STAR-IP CENTRIC PLATFORMS FOR SOC

ARM® PrimeXsys™ Platform Architecture and Methodologies

Jay Alphay, Chris Baxter, Jon Connell, John Goodenough, Antony Harris, Christopher Lennard, Bruce Mathewson, Andrew Nightingale, Ian Thornton, Kath Topping

ARM Ltd

Abstract: We describe the use of star-IP core-based subsystems as the cornerstone of a platform-based design paradigm. An ARM platform is an instantiation of a set of carefully market-targeted architectural-decisions encapsulated in an embedded and configurable subsystem consisting of an ARM core, AMBA™ communications fabric and a ported operating system (OS). Around this pre-specified sub-system, a derivative-product development-package is supplied. This development package provides for configuration and extension of the platform during the creation of an optimized and differentiated system-on-chip (SoC) design. We describe the structure of this development-package, and its foundation in a set of mutually consistent model-views of the platform design. Each platform model provides the speed and visibility required for specific SoC development tasks: hardware integration and development, hardware dependent software development, application software development, and system verification and validation. In this chapter we describe both the theory of platform support, and a specific ARM instantiation of this: the ARM1136JF-S™ PrimeXsys Platform.

Key words: ARM, AMBA, PrimeXsys Platforms, Star-IP, Modeling, Verification and Validation, Standards, Interfaces

1. CORE-BASED ARCHITECTURES

This chapter describes the conceptual framework supporting ARM's platform-based design solutions, and relates this to a deliverable ARM product: the ARM1136 PrimeXsys Platform [1]. The authors of this chapter are the lead technical and product architects of ARM's platform solutions. ARM Limited is a well-established provider of 16/32-bit embedded RISC
microprocessor solutions for embedded-systems design, of which platforms are a cornerstone. ARM was founded in 1990, and rapidly established industry leadership in licensing of high-performance, low-cost, power-efficient RISC processors to international electronics companies. ARM extended this intellectual-property (IP) portfolio to better enable customers' SoC design experience. Initially, this extension focused on providing a broad set of peripherals through the ARM PrimeCell® library, and more recently this has matured into supply of full SoC platform solutions around the ARM PrimeXsys Platform product family. ARM's value proposition is not built solely on the excellence of its design IP, but also in provision of IP integration support. For our platforms as well as our core products, we supply comprehensive models, debugging and hard-prototyping environments, extendible verification and validation IP. To further enhance integration ease of ARM IP, ARM is an active contributor to, and provider of, open standards. ARM is extensively involved in the Virtual Socket Interface Alliance (VSIA™) [2], Open SystemC™ Initiative (OSCI) [3,4], Accellera [5] and is the provider of the AMBA Bus [6] interface standards.

In this chapter, we explain the layered approach to SoC platform based design adopted by ARM, both for internal usage and for support of external customers. The word 'layered' implies that several distinct, but functionally consistent, views of the platform are built and maintained during the SoC architectural-design and implementation process. ARM supports three specific platform layers, as depicted in Figure 9-1. At the lowest layer, the hardware layer (Platform Layer 1), there is a micro-architectural representation of the compute-engine - the interconnected hardware that runs the embedded software; e.g., the register-transfer level (RTL) description of an SoC. Above this is the SoC-integration and middleware layer (Platform Layer 2) that configures the operating-system (OS) to the hardware architecture and encapsulates the hardware-dependent software layer; e.g., specific OS port with memory and peripheral drivers. Finally, there is the software that targets the platform to a specific domain, the application software layer (Platform Layer 3); e.g., a software stack performing graphics acceleration for the multimedia space. The adopter of the platform has the ability to configure the platform at any of the implementation layers. There are several ways in which this custom configuration may be achieved: by providing application-oriented software IP; by extending the platform with application specific design IP that may take the form of: memory-subsystems, custom peripherals, digital signal processing (DSP) accelerators; and by configuring the platform itself in terms of arbitration policies, bus hierarchy, and memory maps.

Integration of the custom-configured platform into a product may be viewed as a final or fourth layer that is built on the three-layer platform
support. Platform based design provides the IP consumer with rapid access to the product-layer, improving time-to-market (TTM) whilst providing suitable customization options for product differentiation.

![Platform Layering](image)

**Figure 9.1. Conceptual Platform Layering**

In Figure 9.1, the abstract platform layering concept is depicted on the left-hand side, with an example 'customer' configuration of the PrimeXsys Platform on the right. Against each of the platform layers introduced above, the main design elements of that layer are defined. At Platform Layer 1, hardware-integration, the critical elements in the definition of a platform are: the bus architecture, including bus-hierarchy, bridging and interface protocol; the core support, both for control and signal processing; the memory architecture, including the local and global connectivity matrix, memory controller policies, caching policies and memory-space mapping; and the hardware support for the SoC, including such components as trace and debug structures (cross triggering, trace-buffering, etc.), watchdog timers, and direct-memory access (DMA) support. At Platform Layer 2, the SoC integration layer, the IP involved is the OS ported to the hardware layer, the hardware-dependent software drivers and the peripheral bundles (hardware object for integration to the hardware layer, provided with a driver that may be configured to the OS and system memory map). At Platform Layer 3, the software development layer, are the complex stacks involved in control of SoC communication (e.g., a telecommunications communications stack), and the application software that programs the SoC to product intent (e.g., PDA user tools like calendars, etc.).

The most essential elements of a platform are depicted in Figure 9.1 as the gray boxes, with the platform extensibility and configurability shown by the white solid boxes. ARM's platform products are defined with a particular configuration of the IP in the gray boxes, and the design is rapidly extendable though use of ARM's PrimeCell® or 3rd-party AMBA-compatible peripheral libraries.
As a platform provider, it is not sufficient to purely offer a platform implementation and a compatible IP library. The benefits of platform-based design only emerge through comprehensive exposure and support of the platform’s configurability and extensibility. Customers require a guarantee of efficient IP integration, and that an efficient representation or ‘view’ of the design supports each platform layer. Each view of the platform must provide sufficient design resolution for the development requirements of a platform layer, and sufficient execution-speed (i.e., equivalent cycles-per-second) to comprehensively validate architectural and design decisions for that layer. Primarily, the representations that must be provided in support of a platform are views for: hardware integration, SoC hardware/software validation, SoC architectural exploration, and application software development (hard and soft prototyping).

ARM provides the following design infrastructure in support of platform-design views:

1. interfaces: standardized interfaces for design, verification and model IP that support product integration at each of the three platform-design layers.
2. models: core and platform.
3. software development support: hard and soft prototyping with a debugging environment that supports re-targeting.
4. verification/validation (V&V): reusable components and methodology, and interface protocol compliance checkers.

This chapter introduces the reader to the concepts of core-based platform support based around the four key aspects listed above. We begin with a discussion of the principles of platform IP exchange (Section 2), and the need for standardization to enable IP and platform reuse (Section 3). We then proceed with a description of the support required for the platform layers: Platform Layer 1: hardware development and integration (Section 4), Platform Layer 2: hardware-dependent software development (Section 5), and Platform Layer 3: application software development (Section 6). In parallel with the design process, verification and validation is supported (Section 7). As ARM does provide all these facilities in support of our platform products, we describe a specific platform product as an example: the ARM1136 PrimeXsys Platform (Section 8). Finally, the chapter closes with a set of conclusions (Section 9).
2. ADOPTING A PLATFORM: THE USER VIEWS

2.1 Introduction

Adoption of a platform requires the communication of design requirements and implementations between multiple users. Models of a platform are critical for enabling this communication. As an example, consider embedded software development kits: the embedded OS company Symbian® supplies a tool-kit for developing applications for Symbian OS [7], but the execution environment on which you develop with this kit is your desktop PC. There is no guarantee that software developed within this environment will directly port to the final product unless there has been a model of the platform on which to execute the software. A platform model provides an executable description of the SoC architecture enabling a high-degree of debugging (better than physical designs), and the flexibility to explore the interplay between software and SoC design requirements. In this way, modeling enables technical communication between the embedded software development teams, and the SoC architectural specification and design teams.

In this section, we will describe the users and adopters of the platform through the design flow from system-design to physical implementation, exposing the required communication and IP exchange channels for assembling of a complete embedded system. Throughout the remaining sections of this chapter, we then build upon these requirements to provide a methodology that supports the technical and business constraints of complex IP exchange.

2.2 Exchange of Platform IP

Platform IP deployment is often seen as a one way street where the IP creator licenses (hopefully) high quality IP to a licensee who integrates this into their design. Simple IP is delivered with perhaps implementation code, simulation models for RTL design and integration guides documenting the function of the platform. For simple hardware-centric components whose functionality is well understood and for which there is little or no software included in the package, this often suffices.

In the more complex case of platform IP, this simple view of IP licensing does not sufficiently capture either the complexity of the relationships between the creator and the user(s) of platform IP, or the difficulties in transferring different views of platform IP between creator and the user(s) of the IP. Required communication between the parties is often complicated by the need for concurrent development of a product.
2.2.1 Platform deployment roles

The deployment of platform IP for multiple licensees who, themselves, target yet more end users is best achieved through use of a streamed, and possibly concurrent, approach to the development of a product. We identify three distinct roles: the IP Creator, the IP Licensee and the IP User.

- IP Creator: creates the base platform and ports suitable operating systems to facilitate the exploitation of the functionality of both hardware and software by the licensee and end user;
- IP Licensee: extends the platform through differentiation, targeting the base functionality to specific application domains, adding their own hardware and software IP;
- IP User: integrates the differentiated platform into a product, making use of the wide range of third-party support attracted by the base platform.

The organizational relationship of each of these three contributors to the deployment of IP has significant bearing on the requirements on that IP. In Figure 9-2 we identify a chain of IP deployment in which each of the three roles can be represented by separate companies or different parts of the same organization:

![Figure 9-2. Three roles in IP deployment](image)

Wherever exchange of IP involves separate organizations, the security of the IP itself becomes important as the IP Creator is likely to have no control over the IP Licensee's choice of IP User. The IP Creator must therefore provide a number of discrete but consistent views of the IP that enable the design processes to be employed by a wide range of IP Users. For example
in the case of a simple hardware component, these 'black-box' views may consist of: a compiled cycle-accurate model for RT design; a compiled phase-accurate model for netlist integration; and a power-model, timing-model, floor-plan and routing-constraint model for physical integration. The set of views which must be provided for complex subsystem IP that includes hardware, software and verification support is significantly more extensive than that required for basic hardware IP, and this creates a significant increase in the scale of the problem for successfully deploying a platform.

2.2.2 Design process interaction

The end users involved in platform IP deployment do not work with that IP in isolation. For a typical product, the IP Creator will create a base platform; the IP Licensee extends that platform adding unique value that attracts customers; and finally the IP User will deploy the differentiated platform in a product.

For IP products to be successful, they need to be attractive to as wide a range of licensees and end users as possible, but in the initial stages of specification and design, an IP Creator and IP Licensee will typically work together with strategically important IP Users to prove the value of that platform. The user interactions depicted in Figure 9-3 identify a number of exchanges of design representation throughout the development of IP. These interactions correspond to well-defined stages in the SoC implementation process, as indicated by the arrows between the idealized top-down development flows [8].

In development of their part of the completed product, each user in the design chain will seek to define their system architecture and then decide how that architecture is best represented by hardware and software components. The correct interaction of those hardware and software components must be proven during the design stage before implementation decisions are made. For the ARM PrimeXsys Platforms, each base platform is provided together with operating system ports and the 'correct interaction of hardware and software components' will include the porting of device drivers and booting of ported operating systems. We describe how this is achieved in Section 5 of this chapter.
Consider the context of interaction between each of the users in the deployment of the platform, Figure 9-3. The results of the IP Creator’s micro-architectural investigation will be used as the basis of the architectural decisions of the IP Licensee. In turn, the specification of the licensee’s differentiated platform will then be used by end users (IP Users) to define the micro-architecture of their products.

Each view of the platform reflects the same system architecture, and designers can use test software in any of the higher-level views, providing a high degree of confidence in the design prior to tape out. This provides a valuable environment in which to investigate system bandwidth and performance requirements. System views must be extendible, allowing designers to exploit the advantages of a well-supported, pre-verified base platform of hardware and software IP, whilst differentiating their own application with their own IP.

Additionally, there will also be a transfer of validation IP. This is critical to the success of the platform since the recipient of the platform IP will need to ensure that any extensions they add to the base platform and to the differentiated platform do not invalidate that core functionality. Were the
licensee unable to prove that his differentiated platform still implemented the base functionality of the licensed platform IP, the IP User would be unable to exploit the wide variety of third party tools that are attracted to a standard base platform. How a consistent and reusable verification and validation methodology is achieved for complex hardware and software IP is described in Section 7 of this chapter.

3. PLATFORM INTERFACES: STANDARDS

3.1 Introduction

Platform, and design and verification reuse in general, hinge upon the existence and broad adoption of interfacing standards. The obvious standard critical to IP integration is a bus interface, for which ARM has created the AMBA open-standard framework [6]. In this section, we describe the mechanisms by which ARM supplies and contributes to industry standardization to better enable IP integration.

Standard bus interfaces are critical to efficient hardware IP integration, but these are only one of several well-qualified interfaces that a platform must expose. We first describe the general open-interface issue as it relates to platforms, then address more specifically hardware IP communications channel (e.g., AMBA bus) that ARM platforms support.

3.2 ARM standards generation and engagement

There are three different types of standards that ARM provides or engages with. Firstly, there are the product-deliverable or 'platform standards' that may include ARM-proprietary interfaces. Secondly, there are the two forms of open-standards with which we engage: ARM-community (e.g., AMBA) and open-committee (e.g., Accellera, OSCI, IEEE, VSIA) standards. We describe each of these in more detail here.

Platform standards describe the requirements for interfacing IP (hardware and software design, models, and debugging environments) to our platform design, and its soft-prototype and hard-prototype instantiations. Platform standards apply across all ARM platform products. The ARM platform customer may choose to structure their internal design and verification flows to conform to these ARM platform standards, thereby assuring compatibility with ARM's future platform products. There are eight basic platform product standards, described in the following subsection. Some of these interfaces are private and provided with the platform deliverables (e.g.,...
debug cross-triggering requirements), some are ARM-community standards (e.g., AMBA 2.0), and some adopt the principles of the open-committee standards (e.g., Virtual Component Transfer (VCT) Specification of VSIA).

Open standards provide common structures across the electronics design industry that enable IP and tool providers to support the IP integrator. The output of open standardization efforts must be attainable and usable in a non-restrictive way. The difference between open-committee and ARM-community standards is the mechanism of standards creation. Both styles of open-standards structures must exist, and both play an important role in enabling platform products.

The ARM-community standards are of foremost importance to ARM. These express the formats, interfaces and packaging choices to which all our IP products are offered. ARM-community open standards ease basic integration issues by ensuring an IP and EDA-support base that offers a broad range of solutions to ARM customers. These standards describe our lead products, so are driven by firm time-to-market requirements. The ARM-community standards are jointly developed by a representative group of ARM partners. These are comprehensively validated through reference implementation before release, then opened for free to the public under a simple non-restrictive license. These licenses are materially similar to the AMBA 2.0 license agreement. This approach provides a good balance between support-breadth, speed of generation, and clarity of ownership.

Besides driving ARM-community standards, ARM commits significant time and effort into engagement with open-committee standards. The emphasis of open-committee standards is broad-based consensus rather than product-support time-to-market requirements. Broad-base consensus is important for industry-wide standardization of a maturing technology or the encouraging of standardized interfaces and formats that affect ARM-IP adoption indirectly. These open-committee standards also play an important role in closing the gap between a set of industry de-facto standards that have become non-differentiating. Through active encouragement of open-committee standards, ARM is ensuring that platform delivery is possible into multi design-language environments (e.g., Verilog / VHDL co-simulation support), and multi-language validation environments (e.g., standard interfaces for verification components, and consolidation of in-line assertion languages).

### 3.3 Platform integration standards

In the following table, Table 9-1, we describe the eight platform standards to which platforms should be provided. This list covers a comprehensive set of concerns for customer integration of IP into a platform,
or integration of the platform IP into a customer’s SoC design flow. In the table we list the integration requirement in the left-hand column. Following that is an indication of whether the requirement is an interface (e.g., API or signal/timing definition) or format (e.g., mechanism of description). Finally, in columns three and four, we define whether the PrimeXsys Platform product family currently exposes these integration requirements explicitly in a private (platform standard), and/or public manner (open standard).

Briefly, the eight platform standards are:

(i) Bus – the protocol (signal definition and timing), as well as extensibility of the bus-hierarchy provided in the platform context.

(ii) OS Configuration Layer – call-structure for upper middle-ware embedded software components. For example, call structure for peripheral drivers and API for hardware-agnostic software implementations (i.e., using memory-map description for the SoC).

(iii) Energy Management – at several layers of integration hierarchy, from exposure of clocking-grids at the physical level to enable gated-clocking, to energy-consumption reporting and a management policy controlled by a dedicated power-control unit at the systems level.

(iv) Memory Structures – description of memory-map and hierarchy to allow user configuration, as well as auto-generation of integration test.

(v) Trace / Debug – at several layers of integration hierarchy, from core integration requirements (e.g., embedded trace managers, and cross triggers), to model API. Also includes structure for capturing of data.

(vi) IP Packaging – both for: the delivery of a complex piece of IP including multiple levels of design hierarchy, models, and design scripts; and tool recognition of structure of a piece of IP (e.g., interface definitions).

(vii) Model Interfaces – interfaces for protected design sign-off models, as well as co-validation and soft-prototype models.

(viii) Verification IP – interfaces to allow an extendible SoC verification environment to be provided (i.e., synchronization of tests), and format for auto-generation of tests (i.e., actions and file-readers).

The (dev) superscript in the table below indicates interfaces in current development within ARM’s platform-based design initiative, described further in Section 4.4. The asterisk (*) indications are areas of currently active standardization work. ARM is expecting to provide these as ARM-community open-standards during 2003. For IP Packaging, an XML description standard is being generated under an ARM-community model,
whilst we are also actively engaged in the efforts of the VCT group of the VSIA. For model interfaces, an AMBA 2 Transfer-Level SystemC Interface for SystemC models is now provided under an AMBA license agreement, while a transaction-level interface standard for abstract models is currently in progress. For verification IP, there is active collaboration between several major verification environment suppliers towards a common verification methodology and verification-component synchronization interface.

<table>
<thead>
<tr>
<th>Description</th>
<th>Interface or Format</th>
<th>Open Standard</th>
<th>Platform Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus (AMBA)</td>
<td>Interface</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OS Config</td>
<td>Interface</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Energy Management</td>
<td>Interface</td>
<td>No</td>
<td>Yes^dev</td>
</tr>
<tr>
<td>Memory Structures</td>
<td>Format</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Trace / Debug</td>
<td>Interface &amp; Format</td>
<td>No</td>
<td>Yes^dev</td>
</tr>
<tr>
<td>IP Packaging</td>
<td>Format</td>
<td>No*</td>
<td>Yes</td>
</tr>
<tr>
<td>Model Interfaces</td>
<td>Interface</td>
<td>Yes*</td>
<td>Yes</td>
</tr>
<tr>
<td>Verification IP</td>
<td>Interface &amp; Format</td>
<td>No*</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The following sub-section describes in further detail the interface standards used and being developed for hardware and model communication: the AMBA protocol family. Further description of some of the platform-standards interfaces is provided in Sections 4 and 8.

3.4 Component communication infrastructure

Since 1994, ARM has invested in the generation of open bus protocols for enhanced IP integration. As SoC modeling becomes a more crucial part of platform delivery, this investment will extend to the model interfaces for support of hardware-dependent software development and application software development. We describe the AMBA protocol family, then the modeling interfaces above that which will need to be openly standardized.

3.4.1 AMBA protocol family

AMBA is an open-standard, on-chip bus specification that details a strategy for the interconnection and management of functional blocks that makes up a System-on-chip (SoC). It is widely recognized as the most commonly used on-chip bus standard [9] and therefore boasts a wide range of ARM and 3rd party support including: bus fabric IP; CPU and peripheral IP; validation, verification and system design tools.
The AMBA specification encompasses a number of protocols intended to satisfy a range of performance, connectivity and power requirements. The latest revision of AMBA (3.0) introduces the highest performance AXI, together with existing AHB and APB specifications.

The Advanced eXtensible Interface (AXI) has de-coupled address and data buses, enabling them to be optimized independently. It also supports out-of-order completion and multiple outstanding transactions, maximizing data throughput and minimizing latency. The AXI specification defines the bus interface and protocols only, enabling the end user to choose or design the interconnect to match their bandwidth, connectivity and power requirements. For example, if out-of-order transaction completion is not required, then this can be left out of the interconnect whilst leaving the peripherals unmodified. Each link in an AXI bus topology is considered a point-to-point master/slave connection which implies that the interconnect is readily extensible and can be optimized at various points in the SoC design flow.

As well as the basic data transfer protocol, the AXI protocol also includes optional extensions to cover DMA, interrupt and low-power operation signaling.

The Advanced High-performance Bus (AHB) was the highest performance bus in the AMBA family before AXI was developed. It is suitable for medium complexity and performance connectivity solutions and currently has the highest levels of 3rd party IP support.

AHB-Lite is a subset of the full AHB specification and is intended for use in designs where only a single master is used. This may be a single master system or a multi-layer AHB system where there is only one AHB master on a layer.

The Advanced Peripheral Bus (APB) is designed for ancillary or general-purpose, register based peripherals such as timers, interrupt controllers, UARTs, I/O ports, etc. This is generally connected to the system bus via a bridge, which helps reduce system power consumption. The APB is very easy to interface to, with little logic involved and few corner-cases to validate.

3.4.2 Interface abstraction hierarchy

Platforms are complex pieces of design IP that must be provided with a set of soft-prototype versions. The reason that soft-prototype modeling of a platform is necessary is two-fold: (i) provide the correct level of design detail for the task performed against the model, and (ii) provide sufficient simulation speed for the design task to be performed. There are four basic tasks (use-models) that must be supported by soft-prototype models of a platform: (i) top-level architectural definition and embedded software
development (supports: Platform Layer 3), (ii) micro-architectural exploration and middle-ware development (supports: Platform Layer 2), (iii) system, or micro-architectural, validation (supports: Platform Layer 2), and (iv) component implementation and integration/verification (supports: Platform Layer 1). Each soft-prototype provided with a platform must support a standard interface for user extendibility of the model.

Four levels of system abstraction are defined to cover the use-models we have just described. In Figure 9-4 below the system-level design layers are depicted against the basic SoC development flow, as first introduced in Section 2. The set of system-level abstractions that we use share many similarities with those described in a technical submission to OSCI [10], and

![Diagram showing SLD Abstractions and Channel Resolution]

**Figure 9-4. System-Design Flow against Platform Abstraction Levels**

In Figure 9-4, the RT level of abstraction for the platform design is depicted in dark-gray, and constitutes the hardware implementation level. Above the RT level, the (loosely termed) ‘transaction levels’ of soft-prototyping are shown. At these levels of abstraction, we envisage that model extensibility will be provided through implementation of interfaces into a common open standard language, primarily SystemC. The platform models themselves may be provided in a separate modeling-specific format (not SystemC) that is more efficient for internal-core simulation, but which can be readily interfaced to the SystemC simulation kernel.

In the abstract system-modeling environment, three abstractions must be supported:
The (TF) Transfer Level – a cycle accurate translation from register-transfer level to transfer-level transactions. The simulation speed of the communication is gained through the efficient handling of abstract types and assembling of atomic (non-interruptible) action sequences into address/data transfers. This abstraction uses clock-based execution semantics, and is directly mappable into RT signals. For cycle accuracy, both blocking and non-blocking interface semantics must be supported. This modeling abstraction will execute in the 100 kCPS (cycles-per-second) range, sufficient for system validation.

The (TX) Transaction Level – a cycle-count accurate model of the system. That is, a datum or block-data request is completed (returned data or time-out/error) in single transactions, and time is indicated as ‘time-passed’ rather than events-per-clock-tick. Unlike at the transfer-level, there can exist models of the SoC in which only blocking communication actions are performed. The models of the bus and interfaces are sufficiently accurate as to be characterized by bus protocol and bus-hierarchy (i.e., domains, layering within a matrix, bridges, etc.). This modeling abstraction will execute in the 1 to 5 MCPS to provide for middle-ware development and micro-architectural exploration, such as bus and memory-management configurations.

The (MS) Message Level – a register-accurate level, with no or very abstract (e.g., “function call took y-time to execute”) timing provided. There is no channel modeling between the components at this level as point-to-point communication is used, with the exception of shared memory spaces. This modeling abstraction is sufficiently accurate for embedded software development as it exposes the OS configuration layer and a register-level, or programmers-model, description of the SoC. This modeling abstraction will execute in the 10 to 100 MCPS.

For each level of abstraction, a system-level interface must be defined. These interfaces should be open standards that are expressed in a comprehensive manner to describe not only the communication API, but also the mechanism of refinement from one abstraction to the next [12]. The concept of interface hierarchy is depicted in Figure 9-5 with the system-abstraction levels defined on the left-hand side of the figure. This figure shows that, while there may be many specific protocols at the RT level, these can be reduced into a smaller set of configurable interfaces as the design is abstracted. At the top-level, the system model becomes bus-generic, though this generic transport-layer can be configured to express the ‘protocol personality’ of a bus. A protocol personality will include, for example, length of burst support, details of request/grant procedures, error-handling, side-band information, etc.
Besides the interface hierarchy, the most important point expressed in Figure 9-5 is the delineation between the system abstraction levels. There are two significant boundaries, one between the RT level and TF level that is a language boundary (i.e., multi-level simulation will imply multi-language) but otherwise a straightforward translation; and one between the TF and TX levels, a communication mechanism boundary. The distinction between the TF and TX levels of abstraction is a fundamental mechanism of model-execution: a clock-based execution semantic is used at Levels RT and TF, and a synchronous-reactive execution semantic is used in Levels TX and MS. Linking or refining models across the TF/TX boundary is not intuitive due to the lack of an atomic-unit of time in the TX/MS abstractions, unlike the clock-period for simulation advancement in the TF/TX abstractions.

Although a separation between the event-based TX and MS levels also exists, this difference can be characterized within the 'protocol personality description' that enables transfers to be weighted with an appropriate delay. There is neither a difference in execution semantics nor language in moving across the TX/MS abstraction boundary.

Models of ARM platforms are provided conforming to the interface hierarchy described above. ARM is currently providing SystemC extensibility to its Instruction Set Simulator models that enable integration into SoC environments supporting MS and TX abstractions. We also provide as an open-standard the AMBA 2 SystemC Transfer-Level Interface Specification that defines requirements for building of AMBA interface class-libraries (ACLs) in SystemC. With partner companies, ARM is actively investigating support for qualified translation from TX level models.
to AMBA 2 and 3 interfaces at the TF levels and below. Application of the modeling abstractions to the platform-design process are detailed in Section 5 and Section 6 of this chapter.

4. **DESIGN SUPPORT: HARDWARE INTEGRATION**

4.1 **Introduction**

Hardware integration is eased by the provision of standardized interfaces, as described in Section 3.3. However, standard interfaces are not sufficient for providing an efficient hardware platform integration strategy. Platforms must be able to be provided interface-protocol checkers (for validating compatible IP design such as the AMBA Compliance Test-bench (ACT)), integration examples (such as the AMBA Design Kit for bus architectures), and SoC resource allocation tools (such as PMAP for memory allocation). This section describes how ARM addresses the most critical hardware integration issues for soft platforms. In this section, we describe the platform-based design support provided for hardware integration, Platform Layer 1. Hardware integration and verification is supported by the clock-based RT and TF system model abstraction levels.

4.2 **Bus-generation and hardware integration**

To enable customer integration of hardware components, ARM provides two products: the AMBA Design Kit (ADK), and the AMBA Compliance Test-bench (ACT).

The ADK provides a generic, stand-alone environment to enable the rapid creation of AMBA-based components and SoC designs. Containing a rich set of basic components, example system designs, example integration software and synthesis scripts, the ADK provides the common foundations for product design based upon the AMBA interface. This frees up engineers to focus on application-specific issues and "value-add" components.

The ADK is comprised of over 50 fully validated components, which can be broadly categorized as interconnect (e.g. decoders, arbiters, bridges), reusable peripherals (e.g. interrupt controller, timers, watchdog), example peripherals (master and slave), ARM CPU wrappers and test components (BFM, memory models etc). All are available both as VHDL or Verilog HDL code.

The ADK introduces the concept of "black-box" interconnect, where an AMBA interconnect can be auto-generated based on a number of parameters
and plug in arbitration and decode schemes. This is then presented to the system integrator as a black-box of master and slave ports to which can be attached peripherals or bridges to other interconnect.

ARM’s ACT enables IP component developers to demonstrate that its AMBA interface is fully compliant with the specification [13] and can therefore be seamlessly integrated with other compliant IP. An IP component is granted AMBA compliance when it has been observed to experience all of a predefined list of protocol scenarios (coverage points) without breaking any protocol rules. ACT is integrated into the HDL simulation to provide this capability of AMBA protocol checking and coverage analysis. The protocol and coverage checkers gather the evidence required to grant AMBA compliance to the device under test (DUT).

**Figure 9-6. Architecture of the ARM ACT product**

Figure 9-6 shows a generic ACT structure. When configured in ‘active-mode’, ACT generates a testbench for the DUT including stimulus driven bus functional models of master, slave and arbitration components. In passive mode the protocol and coverage checkers are overlaid onto an existing user testbench.
4.3 Configuring the Platform

In addition to the AMBA interconnect generator described in the previous section, the platform must support a capability to automatically generate and manage the complete address, DMA and interrupt maps. This requires provision of template-based RTL and models for the address-decoder, interrupt-controller and DMA blocks. Significant co-ordination needs to be achieved across platform components and deliverables by creating shared schemes for describing component parameterization, and providing methods for generating the configured RTL/models and verification vectors. ARM’s own production version today is known as PMAP.

PMAP, or ‘peripheral map’, defines the local register descriptions for each peripheral, including complete register bit maps and reset values. These are concatenated together by a top-level script-file that defines the platform or SoC memory map\(^9\). From the top-level script, the RTL for the top-level address decoder is generated along with integration test vectors for the SoC. The integration test vectors can take the form of either instructions for an AMBA File-Reader Bus Master (FRBM), or C-code for compilation onto a core.

In addition to automatic generation, a PMAP description for an individual peripheral can be optionally extended with a set of canned vectors to perform self-test or run initialization code. This allows basic software integration to be carried out in a simple simulation environment and is particularly useful for toggling interrupt lines to verify the correct integration to the interrupt controller.

PMAP is currently based on an in-house format and script, but it can be described with an XML-based schema. This open-format approach will encourage support of platform configuration in a standard way across a variety of EDA tools.

4.4 Extending platform control structures

As described in Section 3.3, full platform integration requires definition and standardization of interfaces to more than just the bus infrastructure and memory configuration. In particular, significant development activity within

\(^9\) It is not generally desirable to change the base address map of the platform as such a change may necessitate kernel re-write. This problem arises with OS ports when key peripherals are tightly coupled to the behavior of the kernel. It is, however, generally feasible to extend the address map to hook in communications-oriented peripherals.
ARM is being applied to two interfaces not today shipped with platform product: the energy-management interface, and the trace-debug interface.

System-level energy management is a complex task that involves communication between an application or operating system and the hardware platform. This requires a hardware-dependent software abstraction (Platform Layer 2) of the energy management system that is extendible when new components are integrated. At the hardware level (Platform Layer 1), a scheme and protocol for communicating power status from a central energy management controller to peripheral is being defined. This will permit definition of, and plug and play with, different power domains on a configured platform. A first step in this development is the collaboration between ARM and National Semiconductor to define the off chip interfaces between a power management controller and voltage control IC to effect dynamic voltage scaling.

System debug support for a platform requires both hardware and software support. ARM pioneered the use of in-circuit trace and emulation (ICE) and it is natural to extend this to the complete platform and peripherals. From a hardware perspective this requires the definition of a standard multi-master ICE and trace protocol. Trace is already being extended within ARM to provide real-time trace capability for an AMBA bus. For multiprocessor support, a trace/debug hardware construct must also define a cross-triggering standard.

The software support for system-debug is provided by ARM’s Real View Debugger product, a product that already supports a multi-master ICE and trace protocol. The Real View Debugger product allows a complete description of a new peripheral to be rapidly integrated by providing the software developer with symbolic awareness of the platform and its extensions. The Real View framework also permits the straightforward integration of source level debuggers for DSP or other processors into a heterogeneous SoC environment.

5. **DESIGN SUPPORT: HARDWARE DEPENDENT SOFTWARE**

5.1 **Introduction**

The point at which hardware-dependent software, often termed firmware, and the hardware itself comes together is often the first time a system exists in its whole form. Traditionally, this has been done using hardware prototypes where errors in the system design have been corrected with
software workarounds or, much more costly, using re-spins of the hardware. New techniques bring hardware and software together earlier in the design cycle and seek to eliminate the need for the costly hardware re-spins [14]. In this section, we define the platform-based design support for Platform Layer 2. This encompasses modeling abstractions for the clock-based TF system abstraction level and the TX cycle-count abstraction, as well as the process of model refinement. This section also includes a discussion on hardware prototyping support for hardware dependent software development.

5.2 Hardware/software co-verification

Early hardware/software co-verification circa 1995 [15] introduced embedded software to hardware simulation using retargeted code and memory integration libraries in which accesses to hardware memory were replaced by accesses to a Bus-Functional Model (BFM) simulating a processor bus interface. Instruction Set Simulators soon replaced the retargeted embedded software creating a better-integrated tool-chain. Most recently, high-speed cycle-accurate models have been introduced and provide an accurate, but efficient integration of hardware and software. Additionally, co-verification kernels have introduced sophisticated optimizations that allow designers to trade off bus cycle-accuracy for speed.

Whilst co-verification remains an efficient way to interface embedded software to an RTL representation of a design, the software is still introduced to the hardware relatively late in the design of the complete system when the RTL implementation is well under way. Co-verification is also an expensive technology that is not popular with software developers who are unfamiliar with its hardware-centric view of the system.

5.3 Hardware prototyping

As before with application software development, hardware prototyping using field-programmable gate arrays (FPGAs) or FPGA-based multi-board prototyping tools introduces software to real hardware and executes that software on a platform capable of multi-megahertz speeds. Integrated software tool-chains ensure that the software developer has a consistent view of the software running on the target regardless of whether he or she is using a software emulator, a final product or the hardware prototyping tools.

For hardware-dependent software, however, FPGA prototypes can provide too little visibility into the description of the hardware. For the system architect, they are not well suited to rapidly exploring new configurations of system hardware in the context of software benchmarking. A system designer may, for example, wish to explore how different bus or
cache architectures affect the performance of an operating system just as much as they wish to tune an operating system to a particular hardware micro-architecture. This is more directly achievable with soft-prototyping approaches, particularly transaction-level modeling.

Although hardware visibility issues exist with hardware prototyping that make it less desirable for middleware development, the execution speed of the hardware prototype is very applicable to application software development, as described in Section 6.2.2.

5.4 Transaction level modeling

Transaction-Level Modeling (TLM), as introduced in Section 4.2, seeks to bridge the gap in the current available hardware/software integration methodologies. By raising the level of abstraction of the hardware described for the system, designers can produce more efficient system simulations. Speeds of circa 100kHz for transfer-level simulation and 1MHz for transaction-level simulation are achievable using the abstraction technique in a suitable modeling language such as SystemC.

Transfer-level (TF) modeling provides a cycle-by-cycle mapping from one bus transfer cycle to its representation in a complete bus protocol. Speed increases are achieved by a combination of high-value IP models written to simulate efficiently in such environments and the reduction of a number of signal events in a protocol-cycle to a single-cycle operation. At the more abstract transaction (TX) level of design, an entire block-data transfer such as an ATM packet or an AMBA AHB burst can be represented by a single transaction. The timing of these transactions will always be known to within a reasonable approximation because the protocol is well understood. Each transaction can be converted to cycle-based timing using an adapter.

5.4.1 TLM example system

To understand the architecture of a typical TLM system, a simple example is shown in Figure 9-7 in which four IP components: 2 masters and 2 slaves are connected through a shared communications fabric supporting the AHB Multi-Layer bus protocol. This example applies to the clocked system-abstraction levels, transfer (TF) level and implementation (RT) level. The architecture exists within a single clock domain. Masters, slaves and the hierarchical channel model are all clocked components. Arbiters and decoders, as part of the bus fabric, are reactive, un-clocked models.

The communications fabric is modeled by three components: a hierarchical channel model that manages state of component connectivity, an arbiter that receives requests and allocates the shared channel based upon the
priority of the requester (master), and a decoder that resolves addressing into specific block connections (transfers from/to). The arbiter and decoder, though distributed in hardware, are expected to be modeled through use of monolithic SystemC blocks (single interface to each).

Masters in such a system might be processor models or memory controllers, whilst slaves might be peripherals such as a UART. Masters are clocked models that create data structures representing bus transactions, masters pass these structures to the bus fabric. Slaves are clocked models that are invoked by the bus fabric and are given the bus transaction data structures to operate upon.

A critical part of this methodology is the ability to migrate to RTL design and retain a verification infrastructure for the complete system. The verification methodology is described in detail later (Section 7), but the introduction of RTL models is achieved using adapters to the transfer interfaces of the SystemC models. Since there is a known mapping between AHB transactions in the SystemC world and the signals of a physical AHB, such adapters can be generic and guarantee the correct operation of the device.

A single monitor (e.g., waveform analysis tool) is shown in the example architecture, connecting through a monitoring interface to the arbitration
component. As the arbiter maintains a list of channel requests and masters the allocation of the shared channel, this component maintains specific information about the current and pending state of system connectivity not maintained elsewhere. Simple transfer monitors may also be placed on master/slave interfaces, but these are not depicted here.

6. DESIGN SUPPORT: APPLICATION SOFTWARE

6.1 Introduction

This section describes the platform-based design support for embedded software development, Platform Layer 3. The section starts with an overview of current approaches for software development on custom SoC [16] then describes the more efficient and cost-effective support possible in platform-based design. We describe both the modeling support for Platform Layer 3 encompassing the message (MS) level of modeling-abstraction, and the hard-prototyping support also critical for application software development.

For context on the delivery of platform models, we utilize the definition of IP Creator, IP Licensee, and IP User provided in Section 2.

6.2 General application software environments

Most application development environments do not adequately consider the requirements of the IP deployment strategy. Typically in the case of the three contributors to a completed platform-IP based product, the IP User will be responsible for application software development, but will do so based on an operating system port developed primarily by the IP Creator. Since there is no requirement for a business relationship between the IP Creator and the IP User, there must be sophisticated, well-supported development tools in place to support new platforms.

6.2.1 Host-retargeting

Many OS vendors provide an environment in which the OS itself is simulated using a native port of the OS (such as the Symbian WINS emulator [7]) and the application code recompiled for the host platform. Whilst this approach results in excellent software execution performance, often approaching the speed of the host platform itself, there are a number of drawbacks in this mode of application software development.
Companies employing host-retargeting will need to tool up their application developers with host compilers as well as target compilers. The resultant code is also not guaranteed to work due to the subtleties of the different representations of the OS on the target and the host. Threading models available to user processes on the host and to a kernel on the target will differ greatly and may affect the execution of applications such that correct execution on the host does not guarantee execution on the target.

In the case of complex platforms, representations of the underlying hardware must be provided to the user to ensure correct function of an application written for a specific device. Even simple device geometry issues such as the size of the screen on the host and the target can affect the application significantly. Some OS vendors have solved this problem by introducing a virtual machine that traps hardware accesses and emulates the underlying hardware using models. This added complexity typically reduces the performance of the emulation significantly, thus negating much of the benefit.

6.2.2 Hardware prototyping

Prototyping of systems using FPGAs and FPGA-based multi-board prototyping tools is becoming increasingly popular as the complexity of hardware and software increase. These tools allow high-speed execution of software on real hardware, often allowing the interfacing of physical devices such as networks that provide true physical data. In the past a drawback of FPGA prototyping for the IP Creator has been the protection of their IP. Recent developments in the area of secure FPGAs has allowed IP creators to offer pre-compiled FPGAs containing the pre-configured platform IP.

Using hardware debug and trace tools, software developers can gain excellent debug visibility into the software running on a target. Debug tools need to be aware of the underlying OS and provide information about OS-level primitives such as threads and semaphores if they are to provide sufficient debug infrastructure for an application software developer looking to debug complex OS-dependent software.

FPGA-prototypes do suffer from the complexity of new platforms and large systems and whilst newer and larger FPGA devices are constantly available, complete systems will always span a number of FPGAs. This is particularly true in the case of the complex IP deployment chain described in Section 2 as each party in the chain will seek to encapsulate their IP in their own secure FPGAs. When this proliferation of FPGA devices occurs, the performance of the resultant system decreases and falls below that required by application software developers.
Most significantly for the devices that are intended to be open to a huge range of application developers such as set-top boxes, PDAs and new cell-phones, hardware prototypes are expensive. If a product supplier is to encourage the development of software for their device, the development environment must be inexpensive or even free.

6.2.3 Instruction-Set Simulation

Instruction-Set Simulators (ISSs) are typically available for all programmable devices and often provide a low-cost route to the emulation of target software. Modern ISSs also typically include simple infrastructure for modeling the programmer's interfaces of peripheral devices so that the execution of hardware-dependent software such as an OS is supported. They operate at the MS-level of system abstraction. The models are sufficiently accurate, and sufficiently fast in execution, that application software can run on top of the OS and the application developer can debug the application using the same debug and trace tools available for a live target such FPGA prototyping.

Execution speeds of ISSs are constantly improving and the latest generation of software emulators can execute software in the performance range of some FPGA prototypes at around 10 MCPS. Since they are software products, they can be distributed inexpensively as part of a software developers' toolkit. The latest embedded products, however, require higher speed processors and the performance of even a 10 MCPS ISS is unlikely to be suitable for application development. When models of the hardware subsystem are included, execution speeds are affected as much as they are in the case of the host-retargeting OS environment.

6.3 Platform software development environments

ARM currently offers platform solutions in all three areas described above for application software development and each of these solutions provides a unique solution to the software development problem. What is not addressed by any of these environments is how to offer an execution environment in which host-targeted software can be represented at a low cost but at speeds that are suitable for application software development. Specifically:

- Host retargeting does not give the developer the confidence that is gained by targeting software for the physical hardware target and using the target OS.
- FPGA-prototyping does not provide a low-cost environment for the mass proliferation of a platform to thousands of developers.
ISSs are not fast enough to execute the complex application software and OS kernels that are present in modern embedded devices. To address the low-cost requirement of the application software development environment, a software solution is best as the costs of replication are low. The traditional software emulation provided by ISSs is insufficient in terms of performance for application software, so new emulation techniques are required. The virtual platform must include models of the hardware represented by the device and these must be presented to the developer in such a way as to emulate the physical design of the target device. In the case of a cell-phone, this might be in the form of an image or “skin” of the device that is updated by the software emulator in real time.

In terms of where such a product fits into the development cycle of a device, this does not fall into a simple top-down approach. Instead, it is likely that the high-speed platform emulator will be developed once the product is nearing the end of its development and is well understood. The IP Creator will initiate the deployment of the base platform for the emulator and encourage the IP Licensee and IP User to extend the emulator to match particular devices running specific OS ports. The goal of this deployment chain is to ensure that the platform emulator is available at the time of the launch of the product such that it can be supplied to application developers to encourage them to develop new software for the new device.

7. QUALIFICATION: PLATFORM VERIFICATION AND VALIDATION

7.1 Introduction

Although reuse methodologies in design implementation have become an essential part of the design approach for complex SoC development, reuse of the verification environment is much less common. Traditional approaches to verification do not enable reuse of the verification effort from project to project. Typically test stimulus is still handcrafted, from scratch, for each new SoC project using a mix of C, RTL and assembly language. Acceleration and emulation hardware, and new verification tools and languages offer enhancements to the conventional flow, but they do not on their own improve verification IP reuse.

Due to its architectural stability, a platform architecture is an excellent vehicle for the provision of an extensible SoC validation environment. Today, platform verification and validation environments are being supplied by ARM that explicitly support coverage-monitoring and test-bench
synchronisation on a complex SoC. In this section we present a validation flow that directly addresses these problems. This flow has been industrially proven with the release of the ARM PrimeXsys Platform in Q1-2002 [1]. The deliverables in this ARM platform product are described in Section 8.

7.2 Extensible verification and validation

The section below describes the technical structure for support of a verification and validation (V&V) methodology that specifically emphasizes reuse, extendibility, and configurability [17,18]. The section following this describes the process for platform V&V using this structure. Verification in this context refers to functional verification. Validation in this context refers to checking the implementation against the original design intent.

7.2.1 XVCs: reusable V&V components

This methodology is founded on the definition and use of ‘extensible verification components’ (XVCs), which are reusable components that can represent either a SoC component, or an external device interfacing to a SoC component. An XVC is typically implemented in an HVL (high-level verification language, such as ‘e’, Vera or the SystemC verification library), and can be used to create stimulus or monitor activity, and contains device-specific/designer knowledge. XVCs contain a library of test operations from simple I/O to executing stimulus files; these are called ‘actions’.

Although current implementations of actions contain pseudo-random and/or directed test stimulus, and temporal checks for protocol, etc., future implementations may also contain assertion-based verification concepts that are used to capture designer knowledge about a device under verification (DUV).

An external test scenario manager (XTSM) is used in a 1:n connection to XVCs to schedule and synchronize selected lists of actions. The XTSM is programmed via a simple input file, which contains a group of XVC action sequence calls. This input file is designed to represent a typical system usage scenario. Each XVC connects to the XTSM via a two-way messaging API, and new XVCs are added to the test environment simply by registering with the XTSM at runtime.
The external test interface (XTI) receives command actions from the XTSM. The XTI communicates with the action generator interface (AGI) and synchronizes the execution of action sequences. An action can be any number of test sequences to be applied to the DUV. The user can add any number of test actions to the AGI and this process gives the XVC its extensibility. Finally the AGI sends transaction requests to the peripheral facing interface (PFI) that in turn converts the transactions into signal driving and sample operations. Sampled data flows back through the PFI into the AGI, which in turn communicates status information back to the XTSM via its XTI. The PFI can be viewed as a hardware abstraction layer for the XVC.

The XVC architecture explicitly supports multi-layer V&V consistencies. The same test-suite may be used at the hardware-dependent software stage with, for example, a SystemC transfer-level model of the platform, and at the hardware integration level with, for example, Verilog IP. If the transactions are carefully authored, this can be extended to the software-development layer. This multi-layer approach is enabled by using a transaction-based encapsulation of actions, and using the PFI to do the abstraction translation.
7.2.2 Coverage monitoring

One of the uses of the XTSM is to also co-ordinate functional coverage data collection and score-boarding. Having a central store for coverage data also brings the advantage that this data can be read ‘on-the-fly’ by individual XVCs. This important feature allows some degree of automated testing in that random stimulus data can be generated and constrained using feedback from the database managed by the XTSM. Techniques using this or similar approaches are often termed as ‘directed random testing’. Systems can be driven until all coverage goals are met, or report if a particular run did not hit specific functional coverage goals.

7.2.3 System-level platform verification

Performing system-level verification should ensure that the individual blocks have been connected correctly and that they perform as intended, in context. Three testbenches are used as part of the system-level verification solution. These testbenches are optimized for:

- **Hardware Integration (Platform Layer 1):** Checking of hardware
  - DMA, IRQ and bus interconnect;
- **Software/hardware Integration (Platform Layer 2):** Checking that software can boot and initialize the hardware through driver routines;
- **System Validation (Platform Layer 2):** Checking the performance and latency within the system, along with corner case scenarios.

Each of these stages of SoC platform verification is explained below. A platform environment for testing of the application software, Platform Layer 3, is provided separate from this SoC verification flow. Software test for Platform Layer 3 is enabled through provision of hard and soft prototyping environments for the platform (see Section 6) that can be targeted through a common debugging front-end.

7.2.3.1 Hardware integration testbench

The primary objective of this testbench is to verify that the hardware blocks that comprise the system have been connected together correctly. This testbench is not designed to collect functional coverage or to verify software driver integration.

The system integration testbench enables rapid verification and debug of the block interconnectivity. A typical scenario would be to identify incorrectly connected control and/or address signals, IRQ or DMA request lines. A CPU model is deliberately omitted from this hardware integration testbench. This ensures the fastest possible turnaround, since no software is being run on the CPU. In place of test software, instructions are injected...
from a simple text file into the appropriate CPU 'socket' within the design. A file-reader bus master model (FRBM) is used to emulate CPU bus reads/writes. The instruction file itself is automatically generated from a set of easily modifiable configuration templates that describe the locations and capabilities of the system peripherals. Combining this with a memory-map configuration file to control the number of tests generated provides an instruction test sequence that reflects the system connectivity. Note that no XVCs are required in this testbench environment, as simple RTL blocks are used for 'loop-back' connections on any external device ports.

7.2.3.2 Software/hardware integration testbench

The primary objective of this testbench is to verify that software can successfully boot and interact with the system. This testbench is not designed to collect functional coverage, or to verify block interconnect at signal level. The architecture of this testbench is shown in Figure 9-9.

Primarily for performance reasons, a C-based Design Sign-Off Model (DSM) of the target CPU is used. As the name suggests, this is a detailed model of the processor, capable of running software to meet the sign-off requirements of the design. Typical usage of this testbench would be for booting the system, configuring and accessing the memory sub-system, running a device driver through the CPU to exercise system peripherals, and stimulating data block transfers through the system.

A user extendible C++ test manager package supports the software test code environment. This manager allows the verification engineer to focus on developing effective test sequences, without the overhead of integrating the tests together. A library of generic (non-OS specific) peripheral device drivers forms part of the standard test suite.

![Figure 9-9. Software/hardware integration testbench using a CPU model and XVCs](image-url)
7.2.3.3 System validation testbench

The primary objective of this testbench is to allow efficient generation and monitoring of complex hardware block interactions or corner-cases, and at the same time measure functional coverage and system performance. This testbench is not designed to verify software driver integration or to verify block interconnect at signal level.

Typical scenarios might include the bus master programming a peripheral, external stimulus arriving at that peripheral, and another peripheral simultaneously generating an interrupt request to be serviced. In this testbench the CPU DSM is not used.

To ensure that the system validation process attains good coverage, many scenarios may be sequenced, potentially creating a significant simulation burden. Removing the need for the CPU DSM improves the efficiency of the testbench. In its place an XVC Bus Master is used, which combines the ability to read from files, and drive the bus with a wider programming capability. For convenience, this master can also read and drive stimulus used by the FRBM from the Integration Testbench.

In contrast to the CPU in the software/hardware integration testbench, an XTSM is used to drive the testing and to provide sequencing and scheduling of the XVCs — for example, to sequence actions for a Master XVC and for it to program system blocks to output data. Corresponding XVCs respond to output data using actions that are also specified by the XTSM. This environment allows the creation of complex test sequences to be abstracted to a very high level. This testbench environment is easy to control and extend, and the verification team can customize an off-the-shelf XVC providing for reuse of the entire validation environment. The adaptation of the testbench architecture to include a test-scenario manager and mastering XVC as replacement to a core model is shown in Figure 9-10.
8. WORKED EXAMPLE: PRIMEXSYS PLATFORM

ARM is developing and delivering the PrimeXsys Platform (PXP) Product-family of systems IP components. Each member of this product family adheres to the principles outlined in the preceding sections. In addition to the hardware and software IP-bundle that has been optimized and validated to support the demanding requirements of consumer OS and embedded RTOS, the PXP products each contain a range of deliverables to improve the time-to-integration of a customer’s differentiated SoC design. This section describes the ARM1136JF-S™ PrimeXsys Platform product that became available to customers in the first quarter of 2003. An earlier generation of this platform, the ARM926EJ-S™ PrimeXsys Platform, has already been shipping to customers as of the first quarter of 2002.

8.1 Establishing a PrimeXsys Community

PrimeXsys Platform products are founded on support of the following guiding principles: (i) a fully validated integration of a software and hardware subsystem with ported OS; (ii) a well-defined set of standards-based extension ports to allow easy integration of differentiating IP blocks on to the base platform; and (iii) an extensive third-party program consisting of IP components and supporting EDA tool infrastructure.

The final of these three points, the business environment for platform support, must be emphasized. A platform provider must ensure that
standards for extensibility are backed by a widespread adoption program to ensure that a broad base of 3rd-party IP is available for integration, both for hardware as well as for software and OS support. ARM has established this with the formation of the PrimeXsys Community Program. The PrimsXsys Community Program is founded on three types of corporate partnership: (i) component providers of enabling IP (hardware and software) conforming to the PrimeXsys Platform extendibility standards as introduced in Section 3, such as the AMBA 2.0 compliant peripherals of the Synopsys® DesignWare™ library and the OS partners who include the providers of: Windows CE, .NET, Symbian OS 7.0 and VxWorks; (ii) EDA and software-development tool partners who ensure that design environments explicitly support the platform standards, such as AMBA-compliant support of SoC-generation in the CoWare® N2C Design™ System tool, verification in the Verisity® Specman Elite™ environment, and support of emulation with the Aptix® System Explorer™; and (iii) software developers who create application packages for driving the platform through a hardware-abstraction layer provided by the operating system port. This broad technical community that is coordinated around the PrimeXsys extensibility standards is the key business driver for adoption of ARM platforms.

8.2 Architecture of the ARM1136 PXP

The ARM1136 PXP is based around ARMs latest generation ARM1 synthesizable processor, the ARM1136, designed to support demanding next generation applications. The processor supports the latest v6 instruction set, which is targeted to improve OS and media application performance. The ARM1136 processor features a complete level-1 memory system including a memory-management unit coupled with an advanced cache architecture. To address the increasingly demanding debugging requirements of advanced real time software, the ARM1136 PXP integrates an embedded trace macrocell and buffer to complement the integrated ICE feature of the core. The fast interrupt response of the processor is supported through the integration of a vectored interrupt-controller in the platform. More details on the ARM1136 can be found in [19].
Figure 9-11. Architecture of the extendible ARM1136 PXP platform

The Platform architecture, illustrated in Figure 9-11, is based around a multi-layer AMBA bus matrix. This multi-layer structure uses nine full multi-access buses (MAB) bus segments and connects the CPU, peripherals and platform extension ports in a flexible manner to provide maximum concurrent bandwidth in support of the most demanding applications. To ensure smooth passage of information around the platform, the processor's
own instruction, data and DMA channels are supported by a multi-channel DMA block, an advanced multi-port memory controller (MPMC) and static interface. Between these components multiple simultaneous data transactions are supported between platform components, bulk and static memory.

The bus matrix features several extension ports, which are described in more detail in Section 8.3. These allow additional peripherals or subsystems to be connected directly to one of the platform’s bus segments, including the processor local busses. In addition support is provided to allow an external master to access all of the platform peripherals including the memory controller. This facilitates the implementation of unified-memory interface SoC systems.

The platform architecture is completed by two sets of peripherals that enable validated standard OS ports to build efficiently. The first set of peripherals comprises a real-time clock, watchdogs and timers, UART and GPIO. This set provides the basic hardware support required to boot and communicate with an operating system. The second set comprises a LCD controller, synchronous serial port and smart card interface. This set provides complete support for a GUI based OS platform. These peripheral sets provide the platform with a defined set of capabilities and therefore define a common Hardware Dependent Software (HdS) abstraction for the operating systems port. This is an important concept in the delivery of a configurable and extendible platform design since changes to the HdS layer would necessitate re-porting of the OS.\(^1\)

The platform architecture also supports address re-mapping to dynamically re-locate the physical location of memory. This supports boot from static memory and subsequent reuse of the low memory area for interrupt vector code.

### 8.3 Support of hardware integration

The bus fabric features a number of extension ports to which additional hardware blocks or subsystems can be connected. This is a key example of how a standards-based framework enables easier integration. Each extension port of the platform conforms fully to the AMBA standard, so any peripheral-block compliant with this standard can be ‘plugged’ into the extension ports with a high degree of confidence. We provide the AMBA Compliance Testbench (ACT), (see Section 4.2), to verify that a peripheral-

\(^1\) An active area of research within ARM is the pursuit of a standard way of expressing an extendible HdS platform abstraction for memory controllers. This will allow new memory variants to be quickly supported without affecting the platform configuration.
block interface has been tested for all possible AMBA bus transactions or events. The ACT provides AMBA compliance certification that a platform integrator may take as an unambiguous contract of integration quality when engaging with a 3rd-party IP provider.

Verification of 3rd-party IP blocks is also supported through the ARM Integrator product. This is an extensible FPGA platform that enables rapid prototyping of peripherals. The platform is provided as a core module into the Integrator environment that can then be extended to prototype the functionality of the configured SoC.

8.4 Support for software development

Software development is supported by two extendible views of the platform: the Integrator FPGA product, and an ARMulator platform soft-prototype.

Into the Integrator product, the ARM1136 PXP is delivered with a board support package. This includes the integrator configuration itself and device drivers for the major ported operating systems. The device drivers supporting the ARM1136 PXP Hds abstraction are provided for each operating system: Windows CE™, .NET™, Symbian OS™ 7.0 and VxWorks™. Whilst a common driver layer across all operating systems would be desirable, substantive differences between the major kernels offered in the embedded-systems market appear to make this proposal infeasible.

Application software developers can utilize the drivers together with an OS image to run on the ARMulator based Software Development Model (SDM). This provides the basis to port application code and to carry out system performance analysis. The ARMulator model can be extended to provide early functional prototyping of features that will represent the extension and configuration of the platform hardware during a complete SoC integration. The other major use of the ARMulator is to provide an early access model for operating system porting. Since the model is fully functionally consistent with the RTL it greatly accelerates the task of porting a new or derivative OS to the platform.

8.5 Support of system verification and validation

A major issue facing integrators of complex systems IP such as the ARM1136 PXP is that of verifying the final system integration. The ARM1136 PXP product meets this challenge in three ways: (i) by carrying out comprehensive subsystem verification that ensures that the platform itself has been validated under a wide variety of expected corner case
scenarios; (ii) by delivering an extendible integration testbench based on the PMAP script outlined in Section 4.3; and (iii) by providing for reuse of the subsystem validation environment at the SoC integration level.

The cornerstone of the system verification and validation support is the XVC standard for verification component extensibility and reuse, as described in Section 7. The XVC methodology specifically supports the first and third verification goals directly, and supports the second by providing test-replay through FRBMs of the PMAP output. Each platform in the PXP family is delivered with a full set of PMAP scripts for the platform. Furthermore, it is provided with a comprehensive set of XVC components that allow the systems integrator to run out-of-the-box test scenarios that ARM engineers used to validate the platform. These verification components are coordinated by an XTSM component that is provided as part of the platform environment.

A conceptual representation of the validation environment provided with a PXP product is shown in Figure 9-12. The capabilities of this environment can be directly used by the systems integrator to run new scenarios, based on the existing supported set of actions of the environment. The PMAP and XVC IP provided with the platform may be re-run by the systems integrator with either a changed or extended address map to verify the integration of any new peripheral.

Figure 9-12. Extendible PrimeXsys Platform Verification and Validation Environment

Extension to the XVC verification environment may be done in two ways: firstly through extending the set of stimulus and monitoring conditions
within the existing set of platform verification components, and secondly through the addition of new verification components.

In the first case of scenario-extension, consider the Smart Card XVC shipped with the ARM1136 PXP. The basic action set supported by this verification component consists of the ability to respond and set the smart card interface lines. Suppose the integrator wishes to verify a system scenario involving rapid insertions and removal of the smart card (such as might occur when a child ‘plays’ with the end use system). This can be captured using a new action that toggles the ‘card present’ line. This action is now available for inclusion in a new scenario. Note that the XVC has been modularly extended, rather than being subjected to unguided modification. This allows for improved support of the verification environment since it is easier to isolate bugs to delivered or extended code.

In the second case of verification environment extension, consider the addition of a new peripheral to the ARM1136 PXP platform. For example this might be the additional of a USB XVC to correspond to the integration of a 3rd-party USB block. By creating a USB XVC that conforms to the standard verification component interfaces of Section 7, this new XVC can integrate with the XTSM and the existing PrimeXsys Platform verification components. This extended verification environment can then be used to create new scenarios that are based on the action capabilities of both the delivered (and possibly extended XVCs) and the new XVCs written specifically for the integration verification environment.

9. SUMMARY

In this chapter, ARM’s lead technical and product marketing minds have provided a broad description of the principles that need to be supported for effective platform based design. These principles have been illustrated with our experience with development and deployment of our PrimeXsys Platform products. As a star-IP provider, ARM is in a unique position to understand the common issues of IP integration that cross the industry. We feel we have come to comprehend the platform-based design problem well, and are providing industry-leading solutions into this space.

This chapter sets the context of platform use models by defining the roles of IP Creator, IP Licensee and IP User. We have defined the set of platform-layers and associated use-models for application software development, middleware and systems micro-architecture development, and hardware integration. We have expressed how both the IP exchange in support of a platform and internal design-refinement flows are enabled by: modeling and
prototyping views of the platform, standard interface development and compliance, and cross-hierarchy verification and validation support.

ACKNOWLEDGEMENTS

The authors would like to thank the many engineers, managers and product support personnel involved in the ARM PrimeXsys Platforms development. In particular, we thank Jonathan Morris and Pete Aldworth for their review of the chapter, and their critical input into defining the ARM platform products.

REFERENCES

