Exploiting "Architecture for Verification" to Streamline the Verification Process

Dave Whipp, NVIDIA

[dac09@dave.whipp.name]

Abstract

A typical hardware development flow starts the verification process concurrently with RTL, but the overall schedule becomes limited by the effort required to complete all the necessary verification tasks. Being the limiting factor, verification schedules become unpredictable, often resulting in slippage of the tapeout dates. This paper looks at ways to restructure the flow to complete a significant part of this effort during the architectural phase of the project, prior to the start of RTL. This front-loading of the schedule allows a smaller verification team to complete the process with a tighter schedule.

Categories and Subject Descriptors: B.5.2

[Hardware]: RTL – Design Aids – Verification

General Terms: verification, management

Keywords: verification, executable specification, ESL

1   A Verification Architect

What is the goal of Verification Engineer? Most people faced with this question – I use it for job interviews – will frame the answer in terms of the correct operation of the product or the fidelity of the design to its specification. This is part of the answer but, I believe it misses at least one important element: timeliness. The success or failure of a product can be decided by its launch date. Weeks, and days, do make a difference in consumer products. When I describe my goals as a Verification Architect I focus not on the correctness of the product (which I regard as "a given"), but on the workflow that achieves it. My aim is therefore to construct verification strategies where:

- Verification tasks do not appear on the critical path of the project.
- Estimates of task durations are as accurate as possible.

A task that is on the project's critical path contributes directly to the time between the start of the design and the start of volume manufacturing. The critical path should be dominated by design tasks, not verification. Examples of design tasks are: writing RTL, debugging failures, and optimizing the design to achieve timing, area, and power goals. Verification tasks include: creating and debugging tests and test benches, triaging test failures, and achieving coverage goals.

It is not my intent to disparage the job of verification engineers. I focus on verification efficiency because I am a verifier. If I were a designer then I would seek to minimize the cost of design, and to ensure that design tasks do not appear on the critical path.

A strategy that is used to improve efficiency for both Verifiers and Designers is to attempt to front-load the overall project schedule by moving effort to the product architecture phase.

2   A Hardware Development Flow

The major tasks in chip development flow can be grouped by who is responsible for them. The major groups are "architecture", "design", and "verification". Although different companies may have marginally different flows, I believe that a typical breakdown of the tasks is similar to that shown in table 1.

Table 1: Task Responsibilities in a Traditional Flow

Architecture	Design	Verification
Comprehensive Paper Specification	Write RTL	[UTF Model]
	Debug RTL	Write Testbench(es)
ISS Model	Power, area, and timing closure	Write Test Scenarios
[UTF model]		Debug Tests
	Assertions	Triage RTL failures
		Assertions
		Functional Coverage
		Tune Constraints

The work done by the design group is not the focus of this paper. Their work consists primarily of writing RTL and then optimizing it to achieve the power, area, and timing goals of the module. Where a designer makes assumptions for correct behavior, we ask them to code those assumptions as assertions, and not simply as comments. Finally, we charge the schedule hit for debugging RTL failures as an RTL cost. This is balanced by a "triage" cost for the time spent by Verifiers to figure out the meaning of test failures.

The architecture group is responsible for defining the structure of the design. Major blocks are defined, and their functions described. The interfaces between the blocks are sketched out to some level of detail, typically focused on the performance of the interface (latency, throughput) and whether some standard bus (protocol) is to be used.

The work product of the architecture group almost always includes paper documentation, which will later be read and interpreted by the design and verification teams who are responsible for created the detailed design and affirming that this detailed design is indeed constant with the specification.

It is common for architects to create executable models of the proposed architecture. A common abstraction is the instruction set simulator ("ISS"), which defines the operations while avoiding implementation details. An ISS is not limited to traditional CPU architectures: for example, a hardware block that acts as a network switch may have its routing behavior defined as an ISS.

Less frequently, architects may create a detailed transaction level model of the chip. Such a model will provide a detailed representation of each module in the design, along with its interfaces. A common framework for such models is the untimed functional "UTF" abstraction of SystemC [1]. Ideally for verification, such a model will be interface-accurate and usable as a placeholder in the design for actual RTL units: co-simulating with other units for faster simulation (or before the RTL for the unit is written).

Unfortunately, although detailed executable models are not always created by architects, they are frequently needed. It then falls to the verification team to create the models as an interpretation of the written specification. This has two consequences: first, the model writer may misinterpret the specification; second, the architect doesn't get immediate feedback from running code that is a concrete representation of the specification.

The lesson to learn from this is that an efficient flow requires appropriate architectural representation. I will show that having architects just write transaction-models is insufficient: there are many other forms of executable work-product that we can ask our architects to create.

3 Executable Specifications

3.1 C Models

One thing that both ISS and UTF models have in common is that they are simulation models. The way they are used is to create a

stimulus (possibly software) and run the code to see what happens. Although some progress is being made with "sequential equivalence" checking, such tools are not yet ready to replace traditional verification approaches.

Being simulation models, the code is actually an implementation, not a pure specification. Often they are coded in such a way as to improve simulation performance but decrease their usability as a specification. This weakness is a result of the fact that the standard language we have, today, for writing such models is C++.

3.2 Functional Coverage

A C model may be used by the verification team as a reference model. A test is run on both the RTL and the model and the resulting behaviors compared. Depending on the abstraction of the model, the verification effort needed to achieve a reliable comparison can be non-trivial.

When the model is used in this way, the model provides not only a behavioral reference, but also a definition of functional coverage. If a C model defines all the behaviors defined by the architects, then structural coverage of the model is analogous to functional coverage of the design.

When a feature is omitted from an implementation, structural coverage may still achieve 100%. However, if that feature exists in the C model, then structural coverage of that model will show that the test suite did not exercise it. Of course, the feature might have been overlooked even by the architecture group: but if the model is also used for software development then we can have some confidence that its omission will be noticed.

Even if the correlation of model-coverage with functional coverage is not perfect, we can still benefit from the concept. Sometimes we ask: "what 5% of tests should we run for our nightly regression testing?" We have found that a Pareto analysis of structural coverage of the C model is able to answer this question in much less time than it would take to run our test suite on the RTL.

3.3 Leveraging Validation Tests

A simulation model is not the only work-product of architecture that can be directly used for verification. Architects create "validation tests" to check that the behavior of the models matches their expectation. Whilst such tests are often reused for verification, their value can be enhanced if they are able to be retargeted to hit implementation corner cases.

One approach is to create validation tests as self-checking directed tests, and then run the same test multiple times with different "irritators" applied to the design. Examples include context switching between multiple tests, injection of soft errors, and randomizing backpressure from FIFOs. Depending on the design, many other "irritations" may be applied.

Another approach is to construct the validation test suite by generating self-checking directed tests. One way to achieve this

[2] is to define the test space as a graph: random walks of that graph create the tests. The graph itself becomes an executable model of the design, albeit one that is very different from a traditional simulation model. The power of the approach stems from the ability to compose graphs that express difference aspects of the verification space: architects provide a graph for validation; vierifiers add graphs for the implementation corner cases.

3.4   Properties and Assertions

Assertion based verification methodologies often partition assertions into two groups: those written by designers, and those written verifiers [3]. Designers are asked to treat assertions as "executable comments" that document assumptions they are making. Verifiers are asked to create assertions that reflect requirements passed down from the specification.

In the context of an executable specification, it seems absurd to ask verifiers to write architectural assertions. The source of such assertions is the architect, not the verifier, and adding a manual interpretation step into the flow here does not add value.

Today's (temporal) assertion languages (and tools) are not designed to be used with C models. Both SVA and PSL are conceived in terms of sequential regular expressions (SEREs) in the cycle domain. It is possible to map these to transactions and algorithms, but doing so introduces additional complexity that makes it less likely to be adopted. The problem is compounded by the fact that not all architects use Linux as their primary operating system.

Instead, we use an in-house language [4] that enables assertions to be written at higher levels of abstraction, which are then translated to C++, OVL, and SVA for use in different simulation contexts. This is an area that feels ripe for exploration by EDA vendors.

3.5   Interface Definition Language

The UTF C Model, the testbench, and the design all share the same interfaces. It is natural, therefore, to define these interfaces just once, using a formal interface definition language such as IP-XACT [5], and to use code generators to create the RTL and C code that implements them. Indeed, it is only through the use of such a language that it is possible to take architectural assertions and map them to the design.

An interface definition language is the basis of structural models. These define not just the interface signals, but also the modules to which the interfaces connect, and the cycle-level protocols used to mediate the flow of information. From this information it is possible to generate many of the components of the lower layers of a testbench, such as BFMs that translate between the transaction-domain semantics of the higher layers of the testbench (section 3.6) and the cycle-domain semantics of the DUT itself. All that is left, for the testbench author, is to ensure the correct hookup of clocks and resets (plus, perhaps, implementation-specific device ports such as for DFT).

3.6   Test Benches

Comprehensive testing of a transaction level C model demands a testbench capable of interacting with multiple concurrent transaction level interfaces. This is the same requirement as exists for a transaction level testbench for RTL verification. A properly constructed testbench should be reusable from validation to verification.

If the transaction layer of the testbench is written to test the architectural UTF model, and the signal/command layer can be derived from a formal interface definition (section 3.5), then most of the major components of the RTL testbench are derived as work products of architecture.

4   An Improved HW Creation Flow

The task breakdown of table 1 showed that if the work product of the architecture group is limited to just a paper spec (and perhaps a C model or two) then the majority of the downstream tasks fall to the verification group. Table 2 shows how an aggressive embrace of a more general concept of "executable specification" acts to change the distribution of tasks.

Table 2: Task Responsibilities in Enhanced Flow

Architecture	Design	Verification
Minimal Paper Specification	Write RTL	Testbench hookup (clocks and resets)
ISS Model	Debug RTL	Write Test Scenarios
UTF model	Power, area, and timing closure	Triage RTL Failures
Formal Interface Definitions	Assertions	Tune Tests
Assertions and testpoints		
Directed-Test (Graph)		
Transaction Testbench Layer		
Debug Tests		

From the selfish perspective of the verifier it is easy to say that this flow would be preferable. By transferring the majority of the tasks to architecture, the verifier is free to concentrate on the actual goal of verification. But it is not sufficient to improve the workload of the verifier: it must also be advantageous to the architecture group (who are probably not staffed for extra work) and to the efficiency of the overall HW creation flow.

Executable work-products enable architects to better understand the architecture, without depending on the long feedback paths that result from depending upon verification engineers to write models and validate the architectural assumptions. Furthermore, they are creating tangible proof that the architecture works. If the application is video or graphics, then they can see frames (pictures) being created. In other domains the results may be less obvious to outsiders, but no less gratifying to the creators.

The overall flow benefits when bugs can be found earlier in the development process. A model created by an architect is validated as part of its creation, and then stressed by the software group who use it as a platform for their part of the project. The model, validated by both SW and Architecture, can then be trusted by the verification group who then assure that the design is equivalent to the model.

Efficiency is further enhanced when tests and testbenches from architecture are reused for verification. If it is a given that these constructs are needed for architecture, then the additional cost to make them reusable is most likely a cost worth paying.

5   Challenges

When we think of an executable specification, we usually think of a simulation model, written in C++. Such models are useful, but insufficient. Structural coverage of abstract models is a good source of functional coverage, but today's tools do not enable us to fully exploit it. Not only is there limited traceability from lines of code in the model to the equivalent lines in the RTL; but also there is much code in the C++ model that exists only to support its use as a simulator.

When we attempt to stretch a single model between multiple organizations, we find that we need to make tradeoffs. Software groups may want fast models that run on Windows, whereas the Verification group is most likely Linux-centric – and would happily trade performance for a greater fidelity to the micro-architecture.

Initially, the architects reject the idea of adding additional work: instead of just writing a specification document, they are now being asked to write models, tests, and assertions. They feel that these models are not for their benefit: they are for the benefit of verification engineers (who would otherwise be tasked with the work) and for the software group. It is only later that we see attitudes change, and the benefits to both the architects and the project as a whole become apparent.

6   Architecture For Verification

The goals of Verification in the architectural phase of the project are twofold:

- Eliminate unnecessary complexity that would increase verification burden.
- Ensure that the work products of the Architects are directly usable in the verification flow.

Unnecessary complexity is most often found in features that exist to increase performance in areas that are not critical to the product's success. Performance enhancements that are achieved by additional coupling between micro-architectural features lead to a significant increase in the number of corner cases: not only must the features be verified in isolation, but also the optimized interactions must also be verified.

To be most useful, the work products of the architecture group must be more than a set of documents. A multitude of different aspects of executable specification are needed. In addition to ISS and UTF C models, we find value in reusable tests, testbenches, assertions, and various forms of structural definition.

If we understand how the different models will be used by all the stakeholders, then we know how to prioritize the resources to create them. If a model is most useful for verification, then it may make sense to have verification engineers work more closely with architects, and less closely with RTL designers. If the model is to be used by the software authors, then it may make sense to have software engineers work closely with the architects on that model. And, when the model is to be used by all three groups: then all three must work with each other to maximize the overall efficiency of the flow.

7   References

1. SystemC: http://systemc.org
2. Adnan Hamid. "Hope is Not a Verification Strategy – http://www.designcon.com/infovault/paper.asp?PAPER_ID=323 (DesignCon 2008)
3. Harry Foster, Adam Krolnik, David Lacey – "Assertion-Based Design" section 1.4.3 (page 19).
4. Dave Whipp, "Transaction Assertions in an Interface Definition Language" (DesignCon 2008) – http://dave.whipp.name/dv/designcon2008_paper.doc
5. IP-XACT: http://spiritconsortium.org/home