory.

Table II represents data about the size of the compared slices for most challenging instances of the verification. For the instances that include more than 100 variables (inputs + latches), BDD based model checking techniques weren't successful.

Ckt.	Specification Gates	Lats	Inps	Implementation Gates	Lats	Inps
$O1$	5877	27	246	194	28	246
$O2$	5281	533	195	4827	166	218
$O3$	553	129	148	1105	147	152
$O4$	479	129	148	1062	140	148
$O5$	446	153	92	1006	152	152
$O6$	308	48	51	278	68	51
$O7$	317	55	46	292	45	50
$O8$	299	45	44	252	61	44
$O9$	215	11	116	215	13	115
$O10$	168	28	86	603	37	99

TABLE II
REPRESENTATIVE SLICE SIZES

VII. FUTURE WORK

One of the big challenges that remains when performing sequential verification is the question of how to handle complex slices. For some designs, Seqver can easily handle the verification from primary outputs to primary inputs, without any need to decompose the designs. However, in most of the cases, the slices are too big and they exceed the tool limits. Some methodological solutions are proposed to the designers in order to advise how to decompose the design. For this, more automatic mapping tools are being developed. Another big challenge is debugging the non-resettable designs. This turned out to be a tough task as visualization tools need to be developed to help find the root-cause. Today, more methodological solutions are given like initializing the sequential element with constant 0 and checking which continue to retain the same constant value. For most of the cases, initialization issues are caused by an enable logic which is stuck at constant 0. Another challenge to research is the impact of sequential verification on other design domains like power.

VIII. SUMMARY

We have introduced Seqver, an Intel formal equivalence verification tool which enables formally verifying abstract specification vs. a detailed implementation. Seqver is based on a solid theoretical background based on the alignability theory with slight modifications to meet practical design requirements. Seqver also brings completeness to the traditional combinational equivalence verification tools. Representative results from leading next generation Intel CPU designs show a great amount of abstraction between the specification and the implementation with proven reduction of manual mapping effort that shortens the CPU development life cycle.

REFERENCES

[1] R.E. Bryant Graph-based algorithms for Boolean function manipulation, IEEE Trans.Computers, C-35(8), 1986.
[2] Davis,M., G.Logemann, D. Loveland, A machine program for theorem-proving, CACM 5(7), 1962.

[3] C. Pixley, S.-W. Jeong, G.D. Hachtel. *Exact calculation of synchronizing sequences based on binary decision diagrams*, IEEE transactions on Computer-Aided Design, vol. 13, 1994.

[4] A. Rosenmnann and Z. Hanna. *Alignability equivalence of synchronous sequential circuits*, HLDVT, 2002.

[5] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison-Wesley, 1979.

[6] O. Coudert, C. Berthet, J.C. Madre. *Verification of synchronous sequential machines based on symbolic execution*, Workshop of Automatic Verification Methods for Finite State Systems, 1989.

[7] A. Biere, A. Cimatti, E. Clarke. *Symbolic model checking without BDDs*, Tools and Algorithms for the Construction and Analysis of Systems, 1999.

[8] A. Biere, A. Cimatti, E. Clarke, M. Fujita. Y. Zhu. *Symbolic model checking using SAT procedures instead of BDDs*, DAC 1999.

[9] M. Sheeran, S. Singh, G. Stålmarck. *Checking safety properties using induction and a SAT-solver*, FMCAD, 2000.

[10] C. Pixley. *A theory and implementation of sequential hardware equivalence*, IEEE transactions on CAD, 1992.

[11] G.D. Hachtel, F. Somenzi. *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1998.

[12] Z. Kohavi. *Switching and Finite Automata Theory*, McGraw-Hill, 1978.

[13] I. Pomerance, S. M. Reddy. *On removing redundancies from synchronous sequential circuits with synchronizing sequences*, IEEE Trans. Comput., pp.20-32, 1996.

[14] Z. Khasidashvili, M. Skaba, D. Kaiss and Z. Hanna. *Theoretical Framework for Compositional Sequential Hardware Equivalence Verification in Presence of Design Constraints*, Proceedings of the International Conference on Computer Aided Design, IEEE, 2004.

[15] Z. Khasidashvili, M. Skaba, D. Kaiss and Z. Hanna. *Post-reboot Equivalence and Compositional Verification of Hardware*, to appear in FMCAD 2006.

[16] D. Kaiss, M. Skaba, Z. Hanna and Z. Khasidashvili. *SAT based alignability method for sequential hardware verification*, in preparation.