# Stress tests

By Darren Galpin

REUSABLE VERIFICATION COMPONENTS PROVIDE A WAY OF STREAMLINING THE PROCESS OF CHECKING NOT JUST LOGIC BLOCKS BUT PROTOCOL STACKS AND SYSTEM DESIGNS. USED IN COMBINATION WITH OTHER TECHNIQUES, THEY CAN PROVIDE A WAY OF STRESSING THE SYSTEM TO REVEAL BUGS OTHER APPROACHES WOULD NOT FIND.

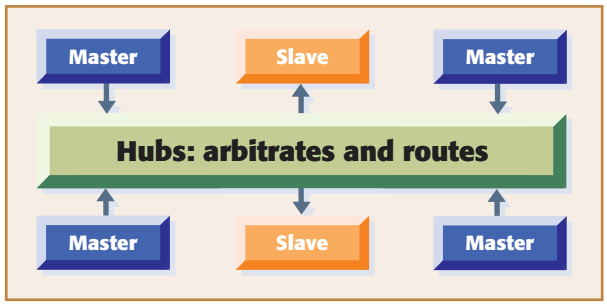



*Fig 1: The protocol and subsystems to be modelled*

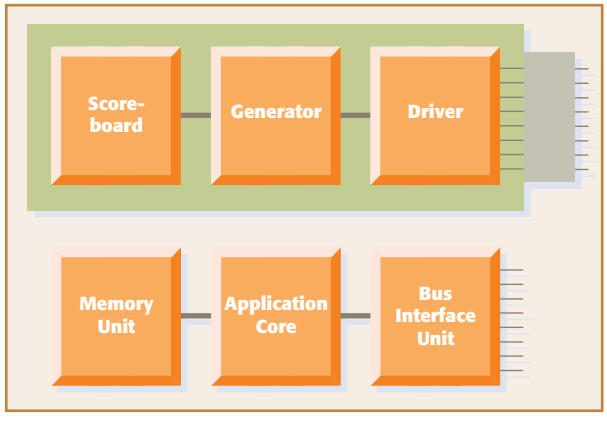



*Fig 2: An eVC Master and its HDL equivalent*

Building reusable devices for verification is becoming a must in order to reduce the verification overhead for designs. However, it is no longer enough to simply build a device which can only attach to a Design Under Test (DUT) or bus and operate in a single way. Instead, Verification Components (VCs) are becoming more common. Such devices provide a reusable way of testing devices with a common functionality or common protocol.

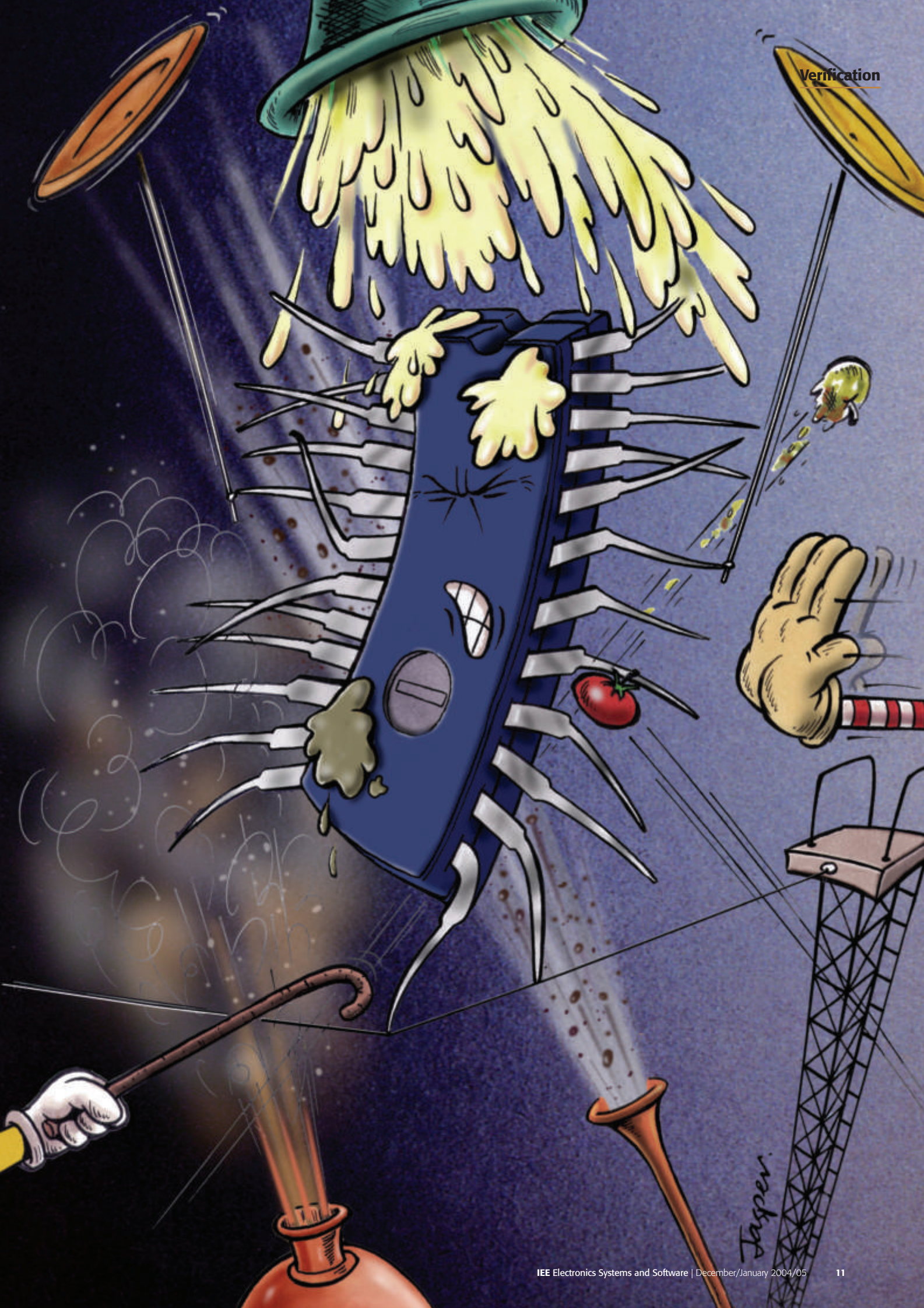
However, it is also becoming necessary to be able to reuse what have traditionally been block-level verification devices at the system level. At Infineon Technologies, we have built reusable verification components in the language 'e' that exploit random testing techniques. Incorporated into these devices is functional coverage, so they are compatible with coverage-driven verification methodologies. In addition, the eVCs have to be switchable, so that they can be replaced with a Hardware Description Language (HDL) equivalent as and when desired.

The techniques described were used on an eVC set written for a new internal protocol which features a hub and spoke system with point-to-point connections. A diagram of the system is shown in Fig 1.

Each of the components in the diagram grey must have an eVC model, and each must be swappable with a HDL equivalent. Thus the hub might be in VHDL and attached to masters and slaves written in e, or the entire environment might be written in e.

Normally, eVCs are written to be reusable in themselves but, if more than one protocol has to be supported, it is useful to make the verification components modular, so that the internal blocks of the verification component are themselves reusable.

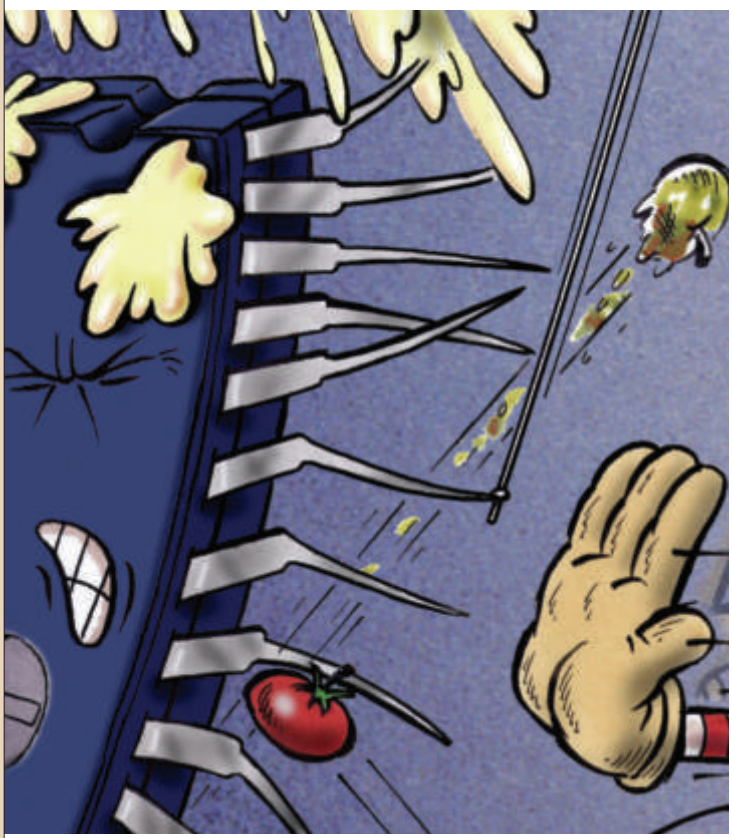
We split our eVC Masters into three logical blocks, based on the logical HDL equivalent. An example of this is shown in Fig 2. This split allowed us to develop components that can be reused across different eVCs. Thus we envisaged a master component as consisting of a scoreboard, a →


Jasper

generator that reads in constraints and a bus driver. The constraints that the generator reads in are protocol specific, but the generator is otherwise reusable across components. It is only the bus driver which is protocol specific.

An eVC slave is constructed in exactly the same way. It too consists of three parts: a bus sampling unit, a response generator, and a RAM model, which in our case consisted of a keyed list which is sorted by address. Again, only the bus sampling unit is protocol specific. The response generator reads in protocol specific constraints, but is otherwise reusable, as is the RAM. At Infineon, we refer to this approach as 'socketised IP'. However, the hub was a completely bespoke device, as it was only required to route and arbitrate.

The point-to-point nature of the system meant that a single protocol checker was inappropriate, so protocol checking was moved into the master, slave and hub as appropriate. It is at this point that the verification of the system starts, before any Register Transfer Level (RTL) code has even been run.



> **It is a fact of life that protocol specifications are rarely complete and often contain holes**

It is a fact of life that protocol specifications are rarely complete and often contain holes. Finding the holes when performing system testing is too late and costly, so moving the discovery upfront saves time and effort. The development of the eVC can be considered as a prototyping of the system – it allows you to try out the protocol and explore it before expensive system simulation is performed.

The construction of the protocol checking components can only be performed through a thorough review of the protocol specification. As the ultimate aim during the construction of the eVC is to create a golden reference to which everything will be verified against, it is very important to get this right. It is also the point at which you are verifying the protocol specification itself. If your verification components are fully random, then you should be able to exercise the full protocol if you run your eVC master against the eVC slave. In this way you can explore the full protocol space for deadlocks, livelocks and inconsistencies.

The protocol checks themselves can be performed in two ways. The first method is to code from the specification. This is both the simplest and hardest method: you read the specification and try to capture what happens at each stage as a transaction progresses. This involves checks for the default values on the signals, error conditions, checking that signals have or have not been asserted at the correct point, checking signal length and so on.

The second method is to reuse your formal environment. Random and formal verification environments are fundamentally the same. In each you have a set of assumptions to describe the environment in which it will operate, and a set of constraints to describe the legal behaviour. So, you can reuse the existing formal environment within the eVC. It is common to use sets of assertions to capture the behaviour of an interface, or a set of properties. Assertions can usually be translated directly into temporal expressions within the eVC.

Properties can be more complex, but typically only involve translating a finite state machine, which is used to keep track of where in the transaction space the ongoing transaction is, creating a set of Boolean expressions and a check. This assumes that you already have a complete formal description, though.

Whichever approach is used, a coverage point should be associated with each of the checks. When simulating, if no failures are observed, the user does not actually know what has been executed – she is only aware that nothing has been observed to fail. The use of coverage on the checks increases the observability of the verification. The user will be able to observe after testing which checks have been triggered and,

more importantly, which ones have not. Those which have not been covered indicate a hole in the testing, and a part of the protocol space that needs to be explored.

For the eVCs, a functional-coverage programme was constructed for both the Slave and the Master, on a per-instance basis, so that we could keep track of what we drove into our DUT, and what the DUT drove out. The functional coverage is constructed so that all facets of a transaction are recorded – the opcode, read/write type, acknowledge code and number of idles. From this, transition and cross-coverage can be performed.

Transition coverage is used to record what preceded a given transaction, so can be used to ensure, for example, that all opcode-pair combinations have been driven. Cross-coverage is used to combine, for instance, opcode coverage with acknowledge coverage, or the transition coverage for opcodes with read/write type coverage. In this way we can ensure that as much of the entire protocol space as possible has been explored.

## SYSTEM TEST BENCH

This type of functional coverage is different to that used for protocol coverage. If the latter is fully achieved, it only shows that we have exercised all of the different conditions, and we have not asserted signals at the wrong time. If all of the cross-coverage points are achieved, assuming we cross-cover all of the measured points, then we are sure that we have covered all possible bus occurrences, with the caveat that the bus pipeline depth is only two stages.

We now have the infrastructure in place to perform the block-level verification of components such as bus bridges, interface components and other, similar, devices and can now move to the system level. However, system testbenches are traditionally much more complex, and cannot use random data in all cases. For example, processors can only execute instructions that they actually understand. How can we reuse our verification IP at the system level?

The system testbench we used was for the TriCore-2 32bit, multithreaded embedded processor. For this, assembler-level tests are typically compiled into a Motorola S-record file, which are then loaded into on-system memories. Various peripherals are then located around the system.

The aim here is to enhance our verification of the processor using the newly developed eVC IP. It needs to be complementary to what already exists, so that all existing tests can be run without modification, and needs to be compatible with the processor. How could do we do this? We achieved this by extending the slave so that the processor could boot from it. This required the slave to behave like a boot ROM, and required the code to be preloaded before the test is run. At the beginning of system tests, the eVC Slave architecture had no facility for this.

We achieved the result by extending the Slave utilising our random environments language. When the random tool creates its environment, it runs a pre-generate method to set itself up before the test is run. By extending this method, we can force the slave to execute something before the test is run. In this specific case, we created a Perl script which processes the Motorola S-Record file into a list of addresses and data which can then be read straight into the slave. The extended method runs the script, and imports the resulting file. Thus the Slave becomes a boot ROM.

This is all very well, but we have not improved the testability beyond adding the potential for protocol checking during booting – although, admittedly, we can randomly vary the wait-states inserted between transaction responses. What else could we do? To generate interrupts, traps and other system exceptions, we would normally have to either cause a system error, or somehow force one of the system controllers to issue the exception. This can be difficult, especially as the exception has to propagate across the system. It would be far more flexible if these exceptions could be driven directly into the processor.

To aid this type of testing, four e-based 'pinwagglers' were created. These small pieces of e-code were used to drive the following: idle, interrupt, system trap and suspend. →

> **"The use of coverage on the checks increases the observability of the verification"**

The code sits in parallel to the eVC code, but also exploits its existence. They sit on top of the existing system connections, and force the processor inputs when they wish to drive.

In order to simplify the code, these units were written to drive the same interrupt value and trap value each time. However, the point at which the interrupt or trap is driven is entirely random. To ensure that a trap is only driven when the processor has been booted, a signal probe is placed within the random code onto the interrupt enable. When this signal changes value, the boot sequence has finished, so the trap can be handled and the context save areas set up – the random trap code can then be run.

No checking is built into the pinwagglers themselves – it is assumed that assertions within the code will trigger if we drive an interrupt or trap and it is not handled correctly. Instead, we are particularly interested in what happens when interrupts randomly arrive while executing a known instruction stream. We have historically used directed testing to test opcode and exception interaction, but this limits the number of scenarios tested considerably. By making it random, scenarios that have not been explicitly thought of can be automatically generated and tested. What has been tested is recorded using functional coverage. In this case, we monitor the exception controller and the pipeline stages and, on accepting an exception, we record what is in the pipeline at the time, and what is the state of our decode block.

The TriCore-2 supports Idle and Suspend modes. Idle switches off the clock to the core, whereas the suspend can halt the processor to allow off-chip debug. The suspend state can only be exited by resetting a register within the core.

The idle pin is easily driven by the construction of a random pin-waggler, again relying on in-line assertions (or checks built into the e testbench) and functional coverage. However, suspend is more complex. If I randomly drive a suspend by simply waggling a pin, it will stay in that state even if I remove the driving source. To get around this, we developed what we term transaction-injection.

The eVC Master discussed above consists of three blocks. However, the bus-driver unit is itself partitioned into two further blocks in order to handle the separate address and data phases. The cores of these blocks are internal pipelines that store the state of the transaction which is being processed. The bus driver takes whatever is on the bottom of the address pipe, and drives it onto the bus. Once granted access to the bus, the transaction is passed to the data pipe.

In order for the eVC Master to know what a transaction is, we have to define a data structure that contains a number of physical fields. Under normal operation, the generator block reads in its constraints, and randomly creates transaction data structures. The resulting structure is then placed into the address pipe for the eVC Master to process when it is ready.

**“Development of the eVC can be considered as a prototyping of the system – it can allow you to try out the protocol before expensive system simulation”**

Note that the eVC Master has no knowledge of where exactly the transaction came from – it simply sees a series of transactions placed into its address pipe. We can use this to generate a transaction of our own. Every time we drive a suspend signal, we wait a random amount of time before creating a new transaction, which corresponds to the transaction necessary for resetting the core register.

This new transaction is then injected into the Master's address pipe, even though the generator is filling the pipe at the same time. Note that the generator has been configured to only put a maximum of two transactions at any one time into the address pipe, so we know that our register-reset transaction will be at most the third in line. Care does have to be taken with this transaction injection – it can only be performed between atomic transactions. If a composite or locked transaction is queued up, we have to ensure that the reset transaction is injected after this has completed. If not, we could cause the master to attempt to violate its own protocol constraints.

We also have to take care with address constraints. A random generator will attempt to generate transactions over the entire legal address space, and this register reset has to be within the legal address space. However, the generator used for the eVC master generates sequences, rather than individual transactions. Thus we constrain the sequence to use all addresses except the register address, while constraining the master to be able to use all of the address space.

The true test of the system is to run all of these exception drivers simultaneously while the core is executing code. However, we can stress the processor still further. The TriCore-2 has two slave ports connecting to the hub, one for the data scratchpad and one for the code. Each of these scratchpads is accessible from the system. We can constrain our eVC Master to write into these areas, being careful in the design to ensure that the eVC and processor do not conflict over the same area of scratchpad memory. In this way we can particularly stress the system, as the memory management unit has to service both the core and external accesses simultaneously.

At the same time, we can locate the context save areas into the SRAM on a slow bus. This has the advantage of forcing random stalls into the simulation – every time that we drive a random interrupt or system trap, the processor will have to access the slow bus, which can take tens of cycles. This increases the likelihood of getting further exceptions while the internal pipeline of the processor has stalled and has branched, a suspected bug hot-spot.

Within two weeks of setting this system environment, six new bugs had been found in the processor block, all caused by the interaction of multiple exceptions while the processor was branching or jumping – and this was despite the directed test suite nearing completion. A further bug was uncovered when the tests were used as part of a regression – this bug manifested itself once other parts of the design had been fixed. All of these bugs had not been found by the previously used directed techniques, and would have been unlikely to be found – the bugs were often extreme corner case bugs with specific entry conditions.

Once this approach had been proven, the hand written assembler component was replaced by assembler generated randomly by an Instruction Stream Generator (ISG). This ISG is again a constraint-based model of the system, and generates its assembler by reading a constraint file, and testing for the expected outcome by using a C-model during the generation process.

This approach in particular has proven adept at finding bugs. Scenarios which would have otherwise been unthought of were hit randomly, and more than 30 new bugs were found. Coverage-oriented verification was of particular help here – by highlighting which areas of the function space were not covered, new tests could be rapidly aimed at new areas of the design, and new bugs found.

The development of eVCs early in the design phase prototypes the system, and allows the exploration of the protocol space during the construction of the units. If done properly, then the holes in the protocol description can be established before the design phase or the system construction is underway.

> **“Six bugs were found in the processor block, all caused by the interaction of multiple exceptions while the processor was branching”**

Block level eVCs can be re-used in the system environment. If used without major modifications to the existing system test infrastructure, they gain rapid acceptance. If simply used to stress standard interfaces, they can find bugs. However, their usefulness can be enhanced by creating extra units which also stress the processor core while exploiting the existing eVC architecture.

Of paramount importance is the functional coverage. As system designs are so big, structural coverage gathering – that is, statement, branch and toggle coverage – is not often run in reality due to the simulation overhead. The ability to record immediately what has been driven and received by the eVC masters into the DUT rapidly allows you to uncover test holes, and to generate new tests to cover them quickly. It should also complement any existing functional coverage of the processor. ■

**Darren Galpin is with Infineon Technologies**