

THE UNIFIED VERIFICATION METHODOLOGY

TABLE OF CONTENTS

1	Introduction	1
2	Key concepts	1
3	Platform requirements	6
4	SoC methodology	7
5	Control-based digital subsystems	17
6	Algorithmic-based digital subsystem	28
7	Analog subsystem	30
8	Evolutionary migration strategy	32
	Glossary	33

FIGURES

Figure 1	The functional virtual prototype	1
Figure 2	Developing the implementation-level FVP	2
Figure 3	Creating FVP TLMs	2
Figure 4	Replacing a FVP TLM with an implementation-level block	3
Figure 5	Linked transactions	4
Figure 6	Assertion types	5
Figure 7	Acceleration-on-Demand	6
Figure 8	FVP with signal level interface	11
Figure 9	Unified verification methodology flow	12
Figure 10	SoC architecture stage flow	12
Figure 11	SoC integration flow	14
Figure 12	Software-based system verification flow	15
Figure 13	Emulation-based system verification flow	16
Figure 14	Hardware-based system verification flow	16
Figure 15	Control-digital testbench development	19
Figure 16	Top-down subsystem testbench development	21
Figure 17	Test development environment	21
Figure 18	Top-down block testbench development	22
Figure 19	Bottom-up subsystem testbench	23
Figure 20	Top-down methodology flow	25
Figure 21	Bottom-up methodology flow	27
Figure 22	Algorithmic TLM development	29
Figure 23	Algorithmic digital methodology flow	30

TABLES

Table 1	Transaction taxonomy	4
Table 2	Coverage types	5

1 INTRODUCTION

The unified verification methodology addresses the most critical verification challenges, while maximizing overall speed and efficiency. The methodology is centered on the creation and use of a transaction-level golden representation of the design and verification environment called the functional virtual prototype (FVP). The methodology encompasses all phases of the verification process and crosses all design domains. Utilizing the unified verification methodology will enable development teams to attain their verification goals on time.

While focusing on the verification of systems on chip (SoCs) the methodology also encompasses verification of individual subsystems. Large application-specific digital or analog designs are often first developed as standalone components and later used as SoC subsystems. The unified verification methodology can be applied in whole to a SoC design or in parts for more application-specific designs. Different designs and design teams will emphasize different aspects of a methodology. The unified verification methodology will produce the greatest gains in speed and efficiency when used in a complete top-down integrated manner. It is understood that a complete top-down flow may not always be feasible for a number of different reasons. Thus the methodology is flexible in providing for both top-down and bottom-up approaches to developing subsystems while still providing an efficient top-down verification methodology.

While it may be impractical for verification teams to move directly to a top-down unified methodology from their existing methodology, this methodology provides a standard for verification teams to work towards. Readers are encouraged to read the entire document, but if the readers focus is solely the verification of a subsystem they can skip to those individual sections of the methodology. Readers should also note that the unified verification methodology as described in this paper is targeted at custom IC and standard ASIC development. Many of the issues described and practices presented are applicable to verification of programmable devices such as FPGAs but some areas differ. Verification of programmable devices will be addressed in a future revision of this paper.

The unified verification methodology is first presented in the Cadence whitepaper "It's About Time: Requirements for the Functional Verification of Nanometer-scale SoCs." The paper presents the drivers of functional verification today and identifies the fragmentation that exists throughout functional verification. The unified verification methodology directly addresses this fragmentation. This paper discusses the unified verification methodology in detail. The first sections of the paper provide an overview of the methodology, describes several key concepts, and presents the requirements for a verification platform to support the methodology. The middle sections detail the methodology for verification of a SoC along with the individual subsystems. The final section details a migration path to the unified verification methodology.

2 KEY CONCEPTS

2.1 FUNCTIONAL VIRTUAL PROTOTYPE

A functional virtual prototype (FVP) is a golden functional representation of the complete SoC and its testbench.

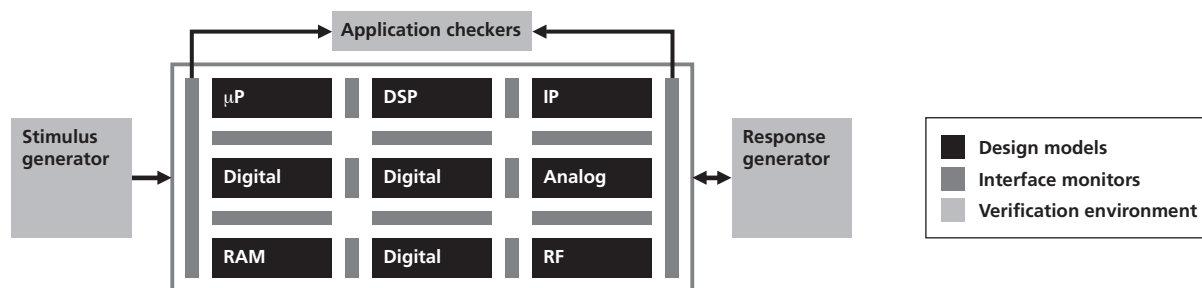


Figure 1: The functional virtual prototype

The FVP exists in many different levels of abstraction throughout the design. The transaction-level FVP is created early in the design process by the SoC verification team working closely with the architects. The FVP is termed transaction-level because the communications backplane that connects the functional models transfers information in the form of transactions. As development progresses implementation-level models provided by the subsystem teams replace transaction-level models creating a FVP with mixed levels of abstraction. During this process the communications backplane of the FVP changes to transferring information in the form of signals. Once all of the transaction-level models are replaced with implementation blocks the FVP is considered an implementation-level FVP.

The FVP unifies the verification of different design domains and processes by providing a source of common reference models as show in diagram below. The design in the diagram begins with four blocks represented first as a transaction-level FVP. Each block is developed standalone using the TLM from the FVP in the testbench. The implementation-level blocks then replace the TLMs in the FVP creating the final implementation-level FVP. More information can be found in the application note [“Reusing the FVP for Testbench Components.”](#)

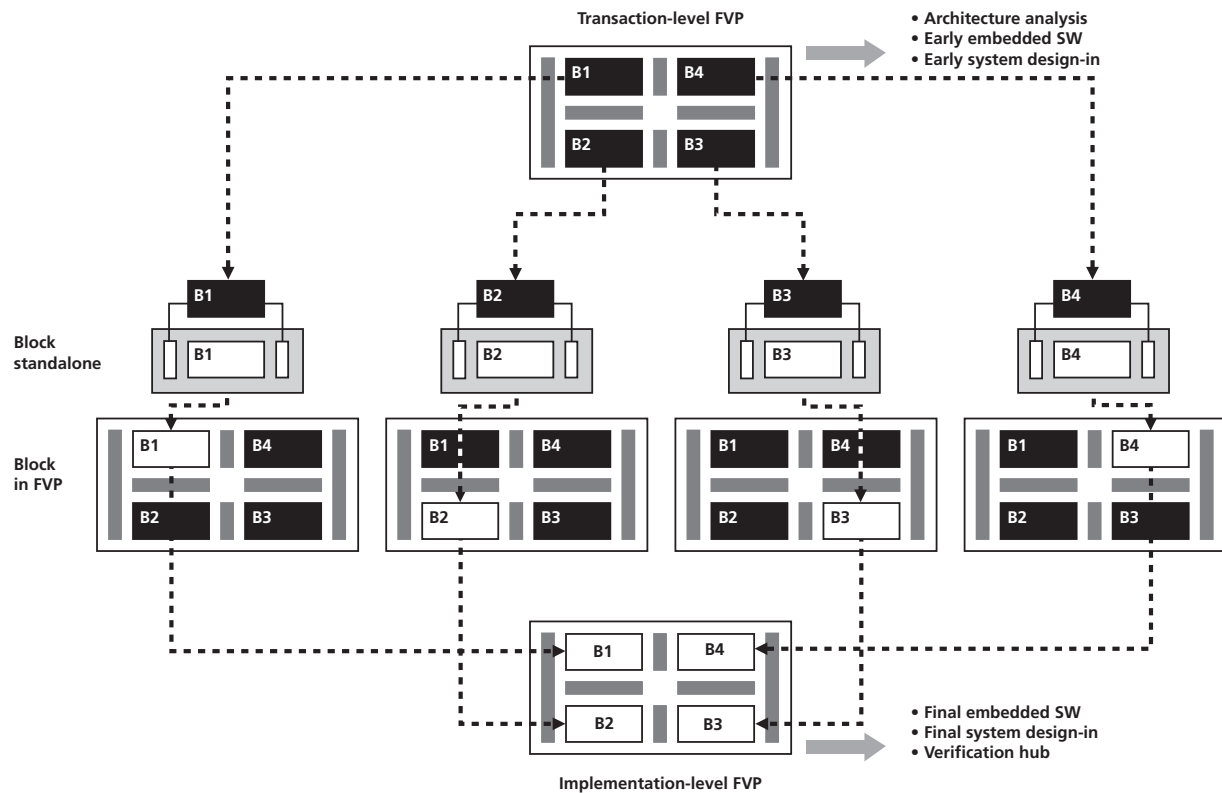


Figure 2: Developing the implementation-level FVP

The transaction-level FVP consists of transaction-level models (TLMs), transaction-level testbench components and mixed-signal instrumentation. TLMs model each of the functional blocks within the design. The initial degree of fidelity of the TLM to the implementation is dependent on the needs of the software and system development teams. If the TLM is to be reused for subsystem verification then the implementation fidelity is also dependent on the verification team’s needs. In most cases the TLMs are behavioral code wrapped in a transaction-level shell for interfacing to other blocks as shown below. More information about TLMs can be found in the application note [“Transaction-Level Modeling.”](#)

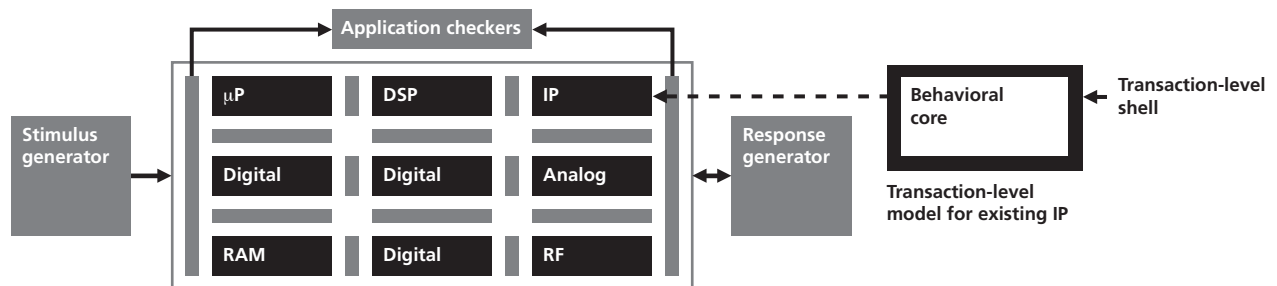


Figure 3: Creating FVP TLMs

Each subsystem has its own TLM defined at the SoC partition. The subsystem team may further partition the subsystem TLM into implementation-level block TLMs to be used for block-level verification components.

The FVP also contains testbench components including stimulus generators, response checkers, interface monitors and mixed-signal instrumentation. The FVP includes these components to provide a comparison environment for subsystems to develop against and to allow for testbench reuse. The testbench components are all at the transaction-level providing high performance. Mixed-signal instrumentation includes specialized stimulus and response analysis monitors used to interpret mixed signaling. A test suite is included with the FVP. This test suite is run on the FVP as each subsystem implementation is integrated into the SoC. The subsystem teams use this environment to test individual blocks as they are developed identifying integration issues with other subsystems early in the process.

The process of integrating implementation-level blocks into a transaction-level FVP is facilitated by the use of transactors, which translate information between the transaction-level and the signal-level. An example of using transactors to integrate a single block into the FVP shown in *Figure 4*. In this example the implementation-level block is surrounded by transactors along with interface monitors at the signal-level. This entire group of elements replaces the Digital TLM in the FVP allowing for a mixed-level simulation.

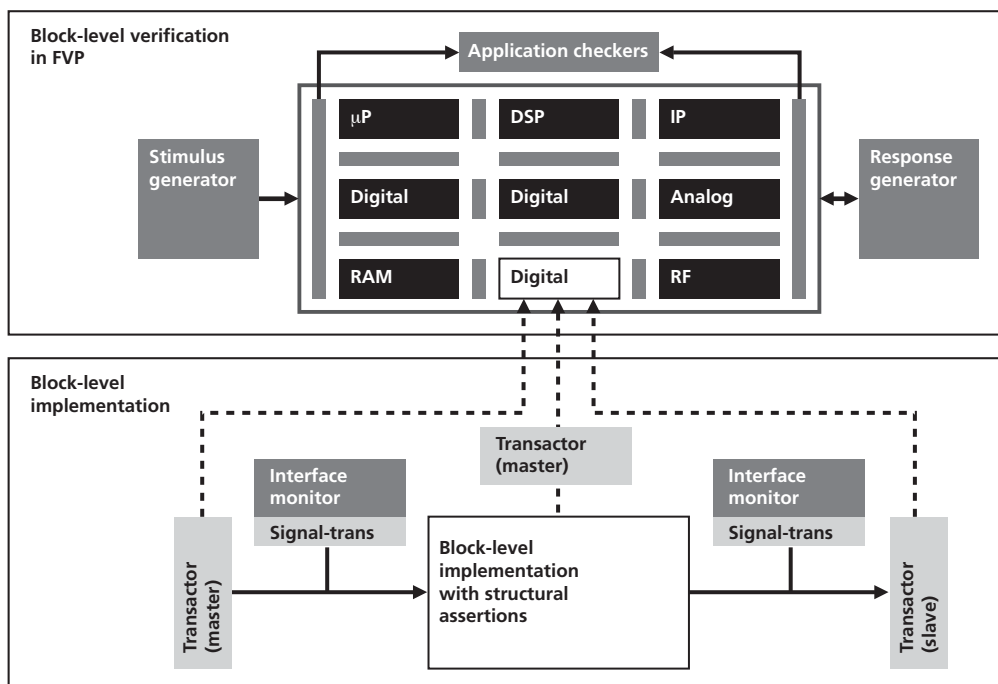


Figure 4: Replacing a FVP TLM with an implementation-level block

The FVP serves several critical roles in the methodology:

- An unambiguous executable specification
- A fast executable model for early embedded software development
- A model for performing architectural tradeoffs
- An early handoff vehicle to system development teams
- The reference for defining transaction coverage requirements
- The source for subsystem-level golden reference models
- A golden top-level verification environment and integration vehicle

2.2 TRANSACTIONS

Transactions improve the efficiency and performance of the unified verification methodology by raising the level of abstraction from the signal level. A transaction is the basic representation for the exchange of information between two blocks. Put simply, a transaction is the representation of a collection of signals and signal transitions between two blocks in a form that is easily understood by the test writer and debugger. A transaction can be as simple as a single data write operation or a sequences of transactions can be linked together to represent a complex transaction such as transfer of an IP packet. The following diagram shows a simple basic transaction data transfer (represented as B) linked together to form a frame, further linked together to form a packet and finally a session.

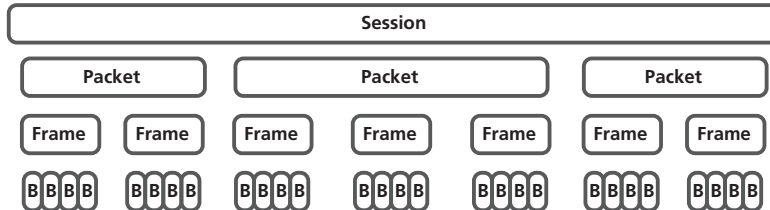


Figure 5: Linked transactions

Transactions are used throughout the unified verification methodology. A transaction taxonomy is created early in the development process to assure the common use of transaction types throughout the process. This provides for re-use of testbench elements and analysis software with a common transaction reference. The transaction taxonomy specifies the layers of transactions from the simplest building block to the most complex protocol. The transaction taxonomy is created by defining the appropriate levels the test writers and debuggers will be most efficient operating at. All models and software programs are written to these levels of abstraction depending on their use. The following table gives the transaction taxonomy for the previous diagrams transactions.

Level	Data unit	Operations	Fields
Interface	Byte	Send, receive, gap	Bits
Unit	Frame	Assemble, segment, address, switch	Preamble, data, FCS
Feature	Packet	Encapsule, retry, ack, route	Header, address, data
Application	Session	Initiate, transmit, complete	Stream

Table 1: Transaction taxonomy

The use of transactions in the unified verification methodology improves the speed and efficiency of the verification process in several ways:

- Provides increased simulation performance for the transaction-based FVP
- Allows the test writer to create tests in a more efficient manner by removing the details of low-level signaling
- Simplifies the debug process by presenting the engineer information in a manner that is easy to interpret.
- Provides increased simulation performance for hardware-based acceleration
- Allows easy collection and analysis of interface-level coverage information

More information on transactions can be found in the application note [“Verification Using Transactions.”](#)

2.3 ASSERTIONS

Assertions are used to capture the intent of the architect or implementer as they develop the system. Verification software can be used to verify the assertions either in a dynamic manner using simulation or in a static manner with formal mathematical techniques. Assertions are created in the unified verification methodology whenever design or architecture information is captured. These assertions are then used throughout the verification process to efficiently verify the design. There are three types of assertions used in the unified verification methodology. First, application assertions are used to prove architectural properties such as fairness and deadlocks. Second, interface assertions are used to check the protocol of interfaces between blocks. Finally, structural assertions are used to verify low-level internal structures within an implementation such as FIFO overflow or incorrect FSM transitions.

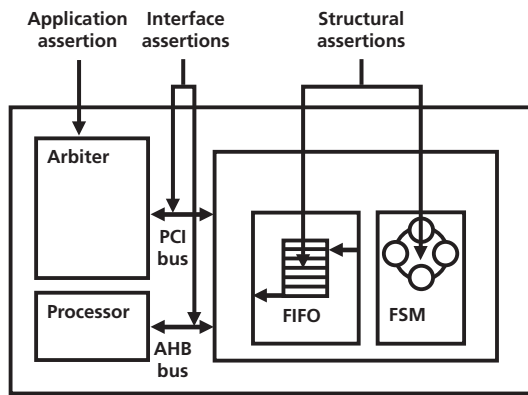


Figure 6: Assertion types

The use of assertions in the unified verification methodology improves the speed and efficiency of the verification process in several ways:

- Speed the time to locating difficult bugs by identifying where in a design the bug first appears
- Automate the instrumentation of monitors and checkers within the design
- Quickly identify when stimulus or interfacing blocks are not behaving as the implementer intended
- Identify bugs that did not propagate to checkers or monitors
- Protocol violations can be detected even if they do not cause a functional error
- Provide feedback to stimulus generators to modify their operation as the test progresses

More information on assertions can be found in the application note [“The Role of Assertions.”](#)

2.4 COVERAGE

Verification coverage techniques increase the efficiency of the verification team by providing feedback on the effectiveness of the verification processes. Coverage cannot determine when the design has been verified but it can indicate areas for more concentration. Coverage information improves efficiency by identifying areas of the design that have not been stimulated, by identifying tests that are not verifying what is intended, and in some cases helps to identify incorrect functionality.

The unified verification methodology uses four different types of coverage information. Application coverage identifies if specific high-level features of the design have been stimulated such as automatic data retries. Interface coverage identifies the types and sequences of stimulus and responses at the interfaces of the DUV to identify what sequences and correlated responses have not yet been verified. Structural coverage monitors the operation of low-level structural elements such as FIFOs and FSMs to identify what parts of the implementation have been verified. Finally, Code coverage also can be used to identify what areas of code have been stimulated. More information on coverage can be found in the application note [“Using Coverage.”](#)

Coverage type	When defined	When measured	Examples
Application	Architecture definition	System modeling and System verification	Auto retry, cache hit
Interface	Implementation definition	After block tests and subsystem tests	Packet types instruction sequences
Structural	Micro-architecture	After block tests	FSM states and arcs FIFO thresholds
Code	Coding	After block tests	Statement, expressions

Table 2: Coverage types

2.5 ACCELERATION

Accelerating the speed of simulation allows more testing to be completed in a shorter period of time. The key to using simulation acceleration in the unified verification methodology is choosing the most efficient method of simulation for the test you are running. As the verification process moves from short unit-level tests to longer subsystem-based tests it is important to monitor the performance and debug times required for the tests. Hardware acceleration should be used once the test process has reached a stage where the runtime is the dominant performance factor. Technologies such as Acceleration-on-Demand provide the user the ability to switch from simulation based testing to hardware accelerated testing using the same environment for development and debug.

An example of this decision is shown in *Figure 7*. The standard runtime for tests of an average size design block is shown on the horizontal axis. The vertical axis shows the total test time including compile time, run time, and debug time for a test. It can be seen in the diagram that for short tests simulation is the fastest method. Once the test length reaches around 70 minutes acceleration is the fastest method in this example. This is the point where Acceleration-on-Demand is used. This example was for a small block design with a non-synthesizeable testbench. Each team should do a similar analysis based on their specific design size, testbench performance, and debug times to determine the correct time to accelerate their simulations. More information on acceleration can be found in the application note [“Incisive Accelerated Co-Simulation.”](#)

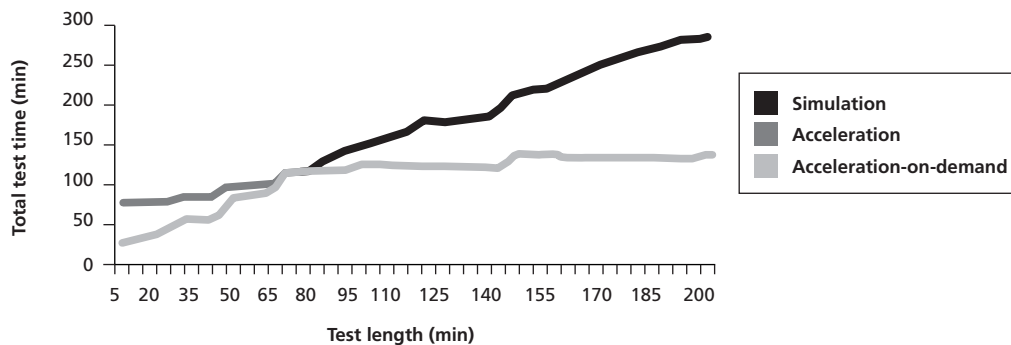


Figure 7: Acceleration-on-Demand

Acceleration is also used in the unified verification methodology to speed regression testing. Often small changes to the design or test environment can have unwanted and unknown effects on other parts of the system. Development teams set up a regression environment to verify that changes in the design or environment have not caused unwanted effects. It is important that the development teams receive confidence in their changes in the shortest possible turnaround time. Regression testing is often performed on large server farms running jobs in parallel. The time to complete the regression is dependent on the number of tests, length of the individual tests, the number of servers in the farm and the simulation speed. Once the number of tests and length of tests causes the total regression time to exceed the acceptable turnaround time acceleration should be used. Acceleration reduces the runtime of longer tests and allows large groups of shorter tests to be run in less total time than before.

Finally acceleration is used in the unified verification methodology to provide the necessary speed improvements to facilitate system verification. Acceleration can be used in a simulation based system verification environment to simulate large system integrations with embedded software. Acceleration is also at the heart of emulation based system verification. Accelerating the design in an emulation environment allows connection of the design to real-world stimulus and instrumentation while running and developing the system software.

3 PLATFORM REQUIREMENTS

The unified verification methodology is implemented by the software and processes provided in a verification platform. The platform provides the infrastructure and support for verifying complex systems in an efficient manner. Using a common platform across projects and throughout the design chain promotes efficiency and speeds the overall development of products. There are several important requirements of any verification platform that supports the unified verification methodology.

3.1 SINGLE-KERNEL ARCHITECTURE

The single-kernel architecture must be heterogeneous with native support for all standard design and verification languages. The design and verification languages used in the platform must be based on industry standards with standard APIs.

- Native Verilog® and VHDL. Most designers will continue to use HDLs for specifying their designs and to perform unit-level verification. Verification engineers may choose to use HDLs for unit testing and for developing parts of low-level transactors. The platform must support extensions to these languages as they become industry standards
- Native SystemC®. SystemC is a set of industry standard C/C++ libraries that supports highly efficient transaction-level modeling, RTL specification, and verification extensions for test generation. Most complex verification environments are based on C/C++ object-oriented languages. SystemC provides a set of industry standard libraries for developing these complex environments in a reusable fashion. The vast majority of embedded software for SoCs is written in C/C++ based languages, which makes a C/C++ based language a requirement
- Native Verilog-AMS and VHDL-AMS. Most nanometer scale ICs will include analog/mixed-signal/RF units, the verification environment must support native kernel support for the industry standard Verilog-AMS and VHDL-AMS languages
- Native property specification language (PSL). The platform must provide support for new standard PSL, based on the IBM Sugar assertion language. A single-kernel implementation insures the lowest performance overhead for using assertions and enables a single compilation process

3.2 TRANSACTION-LEVEL SUPPORT

The platform must support the use of transaction-level abstraction throughout. The platform must provide for the abstraction of information to the transaction-level to simplify test development, support reusable testbench elements and improve performance throughout the verification process.

3.3 ACCELERATION-ON-DEMAND

The platform must provide Acceleration-on-Demand. The platform must support the ability to increase simulation performance using hardware acceleration with minimal impact to the process.

3.4 HIGH-SPEED, UNIFIED TEST GENERATION

The platform must provide high-speed unified test generation.

- The platform must support reuse throughout the development process. The platform must enable the reuse of verification models and testbench components between process steps and between design domains. Common reusable verification models must span all of the verification stages and different design domains
- The platform must provide a unified development environment. The user interface to all software used within the platform must be common so that retraining is not required. Debug and analysis information must be presented in an efficient and common method. The exchange of information between software applications must be easily facilitated by the platform
- The platform must support the use of assertions throughout. The platform must provide for the development of assertions at the time when the designers or architects intent is formed and must provide for the use of these assertions throughout the rest of the process
- The platform must support the use of coverage information throughout. The platform must provide for the collection and analysis of coverage information at the application, interface, and structural levels throughout the verification process

4 SoC METHODOLOGY

Today, the development of SoCs often encompasses several different design teams. The SoC development team often is responsible only for architecture, integration, and system test of the SoC. The individual subsystems are either developed by different teams or vendors or they are reused from existing designs. The unified verification methodology begins with the SoC development team as they architect the SoC. The SoC development team has four main concerns at the start.

- Handoff and updating of subsystem specifications
- Providing the software team with early access to the design to begin development
- Providing early design and implementation details to the system design-in team
- Creation of a system-level test environment for easy integration

The unified verification methodology addresses these concerns with the use of the functional virtual prototype (FVP). Once completed the FVP is provided to each of the subsystem teams for use in verification of the subsystem. The FVP is an unambiguous executable specification that allows the SoC team to handoff and maintain updates of the subsystem specification in an efficient unifying manner. The FVP is a fast transaction-level model allowing for early software and system design-in development. The FVP contains stimulus generators, interface monitors, and response checkers that are used to verify the subsystems interface correctly as they are being developed speeding the integration and system test processes.

Once the subsystem teams have been provided the FVP, they begin to develop the individual subsystems. The fastest and most efficient method for development of most subsystems is to continue using the FVP in a top-down manner, reusing the FVP for block and multi-block verification. It may not always be feasible to use a top-down verification methodology for subsystem development; the teams may choose to develop in a more sequential bottom-up environment with separate dedicated testbenches. The Unified Verification methodology supports the evolution from a bottom-up to the more efficient top-down methodologies through extensive reuse and advanced verification techniques. Whichever method is chosen the subsystem is verified inside the FVP before it is handed off to the SoC team for integration. The transaction-level model is removed from the FVP and the implementation model is used in its place. The same suite of tests is run and verified with this implementation FVP before the subsystem is integrated into the SoC.

The final stage of the unified verification methodology is final system verification. This stage replaces the models and stimulus generators used to verify the SoC with the real-world environment. Emulators and hardware prototypes are two of the methods often used for system verification. The increased speed and performance provided by these methods allows for more complete software integration with the SoC. The interfacing options provided by these methods also allows for connection to real world stimulus and debugging equipment.

4.1 THE FUNCTIONAL VIRTUAL PROTOTYPE

The FVP is developed by the SoC verification team early in the development process. It may be impractical to expect the team to complete a detailed reference model of a complex SoC in a short period of time. It is important to understand the level of fidelity required for modeling at the SoC level and match the requirements to the time and resources available. There is a distinct difference between the accuracy of a model and the fidelity of the model. Accuracy describes the functional equivalency of the model to the implementation. Fidelity describes how close the model represents the implementation. The FVP must always be an accurate representation of the design, but the degree of fidelity may change as the development process progresses. There are two factors in determining the required fidelity for the FVP, the needs of the software development team and the needs of the integration test teams. The level of fidelity necessary for software development is highly dependent on the type of SoC being developed:

- Service processor application. In these SoCs the processor is used for start-up, basic maintenance and statistic collection. Software development for these SoCs is limited to basic low-level drivers and standard operating systems. The FVP for these SoCs will require accesses to software accessible registers and memory map locations with limited functionality
- User interface application. In these SoCs the processor is used for user interface routines to dynamically monitor or modify the operation of the processes. Software development for these SoCs include several layers of drivers and application interfaces. The FVP for these SoCs will require register and memory map accesses with normal mode functionality
- Real-time application. In these SoCs the processor is tightly coupled to the functional operation of the SoC. Software development for these SoCs includes very detailed multi-layered drivers and algorithms. The FVP for these SoCs will require register and memory map accesses and detailed functionality including special case handling

The level of fidelity necessary for the integration and test teams is dependent on the interactions between the subsystems. The FVP tests that each subsystem implementation will correctly integrate with other subsystems. The FVP models should be close enough to the implementation to pass data between each block in all the different possible modes and sizes. The FVP models should be close enough to the implementation to handshake correctly with each interfacing block and the FVP should be close enough to correctly handle backpressure and stall cases between each block.

The SoC development team does not need to completely define the FVP for the subsystem teams to begin. The FVP can, and should, evolve as the development process continues. Basic functionality such as register accesses can be modeled to begin the FVP with more complex functionality added as the development process requires. The importance of the FVP as a unifying vehicle is that each subsystem and development team uses a common reference model to develop against at the correct time.

4.1.1 Designing the FVP

The FVP is created in a top-down fashion following the partitioning defined in the specification. The modeling team first identifies the modules for the systems and creates a hollow shell of each module. The hollow shells contain the module name and the external ports for each module. Careful consideration must be taken at this stage to define the ports in a consistent manner with the correct attributes. After the modules have been defined the modeling team defines the channels that the modules will be interfaced to. Again careful consideration and planning goes into this step to provide common interfaces that can be used throughout the process. The final goal is to have these channels communicate at the transaction-level, but higher levels may be more easily defined to begin with and then later refined down to the transaction-level.

Once the modules and channels are defined each of the modules will have their processes defined. The modules are defined to each be functionally correct representations of their corresponding implementation. Separate threads for parallel processes are used with state stored in member variables. As the architects define the implementation to lower levels of detail the FVP is refined with this information. A common communication mechanism such as a document change tracker or bug tracking software is used to log changes between the implementation and the FVP. More information on creating a FVP can be found in the application note [“Creating a Functional Virtual Prototype.”](#)

4.1.2 Verifying the FVP

One of the most complicated tasks in developing the FVP is verifying its operation is correct. The unified verification methodology utilizes the FVP as a golden reference model, therefore it is imperative that the FVP be functionally correct or the implementation will also be flawed. There are three layers to verifying the accuracy of the FVP. The first layer is verifying the model meets the performance and functional requirements detailed in the specification. The second layer is verifying the behavior of the model is as intended and the final layer is verifying the algorithms and processes the SoC is implementing.

The methods for verifying the FVP are similar to verification of the implementation. Stimulus generation is a mix of random and directed methods targeted at early architectural and performance verification along with interface tests to smooth the integration process. Architectural checks are used to monitor and capture the response of the FVP to the stimulus. Finally, advanced verification techniques including coverage analysis and assertions are used to speed the process of verification.

Stimulus generation

The three layers of FVP verification are encompassed in the methods used for stimulus generation.

- **Specification requirements.** It is important to do this verification early in the development process so architectural errors do not cause redesign later in the development process. This testing is done with directed tests to verify architectural behavior in a basic isolated manner. Early random testing may throw too many variables into the process, clouding the verification and making debug difficult. Once the directed tests have verified correct basic operation, directed random testing is used to test special case and stress conditions for the architecture
- **Behavior.** The FVP may be meeting the performance and architectural goals but the implementation of the FVP may be done in a manner that does not represent the architect's intention. Directed and pure random stimulus along with behavioral monitors on individual modules of the FVP will determine if a unit is behaving in an improper manner, such as dropping more packets than it should or bypassing stages in a pipeline
- **Algorithms and processes.** Often the stimulus used to verify the algorithms in isolation can be used in a directed manner on the FVP to prove the algorithms correctness in the FVP. Otherwise directed tests are used to stimulate the basic operation and known corner case of the FVP and directed random tests are then used to cover unknown cases

Architectural checks

All three of the FVP verification layers are encompassed in the Architectural Checks in the FVP.

- **Specification requirements.** The verification team should work with the architects to define all of the functional and performance requirements for the SoC and then develop checkers to verify this operation. Functional requirements can include calculation accuracy, event ordering and adherence to protocols. Performance requirements can include bandwidth, latency of operations, and computation speeds. These checkers may be self checking or may require post processing and are the basis for the application assertions to be defined later.
- **Behavior.** Architects often behaviorally model some of the more complex operations of a SoC to create the best solution and measure tradeoffs. These models are used as a reference models for the FVP to verify that the architects intended behavioral operation of the design is implemented correctly
- **Algorithms and processes.** Many functions of a SoC can be described in algorithms or simple process descriptions. These algorithms and processes can be represented in many different forms or languages. These functions within

the FVP can be verified by either embedding these representations into the models with a transaction-level wrapper or by running the representations in parallel to the model and verifying the outputs are equivalent

It is important to consider the scope of the testing when developing the architectural checks for the FVP. The architectural checks are meant to test the operation of the entire SoC to the intended specification. Implementation details of the subsystems will be tested by the individual subsystem teams and system verification will test the correct interface with real-world stimulus.

Advanced verification techniques

Advanced verification techniques are used to verify the FVP and are continued to be used throughout the development process.

- **Assertions.** As noted in the Architectural Checks section above, application assertions are developed for each of the functional requirements listed in the specification. These assertions can verify correct ordering, fairness of arbiters, contention and special case handling. The assertions are coded outside of the FVP in a separate file so they can be used at the implementation-level and in system verification. Interface assertions are also used in the FVP. The FVP is transaction-level so signaling information is not available to be checked but, protocols specifying the types, sizes, and order of data transfer between units can be checked with interface assertions. These assertions should also be written in a separate file so they can be used as a starting point for signal level interface assertions in the implementation. These assertions will need to be modified to include signal level information. Finally structural assertions can be included in the FVP but are only useful to debug the model and cannot be reused in the implementation. The assertions can be embedded in the SystemC code with print statements that can be disabled to help identify where bugs are first detected
- **Coverage.** Coverage techniques are also used when verifying the FVP. Application coverage monitors are included with the application assertions to verify that each of the functional requirements has been verified before the FVP is handed off. The application coverage should be measured when the basic system tests have been run along with some random tests. Interface coverage is also used throughout the development of the FVP. The FVP is transaction based so verifying handshake and signaling coverage is not relevant, but interface coverage is important for identifying what stimulus has been applied to the design and correlating the responses. Code coverage applications may be used on the FVP, but are dependent on the language the models have been written in
- **Formal verification.** Formal model checking techniques can be a very powerful way to verify specific architectural and performance requirements of an FVP. Assertions defined in the architectural checks can be formally proven using mathematical techniques allowing larger state space coverage. Formal software may require constraints to be declared at the interfaces of the model to limit the number of false-error cases. Model checking for architectural and performance verification is best performed at the FVP level where problems can be detected early. Also, most model-checking software does not have the capacity to encompass an entire design represented at the implementation-level

4.1.3 Architectural and performance analysis with the FVP

The FVP, in addition to having many testbench and software development uses, is used for architectural and performance analysis. The FVP is developed in parallel with the implementation architecture after the high-level architecture has been defined. The FVP will be the first implementation of the architecture at a high-level and is useful in both verification and analysis of the decisions made during the initial architecture stages. Accurate timing models will not be available at the FVP level but, estimates can be specified within a layered communication protocol. Specific architectural analysis may include issues such as using a shared bus versus point to point, time domain crossing, and hardware-software trade-offs. Specific performance issues may include bus utilization, resource sharing, and queue sizing.

4.1.4 Refining and integrating the FVP

The subsystem teams will refine the models within the FVP providing more detail until they are ready to integrate their implementation-level subsystem into the FVP. Subsystem teams will provide updates to the FVP to detail the operation of the subsystem with implementation specifics and may further partition the subsystem down into blocks within the FVP. The implementation-level subsystems will be verified inside the FVP in isolation before they are delivered to the SoC team for integration. This verifies the implementation meets the operation intended by the FVP executable specification.

Verifying the individual subsystem implementations in isolation in the FVP does not verify that two subsystem implementations will work together correctly. The SoC team must integrate each of the subsystem implementations together in an organized fashion to verify these implementations will function correctly together. The SoC team does this testing by adding one implementation block at a time into the FVP separating implementation-level blocks

from transaction-level blocks with transactors. Subsystems should be added one at a time if possible to limit the number of possible causes of integration errors. Once a subsystem is verified to be operating correctly with its surrounding implementations, performance is improved by putting that subsystem into hardware acceleration for subsequent integration of other blocks.

An important consideration for the SoC integration team is when to move to a pure signal-level top-level interconnect. The final SoC implementation will have a pure signal-level top-level that is usually provided by either the physical design team or the system design-in team. These top-levels can consist of tens of thousands of individual signals for large SoCs. The only place these top-levels are verified is with the final integration, so it is imperative that the SoC team provides an efficient method for this verification. The SoC team should move to the physical signal-level top-level interconnect as soon as possible after the transaction-level FVP has been verified. This may mean keeping two top-level interconnect models in sync for a period of time during the transfer. The signal-level top-level will also be necessary as it is often common for many side-band signals to appear between implementation blocks that are not necessary in the transaction-level.

The SoC team makes the integration process smoother by verifying the physical signal-level top-level with the transaction models as shown in the figure below. In this example, transactors are placed at the interfaces of each model and interface monitors are converted to signal-level and placed between pairs of transactors. The test suite is then re-run in this configuration to verify the signal-level top-level. The transactors and interface monitors, which will be reused by the subsystem teams, are also verified in this configuration.

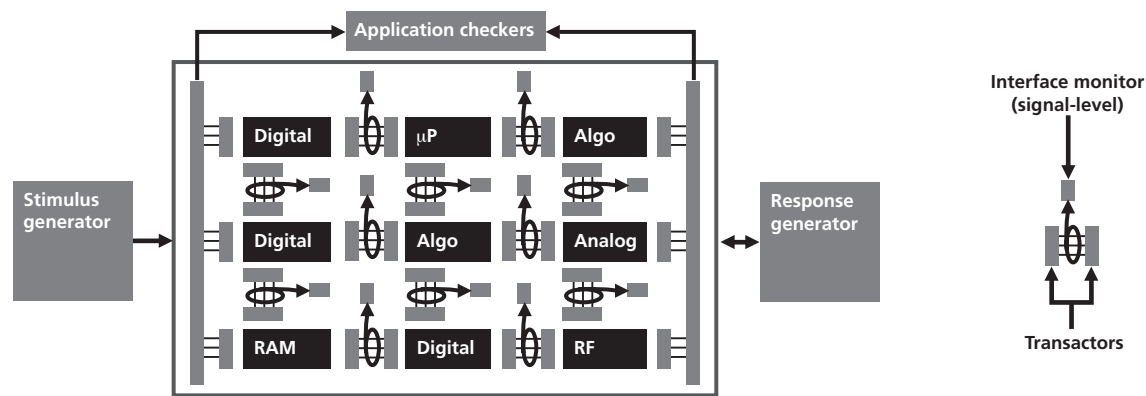


Figure 8: FVP with signal level interface

4.2 SUBSYSTEM REQUIREMENTS

The FVP is the handoff mechanism between the SoC team and the subsystem teams in the unified verification methodology. The subsystem teams will verify their subsystems within the FVP before returning their subsystem implementations to the integration teams. To this point the specifics of what the subsystem teams are to deliver to the SoC team has not been discussed. In addition to the design representation the subsystem team should include testbench components that will more easily facilitate integration. Each subsystem will have its own set of functional handoff requirements and different SoC teams may require more than what is listed as follows.

- Control-based digital subsystem
 - Design representation. Implementation-level representation in RTL
 - Structural assertions. Low-level assertions to aide in debug
 - Transaction-based interface monitors. Interface monitors to aide in debug
 - Reusable testbench. Drivers, transactors, response checkers
 - Acceleration ready. Design has been mapped into hardware accelerator
- Algorithmic-based subsystem
 - Design representation. Implementation-level representation in RTL or gates
 - Transaction-based interface monitors. Interface monitors to aide in debug
 - Reusable testbench. Drivers, transactors, response checkers, models
 - Acceleration ready. Design has been mapped into hardware accelerator

- Mixed-signal instrumentation. Signal analysis software, plots, signal representations
- Analog subsystem
 - Design representation. Behavioral models
 - Reusable testbench. Drivers, transactors, response checkers, models
 - Mixed-signal instrumentation. Signal analysis software, plots, signal representations

4.3 METHODOLOGY DESCRIPTION

The unified verification methodology is broken into four stages:

- SoC design. In this stage the SoC team architects the system creating the specification and the FVP.
- Subsystem design. In this stage the individual subsystem design teams use the FVP to develop their subsystems.
- Subsystem integration and test. In this stage the SoC team integrates the subsystems and tests the complete SoC implementation.
- System verification. In this stage the SoC team verifies the real-world operation of the design.

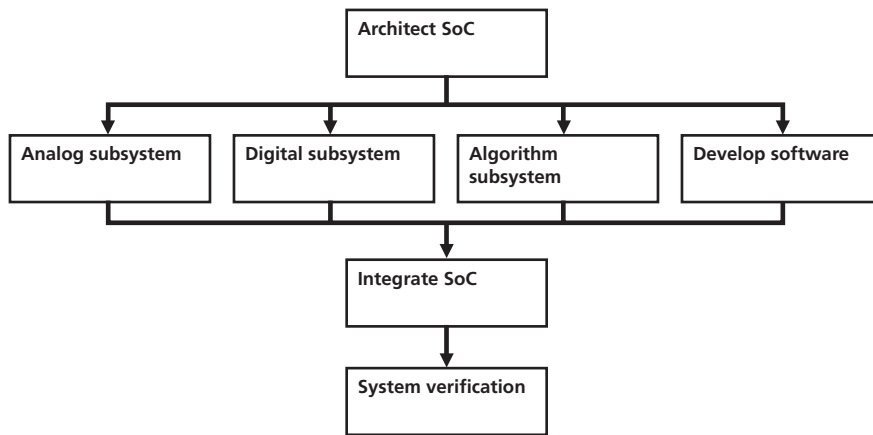


Figure 9: Unified verification methodology flow

4.3.1 SoC design

Projects start in many different ways: some start from a past project, some start from a behavioral or algorithmic representation of a desired function, and some start from a clean sheet of paper. The unified verification methodology begins at the same time as the implementation specific architecture. The verification team engages in two parallel tracks. One team, called the modeling team begins development of the FVP. The modeling team develops the FVP in parallel with the development of the implementation-specific architecture beginning with a high-level model and refining the FVP as the architecture is refined. The modeling team will also begin identifying and developing the testbench components that will be needed. An important part of the modeling teams focus is planning for reuse. Defining the specific reuse requirements of each part of the FVP and testbench makes sure the work is done correctly the first time and speeds the overall development of a unified environment.

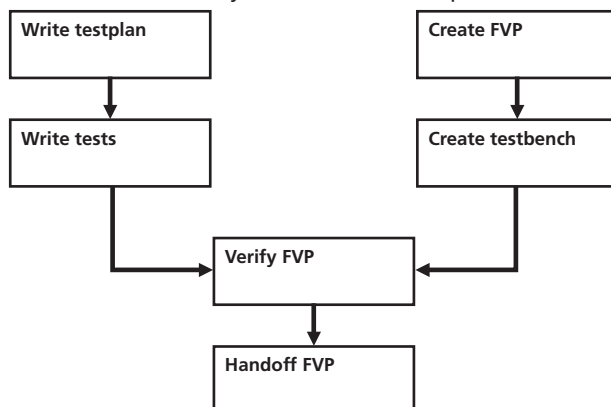


Figure 10: SoC architecture stage flow

The second parallel track is taken by the test team. The test team begins by developing a detailed test plan for the system level. This test plan details the strategy used to test the implementation and sets the standards and goals for the project in areas including coverage, transactions, and assertions. Defining the standards and goals for these areas at the start of the project provides for their use throughout the verification process in a common manner. After the test plan is complete the test team writes the tests for verifying the FVP and executes the tests on the FVP provided by the modeling team. Having separate teams developing the model and testing it provides two independent interpretations of the implementation helping to identify possible areas of confusion in the specification.

After the FVP has been verified in the system environment it is provided to the development teams to be used as an executable model. There are three main customers for the FVP listed below. The modeling team is responsible for keeping these teams up to date with the FVP and resolving any issues discovered by the teams with the FVP.

- Software development teams. The software development teams use the FVP to develop hardware dependent software. An instruction set simulator (ISS) may be attached to the processor along with debugging software for more efficient software development
- System development team. The system development team uses the FVP to speed the design-in of the part. The team will use the FVP to simulate the SoC with other components, verify interfacing, and perform architectural and performance testing
- Subsystem development teams. The subsystem development teams will use the FVP as a source of common reference models to unify the development of subsystems. The teams will substitute the models of their individual subsystems with the implementation when it is available and verify the operation of the implementation within the FVP

4.3.2 Methodology steps

Test team

- Create system-level test plan
 - Develop system test goals
 - Develop system test strategy
 - Develop integration strategy
 - Develop FVP test strategy
 - Identify testbench components
 - Develop transaction taxonomy
 - Identify application coverage goals
 - Identify architectural checkers
- Write system-level directed tests
 - Interface tests
 - Feature tests
 - Stress tests
 - Error tests
 - Performance tests
- Write system-level directed-random tests

Model team

- Create FVP
 - Model interfaces
 - Model processes
 - Model software accessible state
- Create system testbench components
- Create architectural checkers

Both teams

- Verify the FVP
- Handoff the FVP
 - Handoff to subsystem teams
 - Handoff to software teams
 - Handoff to system design-in teams

4.3.3 Subsystem design

Once the FVP has been delivered to the development teams they begin development of the individual subsystems. The methodology of each of these subsystems is detailed in following chapters. During this stage the SoC team continues to refine the FVP with input from the subsystem development teams and adds more detailed tests and checkers to the FVP. The SoC team maintains the consistency of the FVP between subsystem design teams with periodic updates.

4.3.4 SoC integration

In the unified verification methodology each of the subsystems has been continuously verified using the FVP as a common reference so the integration and test of the system should be straight-forward. The SoC team integrates each implementation block into the FVP one at a time and runs the system test suite to verify the integration. Simulation performance will degrade as more blocks of implementation are implemented so, hardware acceleration should be used for the digital blocks as they are integrated. The lower level assertions and monitors should also be included in the integration testing to aide in debug. The test suite is run on the integrated implementation with the FVP used for comparison checking. Once the system has been verified as equivalent to the FVP the implementation is considered the implementation-level FVP and the original FVP is the transaction-level FVP and the design is ready for system verification.

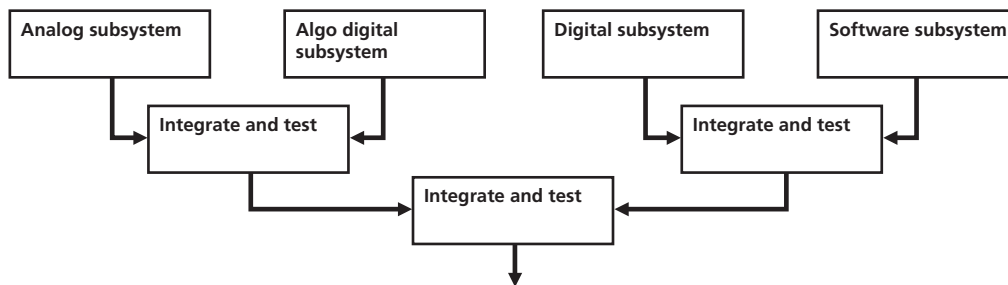


Figure 11: SoC integration flow

The order of integration is dependent on the design and the time of delivery of each piece. If the design contains analog blocks connected to a DSP block with the DSP block connected to a control-based digital block then the analog and DSP blocks should be verified first, and then verified with the control-based digital. If the design contains analog blocks directly connected to control-based digital along with algorithmic based then the control-based digital should be integrated first with the analog independently and then with the algorithmic based digital independently. Once these two integrations have been verified then the system as a whole should be verified.

4.3.5 System verification

The goal of the system verification phase is to verify the system under real world operating conditions. The unified verification methodology utilizes system verification for several roles. First to verify that the testbench environment that has been used to stimulate and check the implementation has accurately reflected the system. System verification is used to provide a mechanism for hardware-software co-verification in a realistic environment. To this point software development has been done on a model or instruction set simulator (ISS) attached to the FVP or implementation model using only basic software debug applications. In system verification software development is run either on the actual CPU or a mapped version of the CPU utilizing all of the software debug applications available in a real world environment. Finally, system verification is used as a design chain handoff mechanism allowing early access of the implementation to help design-in of the system with design chain partners.

There are three basic types of system verification methods that may be used in the unified verification methodology: Simulation-based, emulation, and hardware prototype. The decision as to which method should be used is dependent on the application and the skill set of the team. Software simulation works well for smaller designs that do not need to run at fast speed for long periods of time. Setup and conversion to software simulation methods is quite straightforward. FPGA development platforms work well for modular designs such as SoCs. Setup and conversion can be quite cumbersome for someone not experienced with FPGA development and partitioning. The FPGA solution can run much closer to the speeds of the system but it provides only limited visibility to help debug any problems encountered. Emulation systems work well for large designs that do not need to run at full speeds but do need to be accelerated much faster than simulation. Emulation systems interface well to external devices and provide excellent visibility and debug support for debugging design issues.

The unified verification methodology speeds the system verification process in several ways. First the reuse of models and testbench elements speeds development time. Second, the reuse of transaction based models and interfaces improve the speed of the system, decreasing test time. Third, the reuse of assertions and the use of common user interface from the implementation stages speeds debug. Finally, the hardening process of digital subsystems decreases the time to working emulation or prototype.

4.3.6 Software-based simulation

Software based system verification provides the greatest amount of observability and the smallest modification time of the three methods. The performance of software based system verification can limit the amount of verification performed. The first step in the process is modification of testbench. The implementation is now controlled and monitored by the actual external environment. The testbench is modified to remove stimulus generators. Response checkers are modified to become passive monitors for use in debug. Hardware dependent software will have to be loaded either into the processor model, into the simulation or controlled through an ISS. If the system contains mixed-signal subsystems such as algorithmic-digital or analog subsystems then they are either modeled in a higher level of abstraction for simulation such as a TLM or black boxed and ignored.

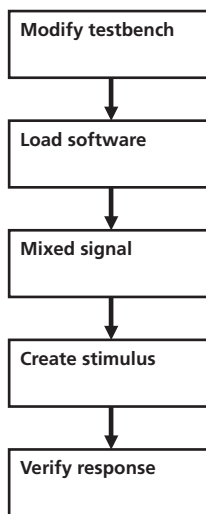


Figure 12: Software-based system verification flow

Stimulus can be provided in several different ways. The stimulus can be provided by interfacing to test equipment and capturing the stimulus for playback on the implementation. The stimulus can be provided by using API interfaces to the simulator to receive and drive data to and from a workstation or network. The stimulus can be provided by a model or TLM that mimics the real world stimulus. The output of the system is verified through comparison and analysis. The output can be compared for accuracy to an existing system or the FVP. Analysis of the output can verify user interfaces and performance requirements.

Advanced verification techniques are continued to be used throughout software-based system verification. All of the assertions defined at the architectural, interface and structural levels are reused to aide in debug and detect errors found with the new stimulus. Application level coverage goals can be re-verified to assure the stimulus is working correctly but lower level coverage monitoring is not used. Transaction-level interfaces are used to provide debugging environments that are the same as used in subsystem development and integration. Acceleration is used to increase the simulation speed of the design in a similar manner as it was used in system integration.

4.3.7 Emulation

Emulation uses a hardware based verification system to emulate the functional operation of the system. The implementation is mapped into the hardware verification system and the emulator is used in place of the final device. The first step in the process is to map the design into the emulator. The subsystem teams may have already mapped the design into a hardware accelerator so this step is often not needed. Next, the team will identify each of the interfaces to the system and develop the necessary instruments. Stimulus can be provided by test equipment, independent workstations or application-specific devices such as transmitters.

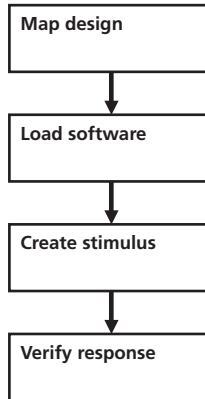


Figure 13: Emulation-based system verification flow

The hardware dependent software will be run on the emulator either through a compiled core where the software will be stored in the emulator memory, or on an external processor board. The software will be loaded through normal system interfaces and debugged with standard applications. Checking the correct behavior of the emulated system is dependent on the application being developed. Test equipment can be used, internal capture and comparison logic can be used, or real world observation can be used to test the response of the system.

Advance verification techniques are continued to be used during emulation. All of the assertions defined at the architectural, interface and structural levels are reused to aide in debug and detect errors found by the new stimulus. The assertions should be synthesizable so they can be mapped into the emulator to run at system speeds. Assertions that do not map to the emulator should be removed; running assertions in lockstep on a simulator will slow the execution speed of the simulator. Coverage monitors are not run on the emulator as they are rarely synthesizable and do not provide useful information. Interface coverage may be used to verify the types and sequences of stimulus provided by the emulation environment.

Transactions are used to improve the performance of the emulator and to improve the efficiency of debug on the emulator. Providing stimulus and measuring responses at the transaction-level allows the emulator to run faster because it will not have to slow down for transfer of signal level information. Also, transaction based monitors are used to provide a debugging environment identical to the one used in subsystem development and integration. Emulation is also used as a verification hub, providing early access of the implementation to design chain partners.

4.3.8 Hardware prototype

Hardware prototypes are hardware systems built to replicate the real system environment using programmable hardware such as FPGAs to represent the implementation. Hardware prototypes may provide the most high performance solution for system verification but also require the most work. The process begins with the development of the prototype system. In most cases the prototype system will have to be built or modified to be used for system verification. The prototype system will require a tested board with standard interface components along with observation interfaces. The digital implementation is placed in programmable hardware, the analog and mixed signal subsystems are implemented on the board.

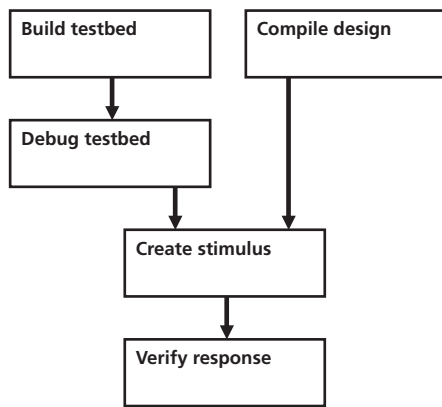


Figure 14: Hardware-based system verification flow

Once the system is built and debugged the design is compiled into programmable hardware devices and system clock speeds are chosen to meet the timing of the compiled implementation. The subsystem teams may have already verified the blocks map correctly into programmable hardware as part of the block hardening process. Stimulus is provided through the prototype system board and can be driven through test equipment, external workstations or networks. Software loading and debugging is done in the same manner as the real system. Service processors can be used to boot the system and software debuggers connected through JTAG interfaces can be used to debug the software. The response checking is done in a manner similar to the real system. Response to the stimulus can be captured for analysis by test equipment or the application can be tested to the user requirements.

Advanced verification techniques are not commonly used with hardware prototypes. Assertions and coverage monitors do not map easily and do not have a common interface to standard programmable hardware devices. Hardware prototypes can be replicated for design chain partners providing early access to the implementation-level FVP. This requires a great deal of up-front planning and back-end support.

5 CONTROL-BASED DIGITAL SUBSYSTEMS

Nanometer scale designs have led to a large scale growth in silicon capacity dramatically affecting the amount of logic a development team can put on a single chip. This growth is most dramatically seen in the control-based digital subsystems being developed today. Control-based digital subsystems differ from algorithmic-based subsystems in that they are developed from a specification of a specific control process. The designs may contain datapaths or algorithms but are not based on algorithmic processing. The complexity associated with these massive digital subsystems has been the focus of much of the functional verification effort to date. Specialized software for adding assertions and measuring coverage along with specialized verification languages and methodologies have been developed to address just this area. Unfortunately, this focus has been the source of much of the fragmentation found in functional verification today. These focused methodologies and software applications force the verification team to start development from scratch, isolate the verification of the subsystem from other design domains and provide for little if any reuse during integration and system verification.

The unified verification methodology removes the fragmentation associated with methodologies focused solely on the digital subsystem level by using the FVP to unify the different stages and design domains. The unified verification methodology continues the process of top-down modeling and reuse of testbench components from the SoC stage through the subsystem stages. The FVP allows the development team to reuse much of the modeling done at the SoC level, provides a common set of models across design domains and leverages the modeling and testbench development through integration and system verification.

A complete top-down flow as detailed in the unified verification methodology may not be practical or most efficient for a development team at this time. The development team needs to balance the benefits of a top-down FVP based methodology with the potential requirements. These benefits and requirements are listed as follows:

Benefits

- Early software development. If the system being developed contains large amounts of new hardware specific software and the software developers are available to work on the FVP then there is great benefit in a top-down FVP based methodology
- Multiple design domains. If the system being developed contains blocks of analog or algorithmic digital designs that must be interfaced with the control-based digital design then there is a large benefit in a top-down FVP based methodology

- Early test development. If the development team has separate verification engineers that can write and develop their tests on the FVP before the implementation is available, there is a large benefit in a top-down FVP based methodology
- Top-down reuse. If the development team can reuse the modeling and testbench components from the FVP in an efficient manner then there is a benefit in a top-down FVP based methodology

Requirements

- Available resources. The hardware and software resources must be available early in the development process to obtain the boost in performance and efficiency of early access to the FVP. If the verification tasks are performed by the design engineers after they have completed design then there is little gain in having an early FVP model
- Complex models. Certain types of designs do not lend themselves to verification by comparison to reference models. These designs are very complex to model and require intimate knowledge of the implementation. In these cases the FVP may not be as useful of a method for verification
- Modeling knowledge base. The development team will need to further refine the FVP for use in the subsystem verification. This requires a subset of the verification team to have understanding of modeling techniques and the use of a standard modeling and verification language
- Maintenance. The development team will need to maintain the consistency of the FVP. For large scale subsystems this can be a complex task managing two models and keeping both consistent

The above list of benefits and requirements will lead some development teams to choose a more specification based bottom-up approach to subsystem verification for today's design. As the use of processor based SoCs grows along with the further integration of digital and analog on the same chip, the benefits of a top-down FVP based methodology will cause many teams to migrate. This section will first present the top-down FVP based flow for SoC based designs as a standard for teams to migrate towards. The section then presents a bottom-up specification based flow utilizing many of the advanced techniques of the top-down flow as a start for teams to begin the migration.

5.1 TOP-DOWN VERSUS BOTTOM-UP FLOWS

There are many similarities between the top-down FVP based flow and the bottom-up specification based flow described below. First, even though the bottom-up flow develops the testbench from the lowest unit-level up to the subsystem level, the planning and architecture for the flow is top-down. The only way to create testbenches that can be reused as the design develops from the unit to the block to the subsystem level is to plan ahead and know what will be required at higher subsystem levels. Similarly even though the top-down flow develops the testbench from the top SoC level down to the lowest unit-level the testing and integration are still performed in a bottom-up manner. Both flows use transaction-based testbenches for performance, assertions for easy debugging, coverage for efficient test development and hardware acceleration for increased performance.

The two flows differ in the development of the verification environment and tests. In the top-down flow the environment is developed from the highest levels (SoC FVP) down to lower levels using common models and testbench components. The SoC level environment is developed first with the SoC-level FVP, the FVP is refined down to the block and sub-block level and the testbench is developed in the same manner. This top-down development allows the verification engineers to use the model to test their code, allows reuse of the models in the FVP at different levels, and promotes parallelism and accelerated integration of the implementation when it is made available.

The bottom-up flow develops the testbench from the lowest level up to higher levels. The flow reuses testbench components from the lower levels as the design is integrated and tested. Since a common model is not used, reference checkers may need to be developed at each level or linked together. Also, the lack of an accurate model means the tests and testbench components are developed in isolation until the implementation is available. This leads to the simultaneous debugging of implementation and testbench.

5.2 TESTBENCH DEVELOPMENT

As has been noted earlier in this paper, the method for developing a testbench has a dramatic impact on the overall performance and efficiency of the entire verification process. This section describes the development of testbenches and describes their individual components. Testbench development for speed and efficiency is centered on two main characteristics-reuse and raising the level of abstraction. Reuse speeds the development of testbench components. The key to reuse is limiting the amount of application-specific information that is encapsulated in the testbench components. The less application-specific information the components contain the more reusable they can be. Raising the level of abstraction makes test writing and debugging much more efficient. The key to raising the level of abstraction in a testbench is having standard defined interfaces. The easier the communication between components is facilitated the easier it is to work at higher levels of abstraction.

High-performance reusable testbenches are based on standard components with a common interface for communications at different levels of abstraction. The basic components are shown in the following diagram and described below.

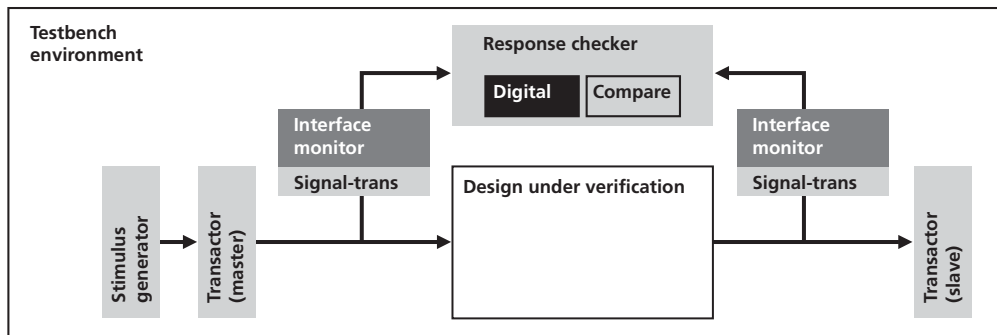


Figure 15: Control-digital testbench development

5.2.1 Stimulus generators

Stimulus generators—directed generators, random generators, and directed random generators—create the data the testbench uses to stimulate the design under verification (DUV). Stimulus generators can create the data in a preprocessing mode with custom scripts or capture programs or they can create the data on-the-fly as the simulation occurs. Stimulus generators are usually classified by the control the test writer exerts on the generation of the stimulus.

Directed generators

Directed generators such as test vector files or tests that specify the exact type and sequence of data are used to directly test individual functions within the design. The test writer needs to have precision control over the data and its application in order to guarantee they have stimulated the function in the desired manner. There are several different types of directed tests—interface, feature, error, and performance.

Interface tests verify the correct operation of each of the major interfaces found in the subsystem. The tests verify correct handshaking and error handling. Stress tests are included to verify the constraints of the interface such as timeouts, aborts, and deadlocks. These tests should be run first to verify the correct communication between the testbench and the DUV.

Feature tests verify the processes contained within the subsystem. The tests verify the correct operation of the features under normal and stress conditions. Stress conditions include system interactions between features such as interrupts, retries, and pipeline flushes. These tests are run second with the goal of verifying each feature in isolation under non-stress conditions before turning on randomness.

Error tests verify the correct operation of the subsystem to error conditions. Error conditions consist of recoverable and non-recoverable errors. Recoverable error tests verify the observation of the error and the recovery from the error. Non-recoverable error tests verify the observation and the correct system response such as a halt, freeze, or interrupt signal. These tests should be run after feature testing is complete and random tests have run for multiple hours without failing.

Performance tests verify that the subsystem meets the performance requirements of the system. Performance requirements can include latency, bandwidth and throughput. The tests stimulate the system with normal rate stimulus as well as corner case stimulus that is known to be performance limiting. Performance tests are run periodically throughout the testing process to verify the design is still meeting test goals and should be run again at the end of testing to verify design changes and bug fixes have not violated the performance goals.

Random generators

Random generators use automation to select the type of data to be applied to the DUV and the order it is applied. The verification engineer uses random generators to cover designs that cannot easily be directly tested and to generate data and data sequences that they may not have considered.

Directed random generators

Directed random generators provide the verification engineer with some controls over the data selected by a random generator. Constraints are placed on individual data fields to guide the stimulus as the test writer desires. Directed random generators are used to guide the creation of data to specific areas of the design or certain modes of operation

Each of the three types of stimulus generators has its place in functional verification. In large complex designs the verification engineer uses directed tests to do initial bring-up of the blocks and to test specific functional requirements. Directed tests are used for initial bring-up instead of random because the engineer wants to limit the number of unknowns and wants repeatability when tracking early bugs. Directed random generators are used by the verification engineer to cover many of the large number of test cases in a fast efficient manner. The number of test cases required to cover a large design is more than could be written in a directed manner. Random generators are used by verification engineers to cover the cases that have not been considered in a test plan. The combination of directed, directed random and random stimulus generators provides a thorough layered approach to verification.

5.2.2 Transactors

Transactors are used to change the levels of abstraction in a testbench. The most common use for transactors is to translate between implementation-level signaling and a higher level transaction representation. Transactors are placed in a testbench at the interfaces of the DUV providing a transaction-level interface to the stimulus generators and the response checkers. Transactors can behave as masters, initiating activity with the DUV, as slaves responding to requests generated by the DUV, or as both a master and a slave. The design of a transactor is application independent to facilitate maximum reuse. Application-specific information can be contained in the stimulus generators or TLMs attached to the transactors. Also, when developing a transactor the designer should consider its use in a hardware accelerator. Developing the signal-level interface in a synthesizable manner will allow it to be accelerated along with the design improving the performance obtained from a hardware accelerator.

5.2.3 Interface monitors

Interface monitors are used to check the correct signaling and protocol of data transfers across DUV interfaces. In some testbenches interface monitors are combined either with the transactors or with the response checkers. Keeping interface monitors separate from these components allows for maximum reuse of the monitors. In addition to passively monitoring the data transfers across interfaces these monitors can encapsulate the data to be communicated to response checkers. This allows the response checkers to concentrate solely on verifying correct operation. Interface monitors contain interface assertions discussed earlier and can be written in a property specification language (PSL) such as Sugar from IBM. The interface monitors should be application independent and written in a manner that allows their easy reuse in hardware acceleration.

5.2.4 Response checkers

Response checkers are used to verify that the data responses received from the DUV are correct. Response checkers contain the most application-specific information in the testbench and usually can only be reused when the block they are monitoring is being reused. There are three basic types of response checkers:

- Reference model response checkers apply the same stimulus the DUV receives to a model of the design and verifies the response of the model is identical to the DUV. The most efficient method of creation is to reuse the TLMs of the FVP for the reference models in the response checker.
- Scoreboard response checkers save the data as it is received by the DUV and monitor the translations made to the data as it passes through the DUV. The translations are tracked with a scoreboard and responses are verified as they are generated by the DUV.
- Performance response checkers monitor the data flowing into and out of the DUV and verify that the correct functional responses are being maintained. These checkers verify characteristics of the responses rather than the details of the response.

Scoreboards are used when the DUV responds in a predictable manner and the data is easily correlated such as a bridge or a switch design. Reference models are used when the DUV can be easily independently modeled with enough accuracy such as a computation unit or a pipeline. Performance checkers are used when the functions in the DUV are unpredictable due to implementation specifics and the correct operation can be specified by the characteristics of the design such as a rate-limiter or a routing algorithm.

5.2.5 Testbench API

The testbench API is used to facilitate the communications between components in the verification environment at different levels of abstraction. The API is the glue that holds the testbench together and provides for reuse of components. Using a standard API allows for the reuse of components within the verification process and between different projects. The testbench API defines the interface layers between components and between different abstraction levels.

5.2.6 Top-down testbench development

Top-down testbench development starts with the development of a top-level subsystem testbench environment. The development starts once the FVP and specification are delivered to the team. The subsystem testbench reuses components from the FVP as shown in the diagram below. In this example the Digital TLM is used as a reference model in the response checker. The transaction-level interface monitors are interfaced to signal-level translators and used in the testbench. In many cases stimulus generators and response generators can also be re-used.

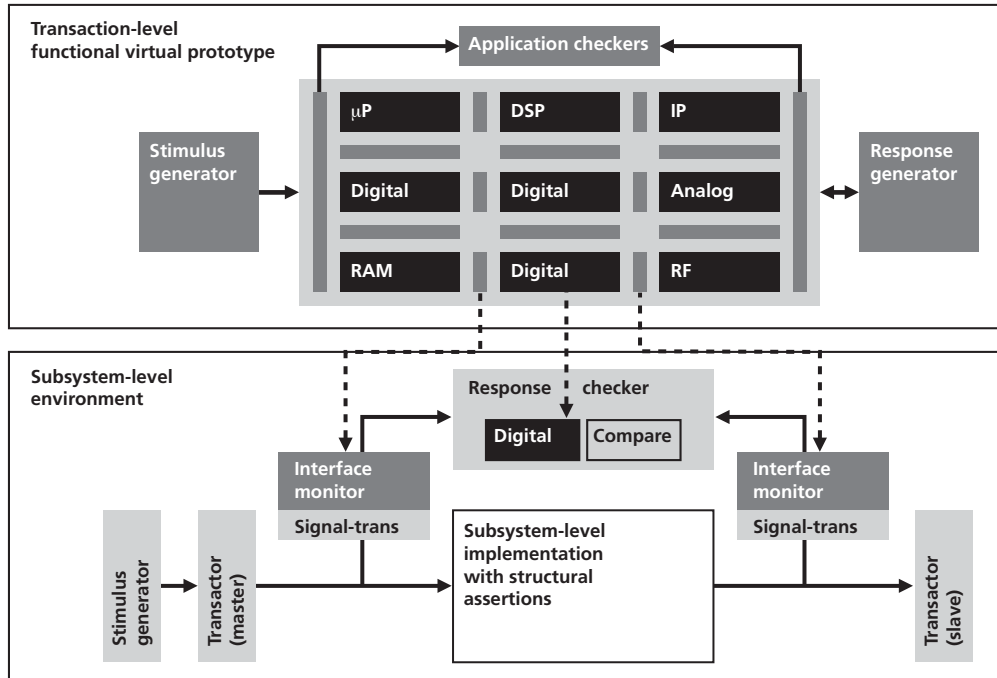


Figure 16: Top-down subsystem testbench development

Signal-level transactors may be provided by the SoC team or they may be developed by the subsystem team. Tests are developed by the subsystem team and are verified in the testbench by substituting the FVP TLM for the implementation until it is available. This will allow the test writers to develop and test their code before the implementation block is available, making the process more efficient. In the example shown below the digital TLM is used in place of the implementation until it is available.

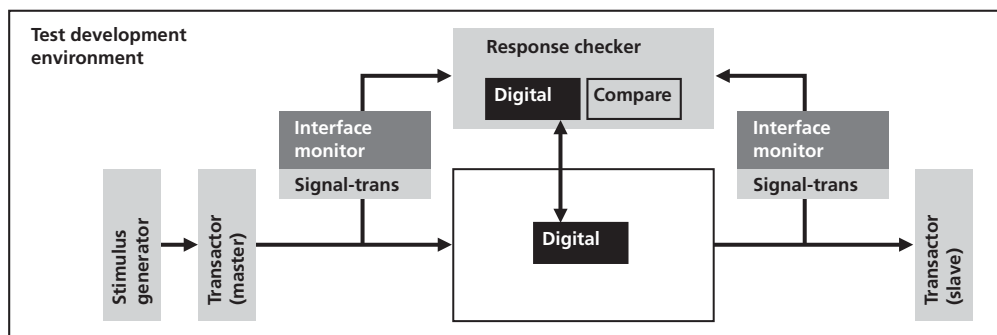


Figure 17: Test development environment

As the subsystem is micro-architected it will be partitioned into smaller hierarchical blocks and units. Large complex subsystems require the verification team to do much of the verification at the block level. The verification team chooses the correct levels of hierarchy to test by selecting blocks with common standard interfaces that provide

enough observability and controllability and are the correct size for simulation software to be efficient. Often designers want to verify at the lowest unit-level as they develop individual modules. This unit-level verification is done by the designer using simple HDLs or HVLs along with the structural assertions they have added. Unit-level verification uses simple methods applying stimulus vectors and waveform inspection or application-specific environments. These environments such as these are not reused as their purpose is simply to prove basic sanity of the unit-level modules.

Block-level testbenches are developed in a similar manner as the subsystem testbench. An example is shown below. In the example the FVP is partitioned to match the verification needs and reused in the individual response checkers. Transactors and interface monitors are reused from the subsystem testbench. Common transactors can be shared in slave or master modes for the different connecting blocks.

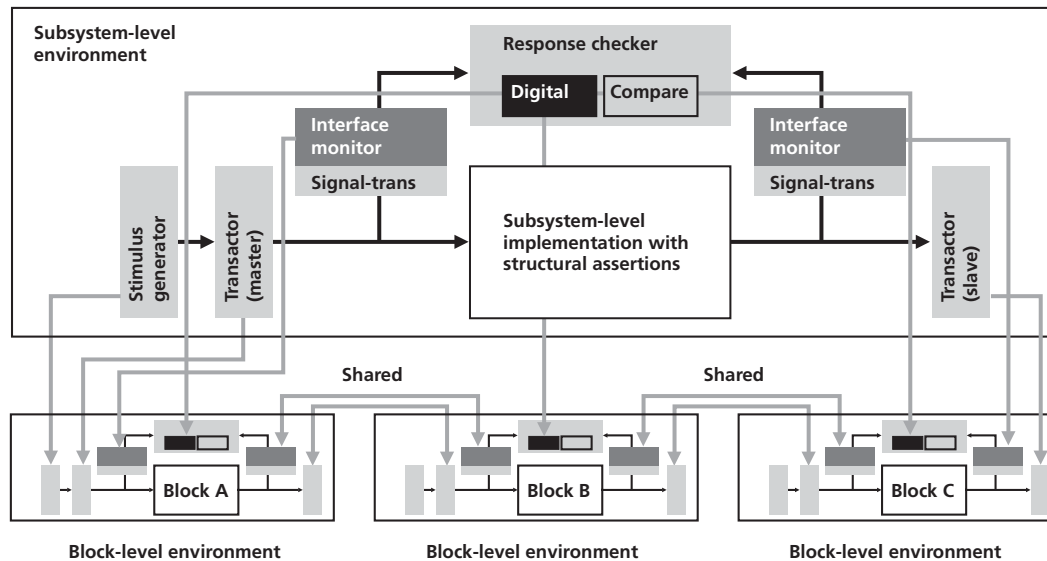


Figure 18: Top-down block testbench development

Tests can be developed in advance of the implementation in the block level testbenches by substituting the models as described in the subsystem testbench. When developed in a correct and efficient manner the top-down testbench method should provide for tests and testbench to be completed and debugged before the implementation is delivered by the design team. This allows for the fastest and most efficient debugging and integration flow.

5.2.6 Bottom-up testbench development

Bottom-up testbench development starts with the partitioning provided by the micro-architecture teams. Once the partitioning is available the verification team can select the correct block and subsystem levels they intend to test at. Often designers want to verify at the lowest unit-level as they develop individual modules. This verification is done by the designer using simple HDLs or HVLs along with the structural assertions they have added. The verification of these units uses simple methods applying stimulus vectors, waveform inspection or application-specific environments. These environments are not reused as their purpose is simply to prove basic sanity of the unit-level modules.

Block-level development begins with building individual testbenches for the lowest block level verification. New transactors, interface monitors and response checkers are developed or reused from previous projects if available. The response generators can be based on models or modified from the FVP but maintaining consistency between these models is difficult. Unless a behavioral model is available the test teams must wait to run and debug their tests until the implementation is available.

The subsystem testbench is developed from the existing block level testbenches as shown in the diagram below. In the diagram transactors at internal interfaces are removed and the response checkers are linked together with interface monitors to provide a complete subsystem response checker. Close communication must be maintained between block level developers so that the subsystem integration works correctly. Blocks located closer to the stimulus must consider the effects of their environment on connecting blocks. Also, changes in the block level testbenches must be propagated to the subsystem testbench and vice-versa. Good communication and planning are required to make the integration of bottom-up testbenches efficient.

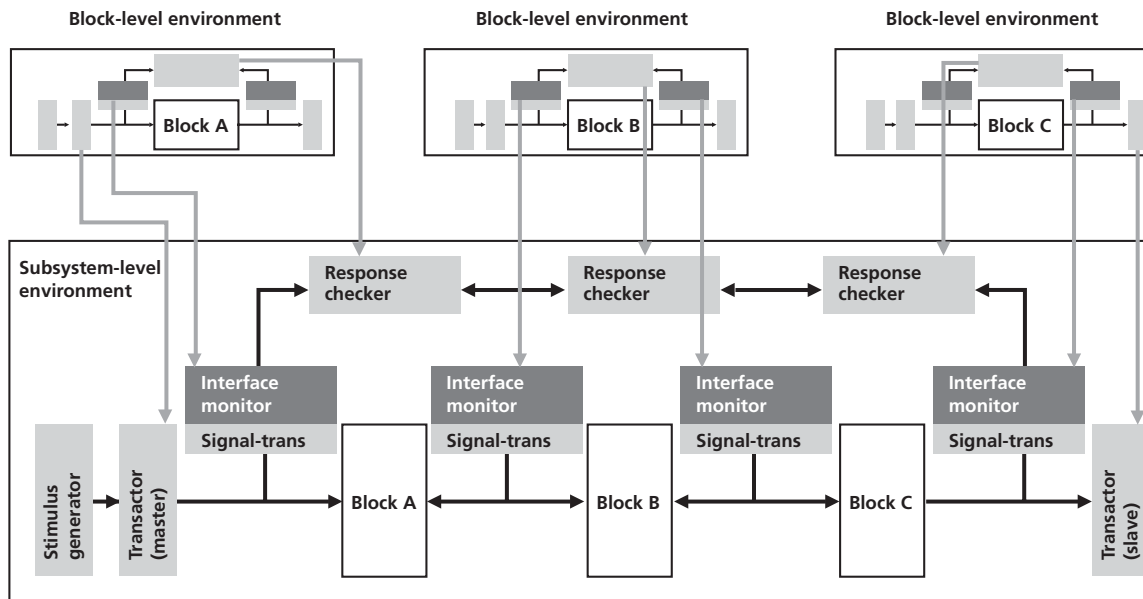


Figure 19: Bottom-up subsystem testbench

5.3 ADVANCED VERIFICATION TECHNIQUES

5.3.1 Assertions

Assertions are used extensively throughout the control-digital subsystem development. Application assertions developed in the FVP that are associated with the subsystem are reused if possible. In many cases the assertions are no longer complete as the design has been partitioned. These assertions are then disabled at lower testbench levels. Interface assertions are also reused from the FVP for external subsystem interfaces. Interface assertions are added to internal interfaces as they are defined during micro-architecture. These assertions are usually part of the interface monitor testbench components and are specified outside of the implementation code. Structural assertions are added by the design teams as they code their blocks. These low-level assertions should be added within the code in comments so that they are not lost during integration and can be updated along with the code changes.

All of the assertions are used to verify the subsystem in a dynamic manner using simulation. The assertions can also be used along with formal verification techniques to verify the subsystem in a static manner. Most formal methods cannot verify large blocks of code with a large number of assertions. Formal static verification should be done at the block or unit-level early in the development process.

5.3.2 Coverage

Coverage techniques are used throughout the control-digital subsystem development. Application level coverage defined at the FVP is usually measured at the SoC modeling and integration stages as it encompasses the entire SoC. New application level goals for just the control-digital subsystem are defined during micro-architecture of the subsystem and are implemented in testbench code as monitors. These subsystem level application coverage goals are measured at the end of subsystem testing. Interface coverage is first defined in the system test plan as part of the transaction taxonomy. The verification team defines the transaction types used at the subsystem level and also defines goals for the types and sequences of transactions driven into the DUV. Specific correlation goals between stimulus and response are also defined at this time. Interface coverage is first measured at the completion of block-level testing. The team verifies that all specified types of transactions have been stimulated along with a large percentage, if not all, of the combinations of transaction types. Specified correlation goals should also all be measured to be met before block verification is considered complete. Interface coverage is also measured at the subsystem level to verify that all transaction types have been simulated along with a subset of the possible transaction sequences.

Structural coverage monitors are added first by the designer along with the structural assertions. In many cases the assertions can be used to monitor an implementation structure's correct behavior as well as incorrect behavior. The verification team adds to these coverage monitors once they first receive the implementation from the design team again in the form of structural assertions. Structural coverage is measured at the completion of block-level testing.

Block-level testing focuses on the specific implementation features of the design and this is where structural coverage provides the most information. Structural coverage information is collected after all tests are run and passing as it slows down the run times and will not provide accurate information until the tests pass. Code coverage is also run after block verification is complete. The verification team identifies holes in the structural and code coverage and investigates to determine if the stimulus is lacking, the design is in error, or there is dead code. Coverage is an iterative process where results are analyzed and modifications are made to the tests or implementation until the team has addressed all coverage holes.

5.3.3 Acceleration-on-Demand

Acceleration is used in the control-digital subsystem to run long tests faster and run efficient test regressions. Many test sequences take long periods of time to set up due to deep memory queues and complex control space. Acceleration is used on these tests to reach the desired states faster than standard simulation techniques. The testbench can be run on the simulator in lockstep with the accelerator or for faster performance the testbench can be compiled into the accelerator if they are synthesizable. Testbench components should be developed with consideration for use in acceleration.

Once a few tests have been run on the implementation and are passing an automated regression environment should be established. A periodic regression is run to assure that changes to the implementation or the testbench environment have not broken existing logic. As the number of tests grows in the regression, acceleration and server farms are used to complete the regression in a timely manner. Server farms are used to run small runtime jobs in parallel with automated scripts. Accelerators are used to run long regression tests and run groups of shorter tests quickly in a serial manner requiring a smaller server farm and fewer licenses.

5.4 HARDENING THE BLOCKS

After interface and feature testing is complete and constrained random testing has run for multiple hours of simulation the block is ready for integration into the subsystem. In addition to integration testing the block is “hardened” by verifying it to the FVP and verifying it is ready for use in acceleration, emulation or prototype. If a top-down FVP based flow is being used then the block is verified against the FVP by running the FVP test suite against the prototype with the implementation block replacing the model. Transactors are added to the block to translate the transaction layer down to the signal layer as is shown in previous diagrams. Adding the implementation block may slow down the simulations and require limiting the tests run. The tests stress features that cross blocks such as pipelines and multiple encapsulations.

Large design blocks will need the performance of a hardware accelerator to run long test sequences or to test multiple blocks integrated together. Once the block is stable it is hardened by running it in isolation through the mapping process on the accelerator. This allows for the early detection of mapping or library issues that could stall the later use of acceleration. In a similar manner the design is run through any synthesis or mapping processes needed for FPGAs or hardware prototypes used in system verification.

5.5 TOP-DOWN FVP-BASED FLOW

The top-down verification methodology for a control-based digital subsystem is broken into three separate parallel tracks that converge throughout the process. The first track is the further development of the FVP. The modeling team updates the FVP with detail as the implementation is refined. The FVP will be used by the other subsystems and possibly at other levels of testbench hierarchy. The modeling team uses the FVP to develop models of other subsystems and reuse TLMs for reference models.

The second track is the implementation of the design. The design team uses an HDL to implement the design starting with development of small implementation units. The development team will add structural assertions and verify these small units to a basic level of operation. The team then will integrate these units into design blocks and provide them to the test team for verification.

The final track in this process is the development of stimulus. The test team begins by writing a test plan focused at the subsystem level. The testplan is a continuation of the system-level test plan and further refines the test strategy along with plans for transactions, assertions, and coverage. The test team will execute the test plan on the blocks provided by the design team to verify their functionality. The design team will continue integrating units together into blocks and providing them to the test team to verify.

Once the blocks have been verified they are integrated into the subsystem and in parallel go through a hardening process. The hardening process measures coverage, verifies the block within the FVP, and runs the blocks through the mapping to acceleration. Once all of the blocks have been verified, hardened, and integrated into the subsystem, the subsystem is verified as a whole. It is important to not lose time in the integration and subsystem verification process waiting for slow or long simulations. The unified verification methodology uses hardware acceleration early in the verification process to provide the necessary performance.

Implementation team

- Codes individual blocks
- Add structural assertions.
- Static unit verification
- Dynamic unit verification
- Integrate units into block level
- Integrate blocks into subsystem

Test and model teams

- Add structural assertions, interface assertions and coverage monitors
- Execute block level test plans
- Create mini-regression
- Measure coverage and improve tests
- Harden blocks.
 - Verify blocks in FVP
 - Map for acceleration and system verification
- Execute subsystem test plan
- Measure coverage and improve tests
- Verify subsystem in the FVP

5.6 BOTTOM-UP SPECIFICATION-BASED FLOW

The bottom-up verification methodology for a control-based digital subsystem is broken into two separate parallel tracks that converge throughout the process. Separate verification and design teams each begin work from the lower blocks of the design and work their way up integrating and testing. If the development teams do not have separate design and verification teams then the two tracks can be performed in serial developing the code first then developing the tests and then verifying the design. The separate serial track is obviously slower and less efficient, but requires fewer resources.

The first track is the implementation of the design. The design team uses an HDL to implement the design starting with development of small implementation units. The development team will add structural assertions and verify these small units to a basic level of operation. The team then will integrate these units into design blocks and provide them to the test team for verification. Once all of the design blocks are developed the design team will be integrated together into the subsystem for integration testing.

The second track is the development of the tests and testbench environment. The verification team begins by developing a detailed test plan for each block in the subsystem and for the subsystem as a whole. The team then creates the block level testbenches in a manner that will allow for reuse at the subsystem level. Once the block-level testbenches are complete the verification team will write block-level tests and wait for the implementation blocks to be delivered by the design team. Once the blocks are delivered they will be verified individually. The team will measure coverage and add tests as required to meet specified goals.

After the blocks have been verified individually the verification team creates testbenches to integrate and test the blocks together. Depending on the size and number of blocks there may be many integration steps or there may be just one integration into a complete subsystem. The testbenches are created with parts from the block-level testbenches where possible. The verification team develops integration-level tests and verifies the integrated blocks together in the testbench. Once the integration testing is complete, the subsystem is then tested inside the FVP provided by the SoC team to verify it will work with other subsystems.

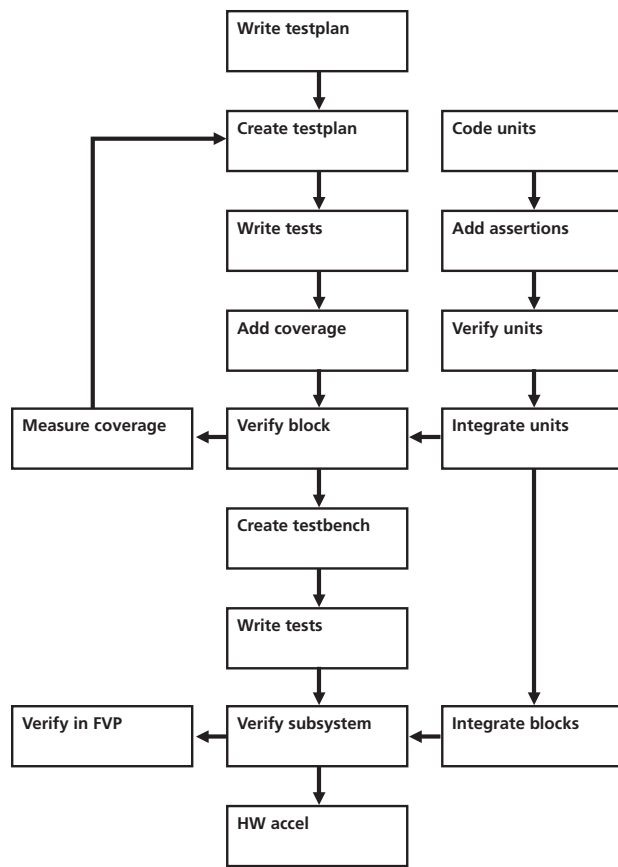


Figure 21: Bottom-up methodology flow

5.6.1 Methodology steps

Verification team

- Create subsystem test plan
 - Develop subsystem test goals
 - Develop subsystem test strategy
 - Develop integration strategy
 - Develop block test strategy
 - Identify testbench components
 - Identify coverage goals
 - Identify assertions
- Create individual block test plans (same categories as subsystem-level)
- Create block-level testbenches
- Write individual block directed tests
- Write individual block directed random tests

Implementation team

- Codes individual blocks
- Add structural assertions
- Static unit verification
- Dynamic unit verification
- Integrate units into block level
- Integrate blocks into subsystem

Verification team

- Add structural assertions and coverage monitors
- Execute block test plans
- Create mini regression
- Measure coverage and improve tests
- Harden blocks
 - Map for acceleration and system verification
- Integrate blocks
- Create subsystem testbench
- Write subsystem directed tests
- Verify subsystem
- Automated regression
- Measure application coverage
- Verify subsystem in the FVP

6 ALGORITHMIC-BASED DIGITAL SUBSYSTEM

Algorithmic based digital subsystems such as DSP subsystems or datapath subsystems are developed in a top-down process that refines a algorithm or protocol to a specific implementation. The unified verification methodology speeds the verification process of these subsystems by closely matching the refinement process with the FVP. Two of the most important aspects of the methodology are the use of accurate models and the use of a common testbench throughout the process.

6.1 ALGORITHMIC MODELS

Algorithmic based subsystems are most commonly used today in the development of communications and multimedia based systems. The system and environmental effects on these types of designs are not as easily predicted as they are in control-based designs. The unpredictability increases the likelihood of an error in the algorithm not being detected until system integration testing. Algorithmic development teams need to verify the intent of the design before implementation is performed. There are too many variables to wait for an accurate implementation before beginning verification.

Algorithmic based subsystems are developed by either modifying some number of blocks in an existing system to provide a new function or reconfiguring the blocks of existing systems to provide a derivative function. Both these processes require the accurate modeling of the individual blocks of the system at different levels of abstraction. A broad range of building blocks is necessary not only for the development of the subsystem but also for the verification of that subsystem. Communications and multimedia applications are usually based on standards and protocol layers. A broad up-to-date library of standard algorithmic system components is a requirement for the accurate development and verification of algorithmic subsystems.

The algorithmic-based subsystem is modeled in the FVP based on the application. The FVP is defined as a transaction-level model of the system but, algorithmic subsystems often operate and interface in a more continuous-time domain. To correctly model an algorithmic subsystem for the FVP each interface should be modeled in the most efficient manner for passing information as shown below. Algorithmic subsystems interface to other mixed signal subsystems where a continuous-time based interface is most efficient. The simulation of these mixed-signal interfaces is discussed in the analog subsystem section. Algorithmic subsystems also often interface to control-based subsystems. These interfaces are defined at the transaction-level similar to control-based digital subsystems.

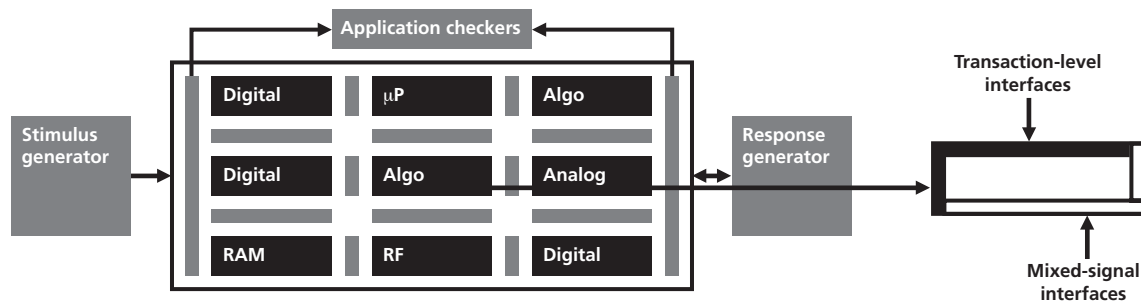


Figure 22: Algorithmic TLM development

6.2 TESTBENCH DEVELOPMENT

As the algorithmic subsystem is refined from the algorithm to the fixed-point representation to the final RTL or gate level representation the accuracy of the algorithm is affected. The testbench environment must verify that at each stage of development the accuracy of the algorithm is still acceptable for the application. Verifying this accuracy causes the testbench to become a mix of models at different abstraction levels. The environment may continue to be modeled at the same level as the block under development moves towards the implementation-level. Unlike the control-digital subsystem where the responses of the subsystem are verified against an expected result, algorithmic subsystems require analysis of the responses to verify the response is still within specification. This is accomplished with detailed instrumentation that captures the response of the subsystem and produces diagrams and calculated error rates for the developer to verify. This instrumentation is at the heart of the testbench environment.

As the subsystem development progresses, integration becomes more important for the testbench environment. The testbench continues to use the signal stimulus generators and instrumentation but also begins to interface with the other subsystems. The design is updated in the FVP so integration testing can be performed before the implementation is final. The software team develops their code for the application on an ISS of the processor core. The analog/RF subsystem is integrated to detail the behavioral effects of the two subsystems. Finally the control path for the subsystem is verified by attaching the implementation to the FVP in place of the TLM. Verifying the interaction with each subsystem in the FVP before integration will speed the final SoC integration.

6.3 ADVANCED VERIFICATION TECHNIQUES

The initial development of an algorithmic subsystem concentrates on the algorithmic and fixed-point representations of the design. Advanced techniques such as assertions, coverage, and acceleration are not used during these early stages. The conversion of the fixed-point representation to an implementation can be done with automated synthesis software or can it be done in a more traditional hand coded method. If the design is synthesized or hand coded then interface assertions are added to the digital interfaces of the subsystem. Transaction-level interfaces are defined for access to control elements of the SoC such as the processor. The definition of these interfaces allows for interface coverage and transaction debug analysis to be used in developing the final implementation-level representation of the design.

To obtain optimal performance from the implementation the subsystem is often developed by engineers in an RTL format similar to the control-digital interface. In these cases structural assertions and coverage analysis is performed in a similar manner as defined in the control-based digital subsystem.

Tests developed at the algorithmic and fixed-point level will run significantly slower as the level of abstraction is moved down to the implementation-level. Acceleration is used to speed the execution of these tests and provide for integration testing with larger subsystems. As part of the development process blocks of the design are put through a hardening process where they are run through the mapping and synthesis stages of the hardware accelerator or FPGA that will be used for system verification. Preparing the design for acceleration early in the development process will smooth the use of acceleration of the subsystem at integration and system verification stages.

6.4 METHODOLOGY

The verification process begins with the development of a detailed test plan and the assembly of the testbench environment. Testbench models are reused from the FVP, such as the behavioral analog models or the control-based TLMs. The models can also be obtained from standard libraries along with instrumentation for examining and analyzing responses. Once the testbench is assembled the algorithm is verified for correctness and updated in the FVP. Next, the algorithm is translated to a fixed-point representation and the same testbench is reused to verify the

correctness. Once the fixed-point representation is verified the FVP is updated and the representation is implemented in HDL and again verified with the testbench.

The unified verification methodology speeds the development of algorithmic digital subsystems by using a single unified engine to simulate across design domains and using a common testbench environment throughout the process.

6.4.1 Methodology steps

- Develop subsystem test plan
 - Develop test strategy
 - Define system components needed for test environment
 - Define control interface
 - Define instrumentation components
 - Define stimulus ranges to verify
- Assemble testbench models and instrumentation
- Update the FVP
- Verify algorithm
- Verify fixed-point representation of the algorithm
- Create register-level implementation representation of design.
- Verify register level implementation
- Harden blocks
 - Measure coverage
 - Map for acceleration and emulation
- Transaction-based control interface verified
- Verify block inside the FVP

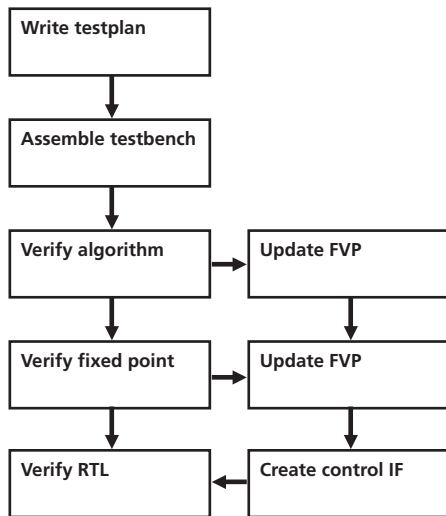


Figure 23: Algorithmic digital methodology flow

7 ANALOG SUBSYSTEM

Analog subsystems are developed from low-level components with the major focus of the verification centering on transient and AC effects. Similar to the control-based digital subsystem the dramatic increase in silicon capacity has led to the evolution from the more basic bottom-up development flows to a more model centric top-down development flows. The unified verification methodology focuses on the integration of the analog subsystem with the rest of the system. The key to this integration is the development of behavioral models of the subsystem that can be simulated with the other subsystems.

7.1 TOP-DOWN VERSUS BOTTOM-UP FLOWS

The most efficient flow for verifying the analog subsystem is to first create behavioral models and then develop the lower level components in a top-down flow. This method follows the top-down FVP development flow of the Unified Verification methodology. In many cases the analog subsystem is a modification of an existing subsystem or the combination of several analog subsystems into a single block. In these cases a bottom-up development flow may be the most efficient method for verification. In a bottom-up flow the behavioral models are developed from the lower level models as they are put together. The advantage of the top-down flow is early analysis of system effects on the subsystem and the reuse of behavioral models for development of other subsystems.

7.2 BEHAVIORAL MODELS

As the size of analog and mixed-signal designs grows the verification and development of these designs requires the ability to represent these models at different levels of abstraction. Smaller designs could be represented in low-level transistor level models and still be able to verify the operation through simulation. Less complex mixed signal designs could get by with verification of analog subsystems as a simple C model embedded in a digital environment. As design size grows the added complexity requires the representation of subsystem at a level that can simulate large designs and still be accurate enough to verify mixed-signal integration. Behavioral models provide the ability to represent the design at higher levels of abstraction with the necessary accuracy.

Behavioral models are developed by subsystem developers to model the higher level operation of the analog subsystem. These models can be written in HDL like languages such as Verilog-AMS or VHDL-AMS. They can be modeled from the behavioral algorithm level down to the transistor level allowing the developers to refine the models as they develop them. Technology is also becoming available to capture the characteristics of existing components in a behavioral model automatically.

7.3 MIXED-SIGNAL SIMULATION

Traditionally analog and digital subsystems have been developed separately and not verified together until the design has been built. The growth of silicon capacity and the merging of analog and digital onto the same chip now requires that these subsystems be verified before the design is realized. The unified verification methodology addresses this need through mixed signal simulation. Today there are two basic types of mixed signal SoCs. The first type is a large digital subsystem with one or two small analog blocks, often referred to as a “big-D little-A design”. These designs are digital centric and the integration concerns are speed, capacity, language-based design, and standard components. The second type of SoC designs are a large analog design with a processor or digital control unit, often referred to as a “big-A little-D design”. These designs are analog centric and the integration concerns are speed, accuracy, netlist-based design, and standard libraries.

The key to bridging the needs of the analog and digital development teams to verify the integration and operation of mixed-signal SoCs is the use of a heterogeneous simulator with native support for different design languages. The simulations must be fast and large enough to support thorough digital verification, while at the same time being accurate enough to support detailed analog verification. The important first step in this process is identifying the scope of the simulation. Often analog and digital subsystems interface at a localized point such as an analog-to-digital converter or a digital amplifier. The simulations are centered on the verification of this localized interface. Once the interface is selected and the scope of the simulation is narrowed the integration teams concentrate on the verification of three main areas. First, the teams verify correct connection between the blocks. Second the teams verify correct functionality of the interface for each domain. Finally, the teams verify the accuracy of the response meets the specification. More information can be found in the application note [“Creating an Analog Verification Testbench.”](#)

7.4 TOP-DOWN FLOW

The top-down methodology begins with the development of a test plan and the creation of behavioral models of the major analog blocks. These behavioral models allow for the analysis of the system effects on the analog subsystem as well as modeling the behavior of the analog subsystem. The models are included into the FVP for used by other subsystems testbenches. As the analog subsystem is verified and refined at lower levels of abstraction the behavioral models are updated in the FVP. The unified verification methodology speeds the verification of the analog subsystem by utilizing mixed signal simulations to simulate across domains providing a common design and verification environment for the different levels of abstraction.

7.4.1 Methodology steps

- Develop subsystem test plan
 - Develop test strategy
 - Define system and environment components needed
 - Identify system interactions to be verified
- Create behavioral models
- Verify models in FVP
- Model subsystem components
- Verify subsystem interactions
- Refine models
- Verify the individual components
- Integrate and test components to the behavioral model

7.5 BOTTOM-UP FLOW

The bottom-up methodology begins with the development of the test strategy and the collection of low-level subsystem component models. After the individual low-level subsystem components have been verified in isolation and have been integrated together the behavioral level models can be developed. There is a great amount of research underway to automatically extract the behavioral models from existing devices or low-level models but at this time the behavioral modeling may have to be done by hand. Once the behavioral models are available they subsystem can be verified for interactions between blocks and between subsystems. The models are updated in the FVP for use by other subsystem teams.

- Develop subsystem test plan
 - Develop test strategy
 - Define system and environment components needed
 - Identify system interactions to be verified
- Collect low-level subsystem components and verify
- Integrate low-level components
- Create behavioral models
- Model subsystem components
- Verify subsystem interactions
- Verify models in FVP

8 EVOLUTIONARY MIGRATION STRATEGY

Implementing a new verification methodology is not done in a single step. Development teams must take many things into consideration including their available resources, skill sets, verification needs, and often most importantly their existing methodology. Many engineer-years of effort has gone into developing existing methodologies and they should not be thrown away. Each company should develop a plan for migrating from their existing methodology to a more unified methodology like the unified verification methodology through a series of steps that leverage the work already done.

The first step in a migration plan is to create a methodology team with members from the different verification processes, different design domains, different projects, and if possible design chain partners. This team should review the existing methodologies capture what works, what provides differentiated value, what isn't working and what provides little differentiated value. The review will most likely identify many "low hanging fruit" types of changes to the methodology that can be done immediately. The team should resist the urge to only address a few of the areas of most need as this will often lead from one fragmented methodology to a different fragmented methodology. Instead the team should focus on the process of unifying their entire methodology.

The second step in the migration process is to review the specific software applications and engines used as a platform for creating the desire methodology. A successful methodology is built from a platform that meets the requirements of the methodology. The third section of this paper presented many of those requirements. The team should next develop a plan to move their existing methodology to a unified methodology in the areas that they have identified as most important. A useful method for doing this is to model the methodology on existing best-practices as described in this document. This provides the team a goal to work towards and benchmark their progress.

GLOSSARY

Acceleration-on-Demand

The ability to move from a software simulation based test environment to a hardware accelerated simulation based test environment.

Algorithmic-based digital design

A digital logic design that is directly developed from an algorithm or protocol and does not contain control based operations.

Analog behavioral model

A model of an analog circuit that represents the behavior of the implementation but does not include the implementation-specific information.

Application assertion

An assertion used to specify an application specific architectural property such as fairness of an arbiter.

Application coverage

A measurement of the percentage of application coverage monitors that have measured an event.

Application coverage monitor

A device to monitor the number of times an application-specific event has occurred.

Architectural checks

Checkers that verify the correct functional and performance operation of the FVP

Assertion

A codified representation of a designer's or architect's intent when creating a design. Assertions specify a property or behavior in a structured manner that can be verified to be correct.

Bottom-up development

An approach to development starting at low-level implementation blocks and integrating the blocks together to form system-level representation.

Control-based digital design

A digital logic design that is developed from a specification and not strictly based on an algorithm or protocol.

Design hierarchy

The naming of the design's hierarchical levels in a system. A system is made up of subsystems which are made up of design blocks which are made up of design units.

Device under verification (DUV)

The block, system or subsystem being verified

Evaluation

A system verification technique where an implementation of a design is mapped into a hardware device that emulates the operation of the design at faster speeds than simulation. The device provides standard interfaces to connect the design to real world interfaces.

Functional virtual prototype (FVP)

A golden functional representation of the complete DUV and its testbench.

Implementation-level model

A functional model of the design in which the structure and communications interfaces are defined to be implementation specific. These models are often referred to as Register Transfer Level (RTL) models.

Interface assertion

An assertion used to specify the protocol and handshaking of an interface between two blocks.

Interface coverage

A measurement of the percentage of interface coverage monitors that have measured an event

Interface coverage monitor

A device to monitor the number of time an interface event has occurred

Interface monitors

A testbench component that passively monitors an interface looking for signaling and protocol errors.

Mixed-signal design

Designs that combine analog and digital logic.

Response checker

A testbench component that compares the output of the DUV to the expected response to verify correct operation.

Response generator

A testbench component that responds to requests made by the DUV.

Single-kernel architecture

The ability to natively support all design and verification languages from the same simulation engine.

Stimulus generator

A testbench component that creates stimulus and sequences its delivery to the DUV

Structural assertion

Used to specify the operation of low-level implementation-specific structures in a design.

Structural coverage

A measurement of the percentage of structural coverage monitors that have measured an event.

Structural coverage monitor

A device to monitor the number of times an implementation structure event has occurred.

System on chip (SoC)

An integrated circuit with an on-board processor, memory along with one or more standard interface blocks or application specific blocks.

Top-down development

An approach to development starting at a high-level representation and partitioning down to lower level implementation blocks.

Transaction

A unit of information, abstracted from a lower signal-level representation, used to represent an information transfer separate from the mechanism of transfer

Transaction-level model (TLM)

A functional model of the design in which communications interface is in the form of transactions.

Transaction taxonomy

A classification of the types of transactions used throughout a design.

Transactor

A testbench component that converts different levels of interface abstraction such as signal-level interface to transaction-level interface



Cadence Design Systems, Inc.

Corporate Headquarters

2655 Seely Avenue

San Jose, CA 95134

800.746.6223

408.943.1234

www.cadence.com

© 2005 Cadence Design Systems, Inc. All rights reserved. Cadence, the Cadence logo, and Verilog are registered trademarks of Cadence Design Systems, Inc. Open SystemC Initiative and SystemC are registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission. All others are properties of their respective holders.

6050 02/05