Formal Verification of Digital Systems, from ASICs to HW/SW Codesign - a Pragmatic Approach

Roger B. Hughes
VP Marketing & Technical Support,
Abstract Hardware,
47211 Lakeview Boulevard,
Fremont, CA 94538-6530
rhughes@wcdf.viewlogic.com
WWW. http://www.ahl.co.uk VP

In order to verify digital systems one has to consider the particular verification approach to be adopted and what is meant to be achieved by the application of the method chosen. Over the past few years thee has been increasing use of formal verification both from a specification standpoint and from an equivalency-checking standpoint. Basically these techniques amount to "a priori" and "a posteriori" verification. In hte "a priori" class there are very few tools, LAMBDA being one of these. The LAMBDA system is based on a theorem-prover core and is interesting owing to its ability to transform an initial specification into a lower-level (typically RTL) description of a design. The design partitioning is such that at each stage the partial implementation is formally proven to meet its specification as an *integral* part of the design process, i.e. verification takes place "in situ". The assumption that is made being that the specification is what the engineer actually wants to build. Such a technique allows the engineer to reason about hardware and software as equal entities in the design process, only thinking of partitioning functionality until the functions have been broken down to the stage where they may eeither be implemented in hardware (as RTL for a synthesis process) or software (for a compilation process). It is possible in both cases to go down to either gate-level or assembly code but it is recommended that the LAMBDA approach only be used to get to synthesisable RTL and or readily compilable software.

One of the difficulties of theorem proving is that one needs a language with a formal semantics, not just a formal grammar in which to specify the system which you wish to build. The L2 (Lambda Logic) description language, based on the functional programming language ML with some additional mathematical constructs derived fom HOL (Higher Order Logic) is one of very few such languages. It is clearly a succinct way to abstractly specify digital systems' functionality but most engineers are still reluctant to learn a new language. In fact, it is more than that, it requires a paradigm shift in that a new methodology must be adopted. One cannot "reverse engineer" specifications from quickly "hacked" implementations. As a way to assist the engineering community to bridge this mathematical void, a tool called TimeWarp was developed, based on the theorem proving core of LAMBDA, to totally automatically transform a graphical (or textual) specification of data-flow functionality into a synthesisable piece of RTL Verilog or VHDL. All the complexities of partial and full loop-pipelining

are automatically handled and the engineer gets a graphical display of resource allocation and scheduling information. The hraphical display is interactive allowing the engineer to *alter* the automatically generated result, thereby allowing his/her design expertise to optimise the design further. Naturally, to assist in this process the engineer is provided with information such as throughput, maximum estimated frequency, latency etc. of the design. Whilst TomeWarp does not have the same capability or generality as LAMBDA, it is very easy to use (unlike LAMBDA) and is an important step on the learning curve which all engineers must emabark on now in order to cope with the complexity of designs tomorrow.

At the other end of the verification spectrum lies equivalence-checking. The idea here being to show if two existing designs are equivalent. Equivalence here means functional equivalence, i.e. are the output functions the same in both designs. Theorem provers can also be used here but it is both awkward to enter the design descriptions into rules representing the design state and awkward to use the theorem prover to show equivalence. Quite often, experienced mathematical skill is essential in doing even the most trivial of equivalence proofs. Thus an alternative technique is used. One technique is to represent the output functions as boolean equations (this assumes that the state mappings in both designs, viz. number of registers, is the same). By reducing the boolean equations to a simpler form, they can be compared to see if they are the same. Various comparison methods are used. Another technique is to use Binary Decision Diagrams (BDDs), preferably Reduced Ordered Binary Decision Diagrams (ROBDDs). There are advantages to both of these techniques.

In the equational approach, the advantage is one of speed. This is because it is fairly straightforward to show if two boolean equations are equivalent or not. Basically, there is a choice between reducing the equations to a canonical form, thereby permitting an extremely rapid test of equivalence or a partial reduction, making the comparison process a little more arduous. It is the reduction process that is often the most time-consuming aspect of such equational equivalence checking. As such, the majority of practical tools do not usually seek to reduce down to a canonical form but just a simple set of reductions to minimise the time taken. Of course, this approach can only be simple (and in practice only works for) designs which have the same number of state-bits and identical state-encodings. That is, the state information in the design is effectively ignored, only the combinational logic between register stages is checked for equivalence. Clearly, this demands that the registers in both designs are used to store the same information. Not all designs are of this type, but a large subset of desgns are. For example, a gate-level design compared with a design with a few hand tweaks is typically of this class. An RTL design fed through a non-optimising (i.e. states are not optimised) synthesiser and compared with its gate-level will also fall into this class. Designs that can be handled by equational equivalence checking are typically large, e.g. 500,000 gates is quite achievable in one go. The CheckOff-S product from Abstract achieves this in a matter of 2 minutes. Clearly such a technique is fast, much faster than simulation, yet exhaustive, which simulation typically isn't.

The ROBDD approach also has advantages in that it can compare designs very quickly once the BDD model (actually an FSM (Finite State Machine) view of the design built using BDDs) of the designs have been built, or "compiled". Abstract Hardware's equivalence checker is called CheckOff-E and is based on the same ROBDD technology as used in CheckOff-M, the model checking product. In the case, as in equational equivalence checking, where designs have the same number of states, the comparison process is exceedingly quick owing to the fact that time was taken compiling the design respresentation down to a canonical form. A comparison of such canonical forms is very fast, requiring just a reordering of the nodes followed by a simple graph isomorphism check. Computers are very efficient at such checks, e.g. a 50,000 gate circuit may be fully checked in just 50 seconds. However, one must add that the compilation process for each of the designs can easily take 10 minutes. However, BDD technology has one other major advantage over equational equivalence checking. It is possible to check the equivalence of designs that are not structurally equivalent, but behaviourally equivalent in a sequential sense, i.e. the outputs on a clock-by-clock basis correspond in both designs, regardless of the number of internal states, or state-mappings in each of the designs. This type of equivalence checking is extremely important when an engineer recodes a piece of RTL or if optimising compilers are used to remove state-bits or if the engineer uses one-hot encoding. There are many such cases. Only ROBDD based technology can allow such equivalencies to be shown, equational equiovalence checking will simply state that the two design are not equivalent, despite the fact that they are. Another advantage of compiling down to an intermediate form is that it facilitates the easy comparison of designs written in any level of language and also between different languages, e.g. RTL VHDL vs. GTL Verilog or RTL Verilog vs. EDIF netlist. All combinations of language and level of language used in comaprison being equally possible. What happens if the two designs are not equivalent? In this case, a complete simulation test bench is written, in either VHDL or Verilog, to illustrate the shortest path to find a difference between the two designs. The engineer can feed this test-bench into his favorite simulator to further narrow down the cause of the problem.

Another approach used quite recently is that of model-checking or property-checking. Here a design, quite often the RTL that is supposed to be the golden model, is taken and essential properties that it is supposed to satisfy are proven about it, e.g. absence of deadlock, a bus arbiter responding within a certain time, a traffic light controller satsfying a safety property that both lights are not green at the same time. etc. Model checking can also be used as a wekaer form of equivalence checking, nameely algorithmic equivalence checking. For example, consider two RTL Verilog descriptions of a design for a multiplier. The first multiplier is a booth implementation, the second is a shift & add implementation. Clearly, both multipliers are doing the same thing but: they are not structurally equivalent, owing to the different numbers of state bits and state encodings; they are not sequentially equivalent, owing to the fact that the outputs do not

correspond on a clock-by-clock basis, the time taken being data-dependent and not a simple fixed offset from one design to the other. Yet both are doing the same basic computation, are they not? How do we show this? If we use model-checking, we can define a property which compares the outputs of both designs, independent of time taken, when the "done" flag in both designs is raised. The model-checker can thus show that for all combinations of inputs over all time that the designs are equivalent. Abstract Hardware's model-checking technology is called CheckOff-M.

It is envisaged that the ideal technique is to marry the technologies of theorem-proving and BDD based model-checking. The ability to handle design complexity of the former is well known, yet the automation provided by the latter is regarded as essential. Such a marriage can work in two ways: firstly, to use theorem-proving technology to keep an eye on system obligations that have to be proven during the model-checking process and secondly, to use model-checking to prove awkward lemmas suggested by a theorem based view of design. The difficulty is toi arrange the external appearance of such tools such that it does not appear totally alien to the practising engineer. The author believes that education is still going to be required, but new engineers with these skills are now beginning to be produced by many academic institutions. Senior management need to make themselves aware of the capabilities of these new technologies.

During the talk various slides will be shown to illustrate what can be done with each of the technologies and how best to position them into an existing commercial design-flow. With a suitable use of these technologies it has already been shown that the time-to-market can be reduced by a factor of two to five fold. This is a significant reduction in overall cost as well as ensuring a market window. As more companies adopt this type of technology, the others will be forced to follow suite if they are to meet their return-on-investment in new product development, or their competitors will fill the market gap before them. Formal verification can give the designer the freedom to explore the design space, to find better, more efficient designs in complete safety. Formal verification of properties or equivalence does give exhaustive coverage in far less time than non-exhaustive simulation. This does not mean an end to simulation, but a large reduction in what is required. e.g. it is easier to take a 128-bit adder and give it the numbers 4 and 5 and see if the answer is 9. If the answer is 10, there is clearly no need to formally verify the operation of the adder! Thus simulation will still be used as a fast "sanity-check" on designs and will be used to illustrate when two designs are not equivalent.