An Introduction To Property Checkers For Functional Verification

By Andrew Jones and Jeremy Sonander, Saros Technology

Introduction

The increasing complexity of system-on-a-chip and ASIC designs has caused an ever-widening gap between what can be designed and what can be. It is estimated that between 50-70% of the time required to design a complex IC is spent in verifying that the functionality of the system is correct. Bugs in a design are least expensive to fix just after they are created. At this stage the design is still fresh in the designer's mind and other parts of the project or other design team members are unaffected. Bugs are at least an order of magnitude more expensive to fix during system integration. In this phase it takes more time and people to analyse the cause, regression tests must be rerun, and the entire group may be delayed. These challenges are giving rise to some exciting new tools and approaches in Verification techniques.

The Properties Approach To Verification

In some ways Property Checking is similar to traditional RTL simulation in that it tests the functionality of your design. The most fundamental difference with property checking is that is implemented using mathematical techniques. This approach does not require a vector set, and therefore does not require a test bench to be written to exercise the design. This saves time straight away. However the main advantage of this approach is that typically, a vector set is written to only exercise the behaviour of the design in its expected mode of operation. In reality the input to a block often deviates from the designer’s initial expectations, and the design is then in untested territory. There are of course practical reasons for this: it is hard to “expect the unexpected”. Properties are able to test the design in all possible modes of operation and therefore have the ability to isolate bugs and undesired behavior that a designer might not have thought to test.

An Analogy With More Familiar Technology

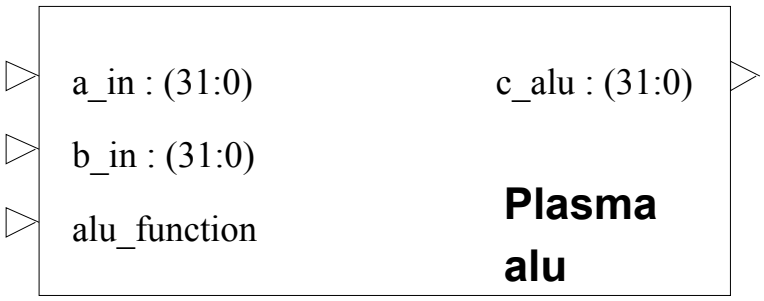
Of course, static tools are not that new to us and we can compare this approach with something much more familiar, namely “Static Timing Analysis”. A timing simulation cannot possibly provide certainty we have hit the slowest set of vectors. Take a ripple carry adder. 1+256 will be much faster than 1+255. If we don’t pick the worst-case set of vectors, we wont get a worst case result. It’s therefore going to be impossible on a large design to learn much with timing simulation. In this example, the property our design should have is “Run at 100MHz”. We know it’s not necessary to exhaustively simulate the design to find if it has this property. The static timing tool will automatically isolate any paths that are too slow. So wouldn’t it be great if we could do the same thing for the design’s behaviour?

Static Functional Verification Re-Visited

Property Checkers are the "Functional Equivalent of Static Timing Analysis. You describe things your design should always do, and things that your design should never do by writing a specification in fragments called properties. The tool will then either confirm your design works and can never fail, or alternatively, provide an example of your design failing.

ALU, Combinatorial Example

So let's look at a practical example. We have two numbers 'a_in' and 'b_in' as well as a function to run. From this, we will then produce a single output 'c_alu'. Also, suppose that it is absolutely critical that this works correctly in all modes of operation.



There are, in total, 17 alu_function modes. If we were to exercise traditional simulation techniques, we would select each of the 17 instructions, and sweep through the entire range of 'a_in' and 'b_in' using counters and checking as we go. This would require approximately 70 lines of Test Bench.

Let us estimate the run time for this process. Typically, we know that a single computer could run 100,000 vectors per second. For a large task like this, we know one company that has created a server farm, where they link up to 5000 workstations together. So potentially, we could run at an impressive 500 million vectors per second.

If we look at each instruction such as add, this is going to require 2^{64} Vectors for a complete proof. Which gives us the following run time:

- **$2^{64}/5x10^{8}$ seconds = 1,170 years per instruction!**
- **Or around 19,890 years to completely verify using simulation!**

Clearly, this is not going to be a practical solution even for our large server farm. How about using a static approach using properties? The complete property to verify the add instruction is:

```
alu_function == alu_add => c_alu == a_in+b_in;
```

This is certainly shorter than our 50 lines of Test Bench. What about run time?

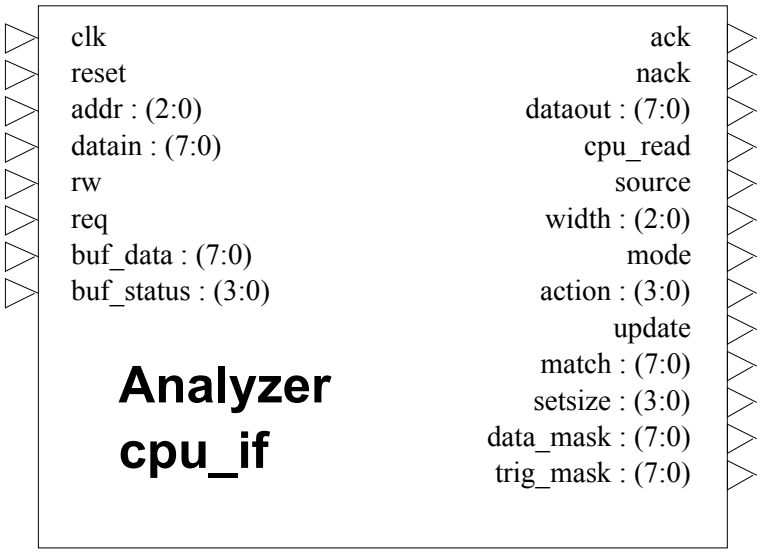
- **We were able to exhaustively verify all 17 properties on a single 850MHZ Pentium III running Windows 2000 in approximately 2 seconds!**

- **The property checking tool we used to exercise this design is known as Solidify® from a U.S. company called Averant**

Quite impressive! Lets now look at an example with a different set of criteria.

CPU Interface, Sequential Example

This is a CPU interface from a Logic Analyser, and performs read and write applications on a CPU Bus. This implements a time out releasing the bus if a device does not respond. We want to verify that the bus cannot lock up, and therefore that each request will get an 'ack' (acknowledge) or 'nack' (no-acknowledge).



Now the task here is not waiting for the vectors to complete, but determining what those vectors should be. We have many inputs here coupled with lots of internal state. One of the beauties of using properties is that everything we want tested is not necessarily mentioned in the property itself. The more exhaustive the test, the more simple the property. The solution as a property is:

req && forever (!reset) => within5 (ack|nack) ;

This means that if we receive a req (request) and reset is held low, this implies that within 5 clock cycles, we will generate an ack (acknowledge) or nack (no-acknowledge). If there are any possible circumstances that can prevent this from happening, the tool will find it and report it to us. Any failures are reported in the form of a counter example. This is a short set of vectors, which if applied to the design will make it fail to behave as specified. You will notice that we deviate from a

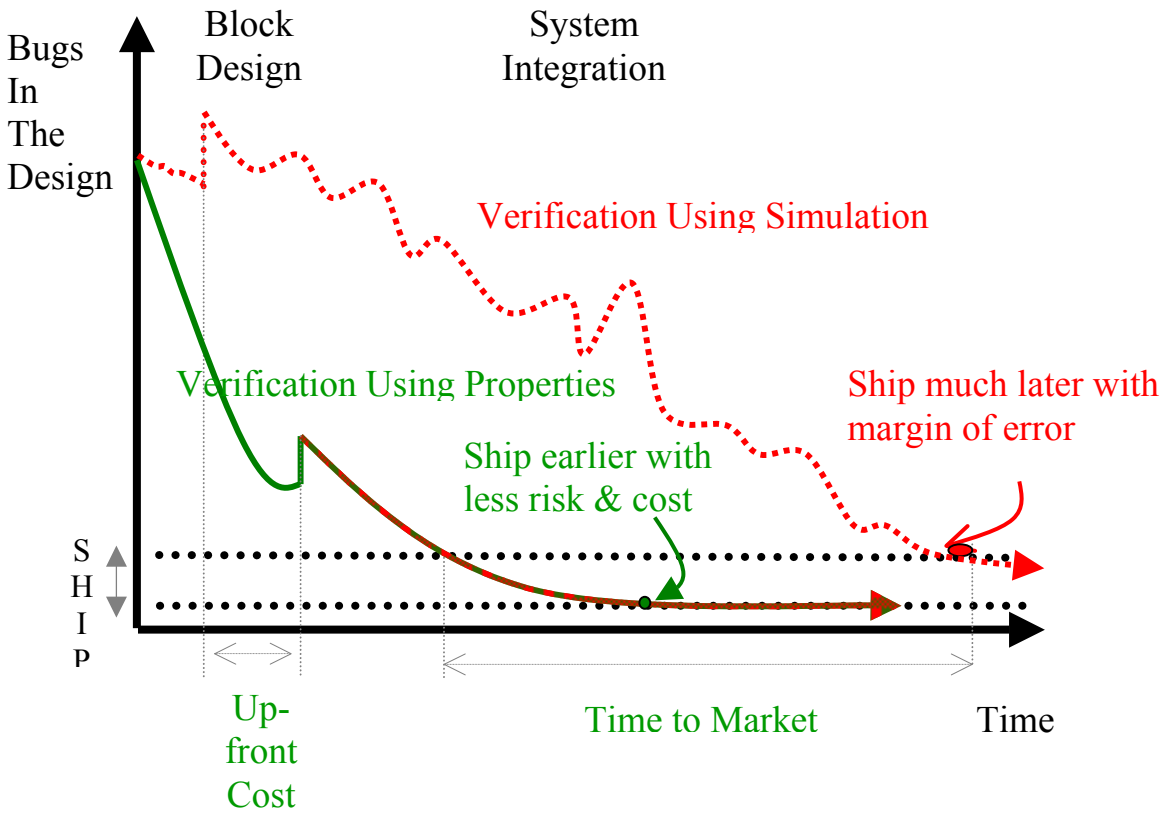
fully exhaustive search in only one way, in that we disallow the assertion of reset since we know that can prevent the design functioning. The above property takes less than a second of CPU time to verify on a Pentium III notebook.

Simulate Or Not To Simulate, That Is The Question?

You may be thinking, "Are they suggesting that we completely change our verification strategy in favour of something based on a static approach?" Well we know that's neither practical nor plausible. A property checking methodology is of most benefit at block level, where bugs and undesired behaviors can be eliminated at this early stage, the clean blocks would then be integrated together and simulations would still be run on the entire system as before. The advantage of using this approach is that the block level bugs have already been addressed, and will not have to be fixed during system simulation time. This will ultimately make the use of simulation resources far more efficient.

Methodology When Using A Static Approach

So the approach we often adopted in the past is that illustrated below by verification from start to finish using simulation. Blocks are written with minimal testing before system simulation begins. It's quite difficult to test and debug the blocks like this, since the controllability and observability is poor. Also system simulations are often large and slow to load and run. With inadequate block level testing, you fix one bug only to uncover something else and it's extremely difficult to estimate how far you are from a fully working system.



The approach we are advocating is the one illustrated by Verification Using Properties. By investing some additional time in developing a set of properties to rigorously test your blocks, you will gain a smoother integration process when performing simulation, with a much improved time to market window from a more efficient design process. In addition, properties can be synthesised and re-used where appropriate as assertions or monitors during simulation run time.

Summary

We have seen that Property Checking for Functional Verification offers an exciting new approach to verification, which works best as a complement to simulation, and is able to make system simulation runs far more efficient. Its key advantage is that it can quickly and exhaustively verify blocks, which are difficult or impossible using a simulation only approach. Property checkers also integrate well into existing verification environments, and do not disrupt the current flow. Indeed, properties can be automatically converted to assertions and monitors for use during the simulation process.