Originally posted at EETimes.com on 29 Mai 2007 at:

http://www.eetimes.com/showArticle.jhtml;jsessionid=DQ4L5J2JPINQ4QSNDLRCKHSCJUNN2JVN?articleID=199900575

and reprinted with permission of EE Times.

Achieving Highest, Certified IP Quality Efficiently

by

Lorenzo di Gregorio, Infineon Technologies AG

Carlo del Giglio, Michael Siegel, OneSpin Solutions GmbH

While the increasing use of design intellectual property (IP) has considerably reduced design effort per gate for the chip designer, it has had an inverse effect on the chip-level integration and functional verification effort. IP verification and correct integration have become a dominant source of effort and risk in system-on-chip (SOC) projects.

In this article, we explain how we used complete formal functional verification that enables us as IP providers to certify highest IP quality, and to do so cost-effectively and with a high productivity of 2,000 to 4,000 lines of verified RTL code per engineer-month. The resulting IP quality significantly reduces the IP integrator's effort, cost and risk. Such results have the potential to fundamentally change the proliferation rate of IP and the profitability of the IP business.

IP development and integration – the effort and risk

With the increasing use of IP as the main building blocks in SoC design, SoC design quality is increasingly determined by IP quality. Unfortunately, there are no measurable criteria or standards to enable fast and objective IP quality assessment. Consequently, IP integrators are obliged to resort to evidence such as the IP provider's reputation, or to restrict IP use to that which has already been successfully integrated into many SoC designs. Any IP integrator who wishes to have a more analytical assessment must undertake time-consuming reviews and inspections of the IP provider's verification process and results for the IP.

Most redesigns and respins are caused by functional errors in the modules and IP, and by the interaction between them. The lack of objective metrics for module and IP quality is a primary effort sink for IP providers and IP integrators trying to control the resulting risks. Nonetheless, all too often, this considerable effort still results in error escapes, costly respins and delayed time-to-market.

Delivering and integrating the highest quality IP is gradually becoming a condition for staying in business for both providers and integrators.

So, what does the IP integrator need?

The IP integrator needs IP that operates error-free and exactly to specification, together with an objective certification to this effect; and a precise description of the conditions under which the IP can be reliably integrated into a chip, thus ensuring and speeding its

correct integration. The degree to which the IP meets (or fails to meet) the integrator's requirements determines the risk of using the IP.

The IP Provider's Challenge

The primary challenge of IP providers is to meet the IP integrators requirements, and to do so with the high productivity that enables the provider to meet the IP's time-to-market and profitability objectives.

But how can you do this? How can you ensure error-free operation using coverage-driven verification methodologies that are inherently incapable of testing all possible functional scenarios under all possible conditions? How can you give the IP integrator an objective certification of the IP's quality? And how can you rigorously define the IP's integration conditions if the verification methodology examines only a subset of the possible operating conditions and scenarios?

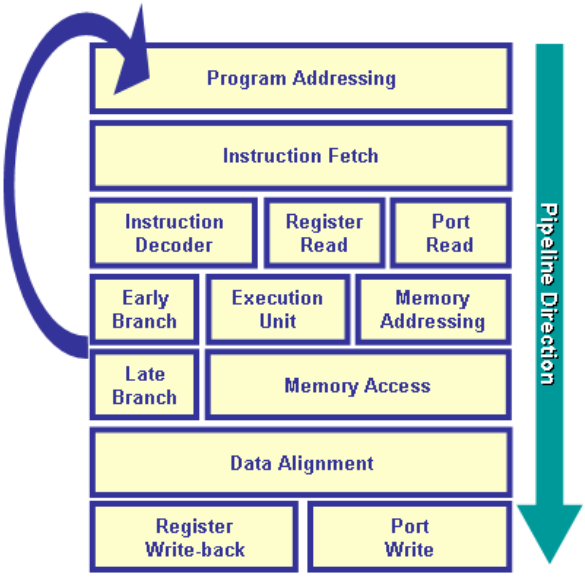
This is how we do it in the Intellectual Property & Reuse department (IPR) of Infineon's Communication Solutions business group (COM).

The IP Provider

IPR supplies COM's worldwide product groups with in-house and third party IP components. The department's comprehensive IP reuse strategy comprises building its own IP, qualifying third-party IP, and qualifying and packaging existing blocks from Infineon's business units for safe and broad reuse. IPR must ensure the IP's right-first-time functional operation and early availability. For this purpose, IPR employs a mix of advanced verification approaches, including testbench automation and formal verification techniques.

IP Example: A Network Processor

IPR developed a network processor, the PPv2, to meet the demanding throughput requirements of COM's wire-line chips across a wide range of applications. The PPv2 is a compact, high-performance, configurable 32-bit RISC processor with an application specific set of 40 instructions, a seven-stage pipeline and fine-grained multi-threading (see Figure 1).



1. This figure shows the pipeline and partitioning of PPv2.

The PPv2 was designed from scratch while retaining full backwards compatibility with its predecessor, PPv1. The PPv2 supports multithreaded execution with up to four contexts, which can be switched with no constraints, overhead and response delay. This feature enables up to four “virtual machines” to execute on the same architecture, allowing a machine to execute (for instance) while another machine has stalled, waiting for a response from the periphery.

Contexts can be arbitrarily switched and/or restarted by both external events and dedicated instructions. In addition to multiple branch instruction types, the context restart can also be employed as a branch. Branches are categorized as “short” branches, which decide whether or not to branch based on the value of status flags, and “long” branches, which make a decision after first performing a more complex operation such as a decrement with a subsequent register value test.

Every branch instruction inherently incurs a time delay penalty, that is, a number of cycles during which the processor executes no instructions. To increase the processor’s performance, the PPv2 architecture supports for each branch instruction configurable “delay slots” for each branch. However, the use of delay slots tends to increase program size, so the PPv2 offers a trade off between performance and code volume by optimizing the use of branch instructions and delay slots. Moreover, the PPv2 micro-architecture has to resolve the complicated conflicts that arise from operations under external exceptions and/or operations in the delay slots. Such conflict resolution must be executed while retaining the consistency of the programmer’s view and at no cycle cost. The resolution logic deploys quite sophisticated structures to dynamically buffer and reproduce instructions and program addresses, that is, “on-the-fly” resolution while the pipeline continues to execute.

The whole design amounts to roughly 11,000 lines of VHDL code, but this is a poor metric of operational complexity, the challenge of verifying the design, and the risks of not doing it thoroughly: being a programmable IP, any corner-case malfunction that remains undetected during verification has the potential to jeopardize the hardware and firmware development in subsequent reuse, together with the processor software

development tool chain that was developed concurrently with the hardware.

The IP Verification Challenge

The verification objectives were broadly similar to those of any processor verification. On the hardware side, we had to ensure the correct pipelined processing of multiple instructions, guaranteeing no undesired interferences between instructions; and ensure the correct operation of permissible, but unpredictable, behaviors such as traps and interrupts. In addition, we had to comprehensively verify data paths with complex bit-manipulations; and ensure independent execution of multiple threads under all possible combinations of instructions, thread switches, traps and interrupts.

To simplify and ease software development, we had to guarantee that pipelining, together with related forwarding and stalling behavior, was transparent to the software programmer. With 40 instructions we had six three-register instructions, eleven two-register instructions, thirteen one-register instructions, and ten no-register instructions. For one pipeline stage, there were thus $(6*16^3 + 11*16^2 + 13*16 +10) = 27{,}610$ scenarios to verify. Given that four pipeline stages are used in forwarding, and that each stage can have a different context, this produces a total of $(27{,}610^4)^4 = 27{,}610^{16} = 1{,}14*10^{71}$ scenarios, and this does not even take data values into account.

Moreover, we had to ensure that there was no branch configuration conflict that could "break" the programmer's view. Now, each branch instruction can be configured to have 0, 1, 2 or 3 slots. With twelve branch instructions (long and short), that yields $(4^{12}) = 16.7$ million different scenarios to be verified.

In addition to these challenges, we had a specification of 130 pages that, like most specifications, contained ambiguities. This specification had to be completed and clarified throughout verification.

Finally, we had to precisely describe the integration conditions that must be met by any hardware/software environment into which the PPv2 is integrated, to ensure the processor's correct operation.

Given the huge number of scenarios arising from the combination of configurability and context switching, we deemed it impossible to thoroughly verify PPv2 using simulation. Simulation would have necessitated choosing a representative subset of scenarios to verify, resulting in verification gaps and possible error escapes. Choosing not to verify parts of the design was clearly not an option. We needed a complete verification.

We had employed a combination of simulation and formal verification to verify PPv1, and found that the completeness and productivity of the OneSpin 360 Module Verifier delivered superior bug-detection effectiveness and efficiency. Consequently, we decided to verified PPv2 using only this formal verification solution.

Property Development

The specification defined, among other things, how instructions modify the registers that are visible to the programmer, for example, program address, register file, status flags, and described the memory traffic for instructions and data as well as responses to exceptions. Based upon our previous experience, the PPV2 verification plan called for the effects of each instruction to be verified using only one property per instruction, thus eliminating the verification decomposition often necessitated by other formal tools.

So, for each instruction, one property was written that described how an instruction passes down the pipeline and how it influences the signals and registers of the

implementation. The team then developed a high-level view of that property that correlates the implementation registers with the specification registers to check compliance between the two. In this high-level view, a typical arithmetic instruction would be described as follows:

```
foreach reg in 0..15:
  if (reg = dest) then
      next(reg_file(reg) = ISS_result(opcode, Ra, Rb)
  else
       next(reg_file(reg))  = reg_file(reg)
  end if;
 end foreach;
```

where the instruction word decoding is specified here:

```
a   =  IW[31:28]
Ra  =  reg_file(a)
b   =  IW[27:24]
Rb  =  reg_file(b)
dest=  IW[27:24]
res =  ISS_res(ADD_m,Ra,Rb)
```

Finally, the actual arithmetic operation is specified in a case statement:

```
ISS_res(opcode,Ra,Rb) :=
case opcode is
  when ADD_m => Ra + Rb;
  when SUB_m => Ra - Rb;
  when AND_m => Ra and Rb;
  …
```

As these definitions are very intuitive and readable, they can easily be reviewed and confirmed to reflect the intended specification.

The definition of the reg_file auxiliary function describes operations at the RT level, for example, forwarding and pipeline stages. This function can be considered as a mapping between the programmer-level view and signal-level view.

Property Debug

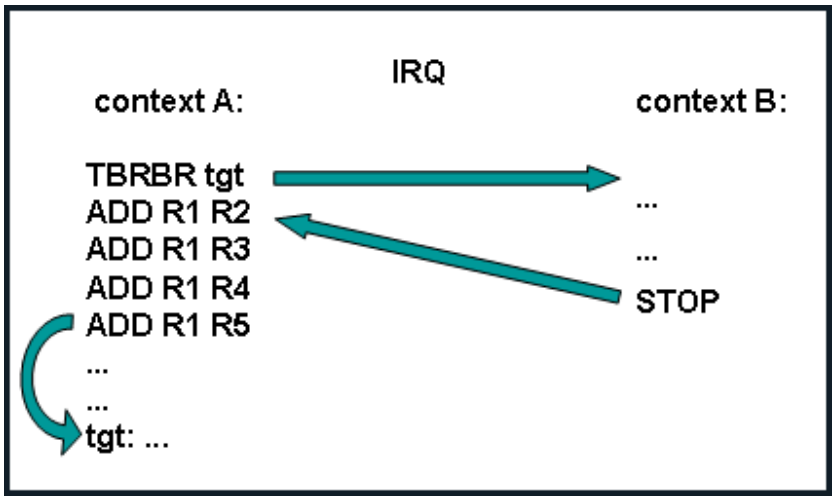
The team used the property checker to prove that the RT code always behaves as specified in the properties. In the case of failing proofs, the checker computes a waveform diagnostic counterexample - which is typically only a few cycles long – that shows where the RTL behavior deviates from the specified behavior. The team then used the debug and diagnosis environment to locate the error in the RT code or in the property.

An example of a complex behavior to be analyzed and debugged is that of a branch instruction, for which a static parameter configures execution of up to four delay slots. Upon receiving an interrupt, these slots must be stored, and their execution must be

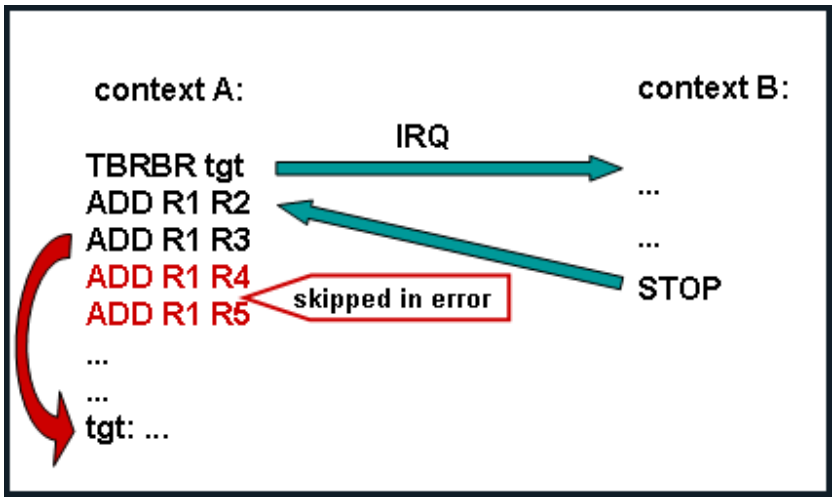
resumed after completion of the interrupt. This sequence must behave correctly in a wide range of configurations. The verification detected a configuration in which the slots were only partially executed, as outlined in figures 2 and 3.

Here we have an interaction of a “long“ branch (opcode TBRBR) with a context switch triggered by an interrupt (IRQ). The “long” branch has 4 delay slots. A bug in context switching during branch operation caused slots 3 and 4 to be “forgotten”, that is, never executed. The bug was discovered through a failing property, the RT description was corrected, and the specification ambiguity was eliminated.

Systematic and efficient detection of such “deep” bugs is difficult to impossible using simulation.



2. This figure shows the specified behavior of a delayed branch with interrupt.



3. The faulty delayed branch with interrupt skipped two instructions.

Certified Quality Level

Writing and proving properties according to the verification plan can result in a

reasonably high quality level. But, how “high” is “high”? And how do you objectively demonstrate this quality to the IP integrator? After all, any property checker can prove single properties exhaustively: if a property holds, the RTL will exhibit no behavior that contradicts the property. But “exhaustive verification” doesn’t imply “complete verification”. So, the essential question is: “Did I write enough properties and are they comprehensive enough to ensure correct functional operation under all possible input scenarios?” No manual or methodological approach can answer this question in a definitive manner.

We employed automatic completeness analysis to ensure that all output signals had been verified at all points in time under all possible input scenarios. This is the highest verification quality that can be achieved by any functional verification approach.

The input to the automatic completeness analysis is captured in a so-called completeness description using statements about the thoroughness of the verification. Take, for example, the following statement:

```
determined(program_counter);
```

It states that the program_counter must be verified by the property set for any possible input scenario, that is, any instruction sequence and any possible context switch sequence. If the check for this statement holds for the property set, there can be no situation in which the program_counter has an unexpected, unanticipated or unspecified value. This confirms 100% input scenario coverage and 100% output behaviour coverage for this signal.

To cover cases where output signals must hold valid values only under certain conditions, a conditional statement can be used:

```
if mem_write_o = 1 then
      determined(mem_data_o)
end if;
```

Such “determined” statements are specified in the completeness description for all output signals. The automatic completeness check confirms termination of the verification only when all “determined” statements are satisfied by the property set. Where any statement is violated, the completeness check detects the verification gap, and displays diagnosis information that enables the expansion or improvement of the property set to prevent error escape.

For example, the completeness analysis indicated that the effect on instruction words while stored in the context switch buffer had not yet been verified. Clearly, the instructions words were intended to remain unchanged. However, the team extended the properties to close this verification gap and detected that an error caused instruction words to be modified while stored in the context switch buffers. This rare phenomenon had not been considered in the specification, and had not been anticipated in the initial property set, and would almost certainly have escaped any functional verification using methods that do not employ completeness analysis. So, the automatic detection of such verification gaps is the key for an efficient, complete verification and highest IP quality.

After the successful completeness check, the completeness description gives a certified, objective and compact assessment of verification quality. In the case of PPv2, the completeness description is three pages long and can easily be reviewed by the IP

integrator.

Completeness analysis provides 100% input coverage, which ensures that there are no unstimulated errors; and 100% output behavior coverage, which ensures that there are no overlooked errors. Moreover, as shown in the example above, completeness analysis is really a specification/implementation co-verification that analytically identifies gaps and ambiguities in the specification, too.

[For a comprehensive discussion of verification completeness, please visit the Electronic Engineering Times website to view the article “Achieving completeness in IP functional verification”:

http://www.eetimes.com/showArticle.jhtml;jsessionid=IGZ2FP35N3ADWQSNDLPSKH0CJUNN2JVN?articleID=197005268]

When the final property set was confirmed as complete, it provably captured every possible behavior of the PPv2. At this point, the design was signed off as fully functionally verified. This is a clear termination criterion. Verification is finished when all properties are proven and the property set is complete – the IP is free of functional bugs.

Reliable Integration

The verification of the PPv2 relied upon certain assumptions about how the PPv2’s environment would interact with it. For example, an enforced context switch to the context that is already running leads to unspecified behavior. So, the verification was conducted under the assumption that such a situation never happens.

Such environmental assumptions are stated as formal assumptions in the property set and are the input for automatically generating HDL monitors that can be used as instrumentation for subsequent integration and system-level verification. The monitors themselves are complete – a by-product of the complete verification. That is, if at any point during the system-level verification the PPv2 is triggered in an illegal way, the monitors will detect this violation and flag it as an error, providing the data necessary to quickly detect and localize the violation. This increased observability, together with the completeness of the monitors, considerably eases and speeds the IP integrator’s chip verification.

Overall Results

Formal verification detected ten serious errors and seventeen issues leading to a redesign of the context switch logic; elimination of ambiguities in the specification; a much better understanding of the architecture; and precise integration requirements.

After elimination of errors and open issues, a suite of 40 proven properties described unambiguously the PPv2’s cycle accurate behavior on fewer than 40 easy-to-read pages. The proofs verified that the VDHL correctly implemented the specification and that no functionality had been overlooked. The entire property suite runs overnight on a standard workstation, enabling fast verification of code changes.

After completing the formal verification, the RTL design was validated by a subcontractor - Informatik Centrum Dortmund - against the cycle and bit accurate C++ simulation model of the processor’s programming tool chain. A testbench and a set of about 1,000 test case templates was developed and employed as a source to randomly generate over 18 million lines of assembly code in order to validate the equivalence of both models.

Further tests were developed for chip integration, regression tests with existing firmware

were performed, and new firmware was developed and verified by intensive emulation. All this was done in different environments by different, independent teams. Not a single functional error was found in the processor's design during any of these tests. The PPv2 has been integrated into four SoC and no bugs have since been found.

How well does the methodology meet the verification objectives?

The project met all of its stated objectives. It achieved error-free functional operation of the PPv2 across all possible configurations. It resulted in a considerably improved specification. It delivered a verification report that certifies the achieved IP quality and enables rapid IP quality assessment. It delivered a precise description of integration conditions. And it did all of this with high productivity: the total verification effort was 4 engineer-months, about 40% less than that in the simulation-based verification of PPv1. However, the largest effort reduction lies in the fact that no IP revision or redesign has been necessary since release. It was right-first-time. All of which translates into a massive reduction in integration effort, cost and risk.

What does this mean for the IP industry?

The results demonstrated on the processor verification are typical of those obtained on a broad range of digital IP. The underlying principle of complete formal verification has been applied successfully to ensure error-free operation of, for example, memory controllers, bus interfaces, communication peripherals, multi-media components as well as error correction modules. Moreover, this high quality was achieved in less time, with less effort and at less expense than is possible with other verification approaches.

This means that the IP industry can eliminate one of the major causes of chip-level IP integration and verification problems – the integrity of the IP blocks themselves. This can change both internal IP deployment and the third-party IP business – because the IP provider can deliver highest certified IP quality without breaking the bank.