Confidence comes from the complete spectrum

Silicon Hive
Maarten Arts

Session #245

Abstract

In this session the importance of using the complete spectrum of verification tools is highlighted, as well as automating random test generator development and reusing verification IP. Only this way Silicon Hive is able to cope with the functional verification complexity, as VLIW DSPs are generated from a high level processor description. The protocol level is where the customer interfaces with the IP, so it is extremely important to get this absolutely right. Using AMBA verification IP with Incisive Formal Verifier (IFV) and Specman gives great confidence in correct protocol implementation. When block level verification is using IFV, bugs are already found and fixed before integration. Random test generation is done on most blocks and memory subsystem. The e code for the memory subsystem is generated automatically using an internally developed tool and the abstract processor description. An evc has been developed internally to verify the proprietary interfaces within the processor. At integration level, accessing the processor from the system and basic processor functionality is verified, using a combination of SystemC and NCsim and using an FGPA prototype. Using code and functional coverage is mandatory in understanding where you are in the verification process. Using functional coverage very powerful constructs are built to see if for example all possible write back buffer read/write conflicts have been generated. By applying all tools in the verification tool spectrum and automating random test generator development we can create very high confidence in Silicon Hive IP functionality, in weeks rather than months.

1. Introduction

There is a growing need for processors that are application specific. Such processors will be used in embedded systems where power consumption should be minimal. So most power, if not all, should be used by the processor performing arithmetic, processing the data that the embedded system has been designed for.

Programmability is important, as standards that these embedded system operate upon will change in the process of drafting them, or are updated in the future.

Silicon Hive has found a solution for this. All the complexity of the processor that is not related to computations has been moved to the C compiler. Processors are generated from an abstract description, which makes it possible to tune the processor exactly to the computational needs of the application, minimizing power consumption and silicon area.

3 prefab cores are available off the shelf: an image processor: ISP, a video processor: VSP, and a communications processor: CSP. Possibilities are not limited to these specific processors: using a proprietary high level language it is possible to construct a wide variety of processors and tune them to a specific application while still being fully C programmable.

Due to this configurability and generating the processor from a high level description, it is difficult for the functional verification team to keep up with the development of the hardware, the verification bottleneck is in our case worse than ever.

This paper describes how we are able to cope with this problem by using all the verification tools available from the spectrum, and automating as much as possible, even generating e code from the processor description. The paper finishes with a conclusion and what we will work on next, but first a short description of the processor and software toolchain are given.

2. The device under verification: processor and software tool chain

To understand what is actually being verified a short description of the processor architecture and generation flow is needed. This description is by no means meant to be exhaustive.

Starting point of the generation flow is the processor description in a proprietary language, TIM. It contains details on which hardware building blocks should be used and how they are connected, for example which arithmetic blocks should be used, the sizes of the register files, data widths, memories, interfaces to the rest of the system. The result from the generation flow is synthesizable RTL and a Software Development Kit (SDK).

The hardware building blocks are written in a proprietary language, CHDL. This language is very similar to VHDL, but has additional constructs for configurability and the RTL generation flow.

The generation of the processor core and software development kit from the processor description is illustrated in Figure 1.

The main advantage of being able to generate the processor is the quick iteration time. With a tool that is part of the software development kit you can analyse how well the application maps onto the processor. With this information you can either optimize the application for the core, or the core for the application, or both, to get to an optimal result.

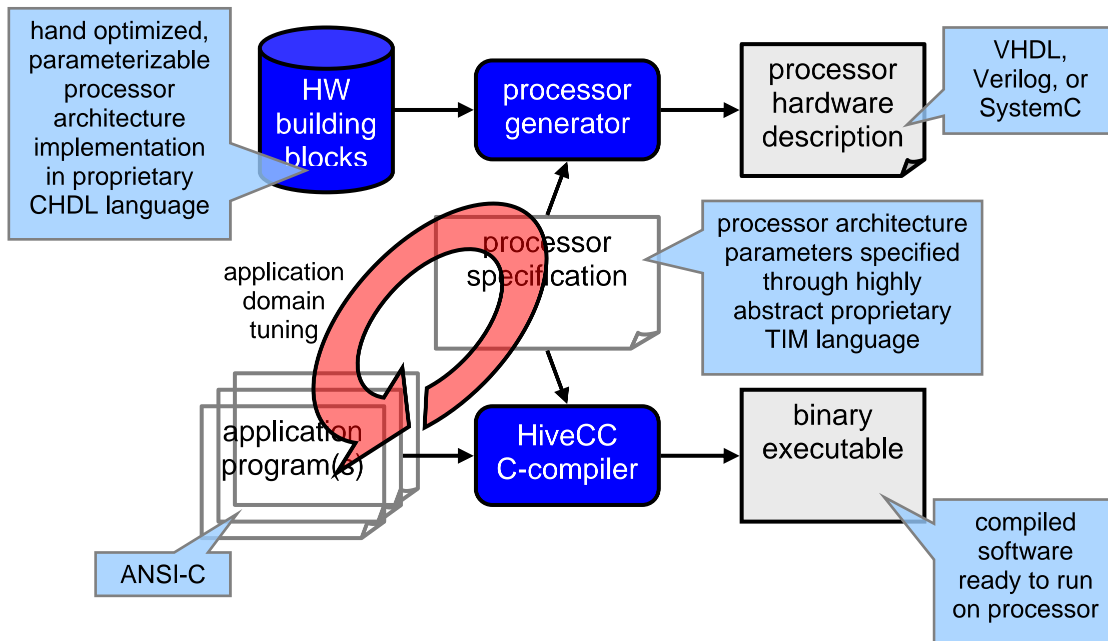

Figure 1: The Silicon Hive generation flow

With this flow it is possible to create a wide variety of processors, optimized for all kinds of different application areas.

The underlying architecture is based on the VLIW (Very Long Instruction Word) principle which enables massive instruction level parallelism. Data level parallelism (SIMD) is implemented by the functional units, and task level parallelism is achieved by using multiple Silicon Hive cores in a system.

The core is designed to be as computationally efficient as possible, a lot of the hardware complexity has been moved to the C compiler. The compiler schedules instructions taking care of all the resource conflicts.

All processors are built from a generic template, see Figure 2. The core contains the functional units, most of which implement arithmetic functions, and some implement instructions that will load or store data from the memory subsystem. These functional units are connected to the memory subsystem through a proprietary bus protocol. An issue slot can contain multiple functional units, and can start a new instruction every clock cycle.

The core also contains the register files that contain the source and result values for the functional units. The sequencer in the core controls the program flow. The host can start an application using the status and control register located in the memory subsystem.

The memory subsystem contains any number of memories. These can be accessed from one or more functional units, and from the system using the slave port. An arbiter between the functional unit and the slave port takes care of access conflicts. A stream interface can be used to synchronise several Silicon Hive cores in the same system, or to stream data to and from the processor. With the master port the processor can actively access data located in the system. These master and slave ports can be any standard bus protocol, AHB, DTL, AXI.

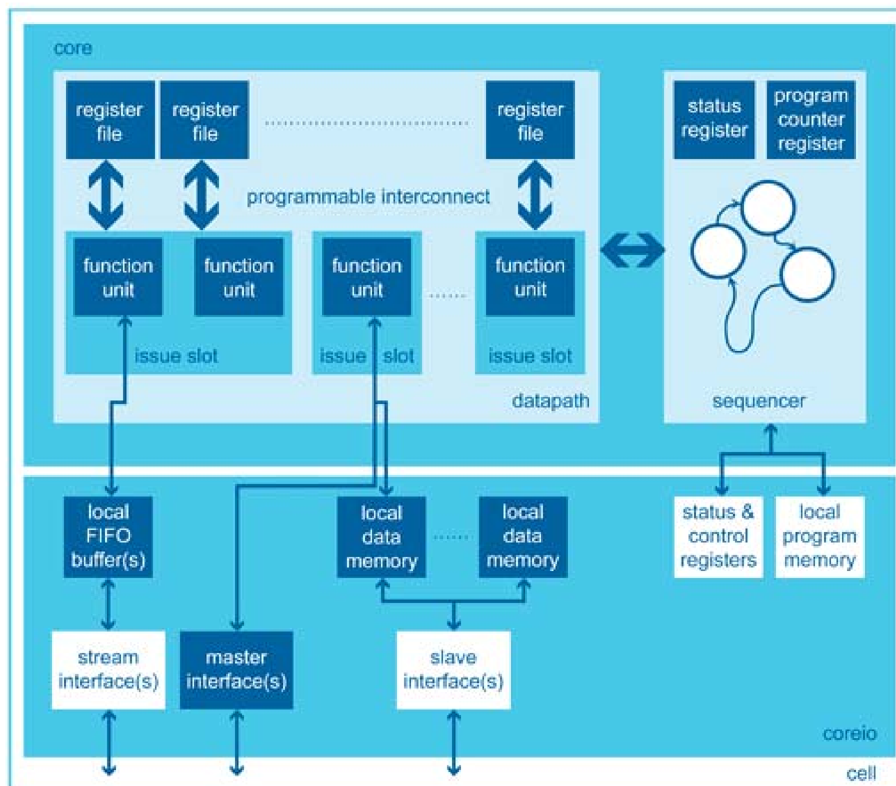

Figure 2 processor architecture

3. Functional verification bottleneck

Without taking proper measures functional verification would be a massive bottleneck for Silicon Hive: the design productivity has been increased being able to describe the processor at a higher level than RTL, and the amount of possible processors with the processor architecture is infinite.

Just making sure that the building blocks are working correctly is not enough, the generation itself and the software tool chain need to be verified as well.

As an IP company, it is very important to have good confidence in the quality of the designs that are shipped. Especially the interfaces to the system, at the level that the customer integrates the IP, need to be absolutely correct.

And all this needs to be done in a reasonable amount of time with a reasonable amount of resources.

Using all the tools in the functional verification tool spectrum, and automating as much as possible was the solution for Silicon Hive. In the next few sections the tool spectrum will be explained in detail.

3.1. Silicon Hive verification spectrum

In the last few years we have been carefully constructing the verification flow to a mature level. This work is of course never finished, see section 4 for our plans for the coming year. Table 1 summarises the verification tools in our spectrum.

	C	IFV	specman	ncsim	systemC	conformal	FPGA
Module	X	X	X	X			
System				X	X	X	X

Table 1: the Silicon Hive verification tool spectrum

C

When a module or functional unit is being developed, this is the entry point in the verification flow. This is a very early step in the design phase, remember that the later a bug is found, the more expensive it is to fix it, so better find bugs early. The Silicon Hive internal development environment creates an executable C model and testbench from the CHDL module description. The testbench generates random instructions and source operand data for the instructions that are implemented by the functional unit. These random stimuli are not very refined, and no code- or functional coverage can be applied to the C model, but this is good start to filter out the first problems. This step in the verification process runs every night for all modules, so if there is a regression in the basic quality of one of the modules it is found instantly.

IFV

Incisive Formal Verifier is the latest addition to our functional verification flow. The modules that the tool is used on come out very clean and ready for integration, saving debug time and iterations later on. Ramping up time is very quick, a couple of PSL properties and assumptions can be written in a few hours for first pipe cleaning.

When more complex PSL properties are written it is likely to encounter some properties that do not complete, that are in the IFV explored status. The tool cannot prove or disprove them in a reasonable amount of time. One solution for this is to simplify the design. There is a risk to oversimplify the design, and to miss bugs.

Compared to formal tools in the past, you do not have to be a formal methods expert to be able to use the tool.

Specman

With the Specman verification environment a test generator for basically any digital design can be implemented. Within Silicon Hive, a test generator has been implemented for the dma module, a special video memory, and for one of the most critical blocks in the processor, the memory subsystem. See Figure 3 for a diagram of this test generator. Next to containing the internal memories, the subsystem also connects to the outside world. The IP is integrated at this level by the customer, so it is absolutely vital that these interfaces work correctly.

An e verification component (evc) has been developed to generate stimuli for the proprietary bus that connects the functional units with the memory subsystem. The AHB evc is licensed to generate stimuli for the AHB interfaces. And a DTL evc is used for the streaming interfaces.

For each of the evc's configured to be a master, a large sequence library has been developed. With this sequence library three types of tests have been written: simple tests that exercise all functionality once, a test that exercises all functionality simultaneously, and a test that stress the design to its maximum, trying to find all corner case bugs. These tests usually run for 2 weeks continuously whenever a major design baseline has been delivered.

A scoreboard and e memory model make sure that all transactions are correct, that no transactions are lost, and that there are not too many transactions. There are also some very specific checks that are used to measure the performance of all transactions that go to/come from the system level. We found out that our performance to the AHB bus was not optimal, so we iterated until we got the maximum performance.

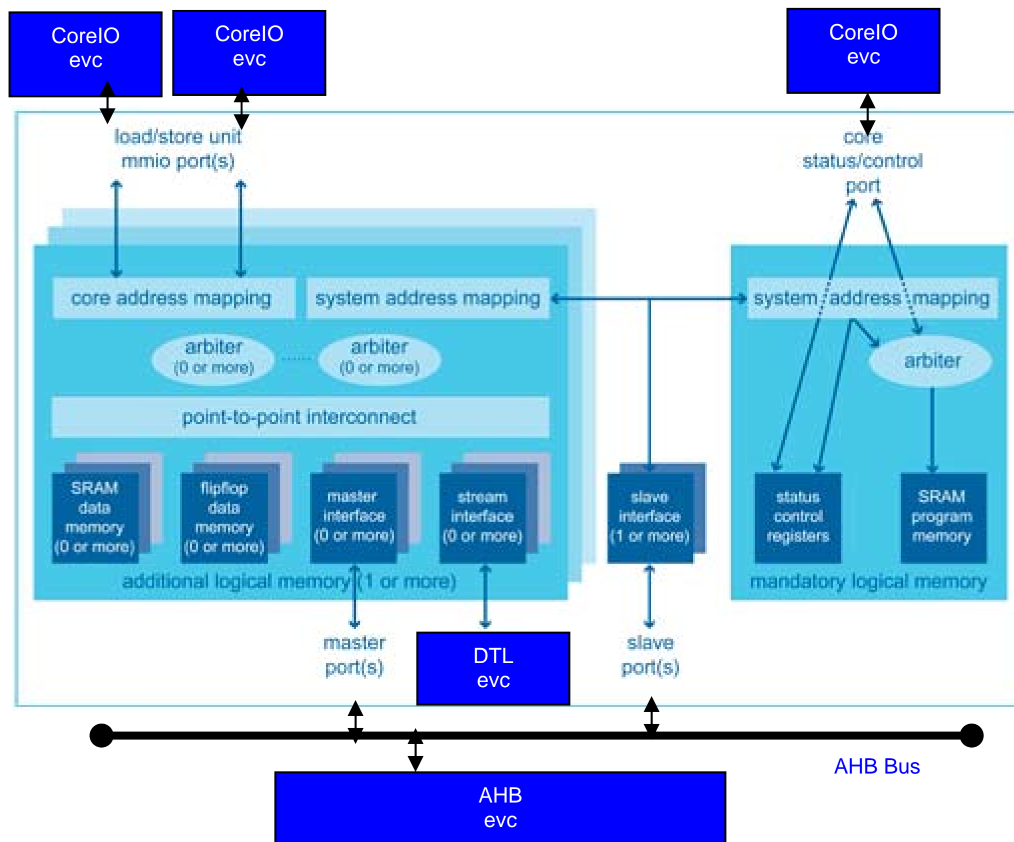

Figure 3: Specman test generator for memory subsystem

Functional coverage is used to make sure some very specific scenarios are generated. We found a mistake in a Specman test where not all AHB transaction types were used in one test. When we corrected the test, a rare sequence of AHB transactions was generated and found a bug.

Functional coverage is also used for example on the write back buffer, a mechanism to speed up write transactions to the system level. The write back buffer keeps several written addresses and corresponding values. There are several access conflicts possible:
- a read from an address that is still kept in the write back buffer should return the value from the write back buffer and not from the system.
- a write to an address that is still kept in the write back buffer should overwrite the value in the write back buffer.
When generating random transactions to the system, you can only be sure that the interesting write back buffer scenarios have been generated when using functional coverage.

Due to configurability there is a infinite amount of possible memory subsystem variations, it is impossible to verify them all. So only test generators are developed for the processors that are on the shelf, and for 12 specific test cores. Even for this number of cores it is impossible to write test generators by hand within a reasonable amount of time. If it is possible to generate RTL from an abstract processor description, then it should be possible as well to generate e code for the test generator, so this is what we did. Of course it is not possible to generate the evc's themselves, but the instantiation and configuration of the evc's is generated, as well as the 3 test as described earlier.

A perl script scans all the log files for failing tests, and creates a batch run script to rerun all the failing simulations with the signal traces enabled for debug, and after fixing the problem, to rerun the test to see if the problem has been fixed. At the end of each test the number of read and write transactions for each evc is printed. The perl script also scans for these numbers, and creates statistics for all transactions in all tests generated for a particular processor.

Ncsim: VHDL/Verilog/SystemC/C cosimulation

Ncsim is used in combination with Specman, but also to run the testbench that executes the tests that are written by hand. These tests are:
- example algorithms for customers and for internal development
- example applications, written by Silicon Hive and received from customers
- benchmarks
- direct tests, verifying some very specific scenarios
- tests for the connection to the host
- integration tests for customers

The SystemC part wraps C code that models the connection to the host. The VLIW processor IP is written in VDHL and/or Verilog at RTL level, or Verilog only at gate level netlist level.

The tests above are part of a regression that automatically runs each time when there is a new baseline. The release of a new baseline is detected, and a regression run is started automatically.

Writing tests by hand is quite time consuming, so the purpose of this platform is mostly to run tests and applications that have been written in C. This way the complete chain from compiling the C code with the SDK, to uploading and starting the test through the host, is exercised.

Block coverage, Expression coverage and FSM coverage numbers are collected, and merged where possible. Analysis of the results and using the feedback to create new test cases is a very time consuming task, but mandatory and worth the effort.

Conformal

The Conformal equivalence checker is used to make sure that several representations of the design are identical to each other. The customer receives the script that checks RTL versus gate level after synthesis. Internally we also use scripts that check the gate level against gate level after layout. We also have 2 RTL to RTL scripts, first when a module is being re-written without changing the functionality or location of registers, and secondly when an option is enabled that slices the register files in order to make routing in the back-end easier. In both cases you want to make sure that the modified RTL is absolutely identical to the original RTL.

The Conformal tool is also able to do some structural checks. We have not spent a lot of time with this so far, it is not a standard part of our verification spectrum yet. Currently we are implementing the checks to make sure that clock domain crossing is correctly implemented.

The tool does not report problems very often, and when it does it is usually a user error. We have found a problem with a synthesis tool from another vendor once. When we moved to another technology library the logic resulting from synthesis was not correct. The tool does

give complete confidence about different designs being functionally identical and does therefore create peace of mind, even when not so many problems are being reported. Also, the equivalence scripts are mainly a service to our customers, as they do the actual synthesis.

FPGA

An FPGA platform has been developed internally, and is used to verify the complete system, for demonstrations and for prototyping. For example, for the ISP we have a prototype with an image sensor and a LCD display. An image is captured by the sensor, delivered to the Silicon Hive ISP running an internal image processing pipeline or one delivered by a customer, and the results from the processing is visible on the LCD. See Figure 4 for the diagram of the HiveGates ISP prototyping platform. A prototype like this is a very good confidence builder with customers as they can actually see, touch and use the system.

In some cases the applications that are used with the Ncsim simulation platform take too much time to complete. Running them on the FPGA saves a lot of time, although debug is a lot more complicated.

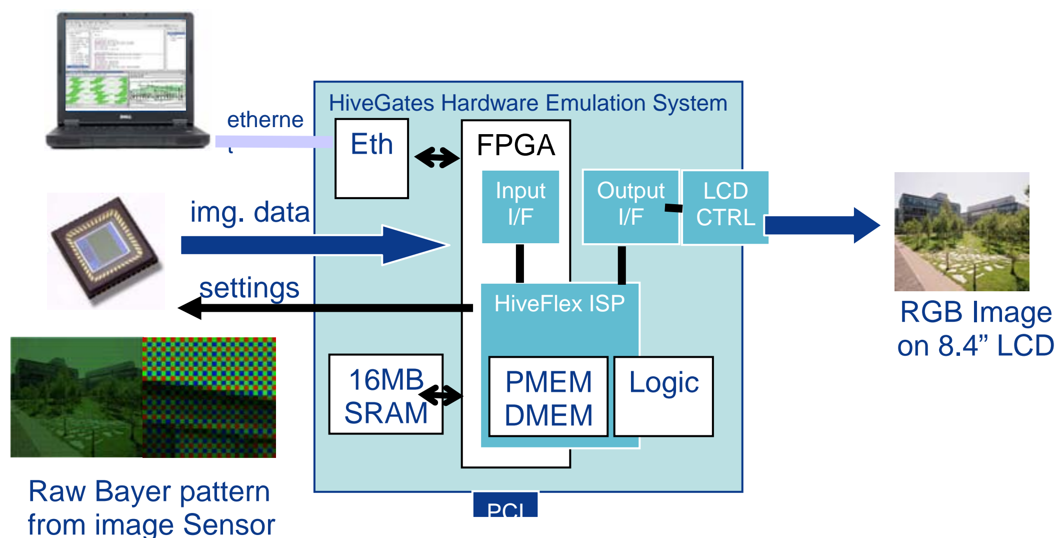

Figure 4: HiveGates FPGA prototyping platform for ISP2200

3.2. *Importance of using the complete spectrum*

Having gradually built our verification flow in the past few years we have come to the conclusion that it is very important to use the complete spectrum of tools that is available to you. Leaving out one element immediately increases the risk of a bug being left undetected.

Each time a new element was added to the verification tool spectrum, new problems have been found. There is certainly some overlap between the different tools, but it is better to deliver the best quality you can and have a very high level of confidence in your design, then to save some time and money and deliver an inferior design to your customers.

Also, each element has its strengths and weaknesses. The weaknesses of each element are covered by other elements in the tool spectrum:
- the C testbench is fast and early (remember that later bugs are more expensive to fix), but it is very random and it will be rare to find a complex corner case. System level issues will not be found by module level tests.
- IFV is thorough, it is early in the design phase, and it is quick to get started with, but sometimes there is a need to simplify or scale down design. In one case, with a dma design, we needed to simplify a counter, and found a bug later on in the verification process when

simulating the design using the Specman dma test generator. We probably oversimplified a counter in the dma design and did not trigger the bug. This would have been a killer bug on silicon as it locked up the complete dma.
- Specman is a very flexible test generator development environment, but it is difficult to be exhaustive. Implementing functional coverage helps, but is a time consuming task. Due to the controllability of the randomness, the Specman test generators are very effective bug finders, also for complex bugs when the test generator is instructed to generate very stressful tests. The dma test generator found a bug that was not seen in IFV due to simplification. This is just one of the justifications to use both tools, it is the combination of tools that gives best possible confidence.
- the NCsim:VHDL/Verilog/SystemC/C environment finds mostly system level bugs, problems with the toolchain or problems with the host accessing the processor. Also, as this environment is used for regressions, it is a vital indicator of the stability of all deliverables.
- Conformal has no overlap with other tools. As we provide our customers with an equivalence checking script to prove that the RTL is identical to the gate level netlist after synthesis, we need to make sure that the script works and that the check passes.
- FPGA is a fast simulation platform, and mandatory for applications which are too long for RTL simulation. And above all it is an excellent confidence builder, also with the customer. If there is one single small problem, anywhere between the image sensor and the LCD, it will be very visible from the image that can be seen on the LCD. There is some overlap with the NCSIM:VHDL/Verilog/SystemC/C environment, but the FPGA environment has its own justifications as just mentioned.

3.3. Automate as much as possible

All these environments are quite a lot of work to implement, but also to use, and to maintain or update according to new or changing functionality. To be able to cope with all this effort, you need to automate all the routine work that can be automated, leaving time for the more interesting implementation work.

Things like regression runs are a good candidate for automation. Why kicking off the flow yourself when you can have something like a cron job that watches for a new release being made, then generates the RTL code and SDK, starts the regression and collects the results automatically ?

Also the development of the Specman subsystem test generator is automated. This is not the case for the evc's, they still need to be developed by hand, as well as sequences, but building the test generator that makes use of the evc's is done automatically. This saves a huge amount of time. In our case it was not so complicated, as there was already a framework in place that could extract design information from the abstract processor description. The same framework is used to generate e code for the complete subsystem test generator: the e code for the instantiation of the Coreio, AHB or DTL evc, the e code to configure these evc's, the e code that implements the simple, normal and stressful tests, and e code that synchronises the different evc's with each other.

In our case it was mandatory to automate the generation of the Specman subsystem test generator as we would not have been able to cope with all the possible configurations.

For others not so fortunate to have a framework lying around for extracting processor design information, there are other possibilities to generate a system level test generator automatically. It already saves time to work from something like a simple system level design description file, and write a script in Perl or Python that generates the e code for all evc instantiation and configuration files.

Specman elements can be re-used, it saves a lot of time to license evc's for standard interfaces, like the AHB, AXI, or OCP bus. Licensing evc's makes sure that IP can be easily integrated into customer system on chip, and by licensing verification IP that is used by a lot of other companies this will also set some kind of common quality standard. Developing your

own evc's for proprietary busses is worth the effort as well, as these busses will probably not change very much after initial development, and the evc can be re-used again and again.

The Ncsim:VHDL/Verilog/SystemC/C environment is automatically configured, there is no need to write any testbench code by hand anymore. After simulations did run, code coverage report generation scripts are called to generate an html report and summary reports.

Also the Conformal equivalence checking scripts are generated automatically. With the recent RTL Compiler tool versions, a Conformal script is generated automatically.

4. Future work

A verification engineer's work is never finished. There is always something to improve or add to the verification flow. The following things are what we will focus on in the next year.

- Formal verification IP for standard bus interfaces.
As with licensing Specman evc's for standard busses, it will save a lot of time when formal verification IP is used. It would be interesting to see if the formal verification IP can find bugs that were not detected by the Specman evc's. There is also a need to understand if there is not too much overlap here.

- Specman test generators for processor and system level.
Some studies we did revealed some frameworks for processor test generation that could be used, like the ISG or OpenCores processor test generators. Such processor test generators need some knowledge about the instruction set. Such information can also be extracted from the processor description using our framework, so processor test generators can be automatically generated to a large extend.

- Use Conformal to make sure that generated Verilog is identical to VHDL
The Silicon Hive RTL generation flow generates VHDL or Verilog RTL code for the same processor. Conformal will be the tool to prove that the generated Verilog model is functionally identical to the VHDL model.

- Advanced regression and test generator results collection and reporting.
A lot of the reporting is done by hand, or merged by hand. It will be possible to collect all the results from all the different elements in the verification tool spectrum, and to generate one report with all results automatically.