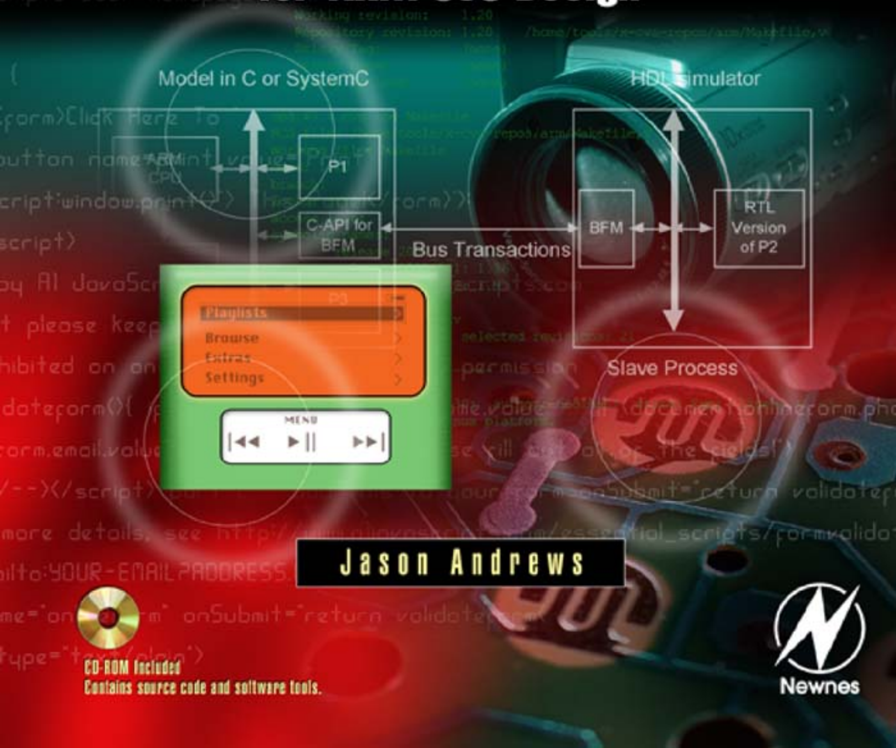
EMBEDDED TECHNOLOGY™
SERIES

Co-Verification of Hardware And Software for ARM SoC Design



Jason Andrews

CD-ROM Included
Contains source code and software tools.

Newnes

Co-Verification of Hardware and Software for ARM SoC Design

This page intentionally left blank

Co-Verification of Hardware and Software for ARM SoC Design

by Jason R. Andrews

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes

Newnes is an imprint of Elsevier
200 Wheeler Road, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Library of Congress Cataloging-in-Publication Data

Andrews, Jason R.
Co-verification of hardware and software for ARM SoC design / Jason R. Andrews
p. cm.
ISBN 0-7506-7730-9
1. Integrated circuits--Vertification. 2. Computer software--Verification. 3. Systems on a chip. I. Title.

TK7874.A595 2004
005.1'4--dc22 2004053860

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

For information on all Newnes publications
visit our website at www.newnespress.com

04 05 06 07 08 09 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

Contents

This page intentionally left blank

Foreword

This is a remarkable book.

Jason Andrews knows about the hardware and the software. He knows about the people, the tools, and the methodologies in the middle ground between hardware and software.

He can also write, explaining complex things so that you can really understand them.

One of the main reasons this middle area is so complex is there are just too many interacting issues to understand and too many decisions to make.

Jason takes care to enumerate the issues, explain how they interact, and describe the options for dealing with them.

Best of all, he explains which tools and methodologies are applicable for each situation. This is crucial because there are many distinct solutions for the problem, and you cannot possibly use them all. You need to make an informed judgment on what to do when.

Jason has either used or implemented most of these solutions, some of them twice, and he gives a very informed tour of the land and guides you through the possible compromises.

Please note that while Jason and I work for a verification company (Verisity) that would love to sell you verification solutions, this book is decidedly generic. It tells you what works, what does not, and why.

While the title of the book is *Co-Verification of Hardware and Software for ARM SoC Design*, I think this book has wider applicability. In fact, if any of the following apply, then you should begin by reading this book:

- You are involved in the verification of products that contain both hardware and software, regardless of whether they are SoC-based or ARM-based.
- You are working on one side of the HW/SW divide, and want to see what the other side looks like.
- You are interested in creating tools for this area.

Yoav Hollander
Founder and CTO, Verisity Inc.
July 2004

Preface

Why Is This Book Important?

This book is the first to document and teach important information about the verification technique known as hardware/software co-verification. Traditional embedded system design has evolved into single chip designs that are pushing past 1M logic gates and headed toward 10M gates. In this era of SoC design, chips now include microprocessors and require software to be developed before hardware fabrication. To develop quality products effectively and in a timely manner, engineers must be armed with necessary information to make educated decisions about which tools and methodology to deploy. SoC verification requires a mix of expertise from the disciplines of microprocessor and computer architecture, logic design and simulation, and C and assembly language embedded software. Individual books exist in each area, but until now the relevant information and how it all fits together has not been available in a single volume. This book provides unique, in-depth information about how co-verification really works, how to be successful using it, and the pitfalls to avoid.

This book also contains an added bonus. It covers important information about developing and verifying SoC designs using ARM microprocessor cores. In the last few years ARM has achieved a dominant market position in the 32-bit embedded microprocessor space and has become the de facto standard for many market segments. This book illustrates the concepts of hardware/software co-verification using concrete ARM SoC examples and provides useful information about co-verification of designs utilizing an ARM microprocessor.

Audience

The primary audience is the engineer looking to develop best practice techniques for SoC verification of both hardware and software not only to increase confidence in the design, but also to complete verification in a shorter period of time. Both hardware and software engineers will benefit from a better understanding of each discipline. Project managers will also benefit from an understanding of the interaction between hardware and software teams and how to encourage collaboration between the two teams. Engineers involved in ARM SoC design projects will also benefit from the information in the book.

Prerequisite Knowledge

Readers should have some knowledge of embedded system design including systems with microprocessors and software. Readers with a hardware engineering background should be familiar with digital logic design and verification. A working knowledge of Verilog or VHDL is useful as well as familiarity with common simulation tools. Readers with a software background should be proficient in C and assembly language programming and should be familiar with embedded system concepts. Verilog is used to present concepts and examples, but everything applies equally to VHDL.

About Hardware/Software Co-Verification

Hardware/software co-verification is about making sure embedded system software works well with hardware before chips and boards are available. It's also about making sure hardware has been designed correctly to run the software successfully. For applications where time-to-market and project cost are important, co-verification saves time and reduces the risk of costly hardware design errors.

Acknowledgments

Thanks to everybody at Axis Systems and Verisity for encouraging me to write this book.

Thanks to Rich Davenport for hiring me into the world of co-verification when I responded to the job ad in the St. Paul Pioneer Press without knowing where it would lead me.

Thanks to David Burns who volunteered to review the manuscript (just for the fun of it) and provided valuable feedback.

Thanks to Yoav Hollander for his review comments and for writing the foreword.

Thanks to Russ Klein for the story of Seamless and making sure I got all the facts straight.

Thanks to all the EDA companies I have worked at, all the great people I have worked with, and to all the users that enabled me to learn so much about this interesting area of verification.

Most of all, big thanks goes out to my wife Deborah and children Hannah, Caroline, Philip, and Charlotte for putting up with all my travel as a remote worker in Minnesota and for their sacrifices in helping me finish this book.

This page intentionally left blank

About the Author

Jason Andrews is currently working in the areas of hardware/software co-verification and testbench methodology for SoC design at Verisity. He has implemented multiple commercial co-verification tools as well as many custom co-verification solutions. His experience in the EDA and embedded marketplace includes software development and product management at Verisity, Axis Systems, Simpod, Summit Design, and Simulation Technologies. He has presented technical papers and tutorials at the Embedded Systems Conference, Communication Design Conference and IP/SoC and written numerous articles related to HW/SW co-verification and design verification. He has a B.S. in electrical engineering from The Citadel, Charleston, SC, and an M.S. in electrical engineering from the University of Minnesota. He currently lives in the Minneapolis area with his wife, Deborah, and their four children.

This page intentionally left *blank*

About Verisity

Verisity Ltd. (NASDAQ: VRST) is the leading provider of verification process automation (VPA) solutions that automate and simplify the complete verification process to increase productivity, predictability and quality. Verisity addresses critical business issues with its verification systems and intellectual property (IP) that effectively verify the design of electronic systems and complex integrated circuits for the communications, computing and consumer electronics markets. Verisity's VPA solutions enable projects to move from an executable verification plan to unit, chip, system and project level 'total coverage' and verification closure. Verisity is a global organization with offices throughout Asia, Europe, and North America. For more information, visit www.verisity.com and also look for the product summaries in the Afterward at the end of this book.

This page intentionally left blank

What's on the CD-ROM?

Included on the accompanying CD-ROM:

- A fully searchable eBook version of the text in Adobe PDF format
- Source code for Figure 6-1
- Verisity product literature

This page intentionally left blank

CHAPTER 1

Embedded System Verification: An Introduction

It works! These are two of the most gratifying words engineers may ever hear. A co-worker once told me that engineering (especially hardware engineering) consists of extended periods of boredom followed by a few brief moments of excitement that result in either bitter disappointment or great satisfaction. The ability to define, architect, design, integrate, verify, test, and deliver a working product provides the drive for continued innovation in the electronics industry.

Recently, I worked on a project that required me to do FPGA design for an ARM CPU board to be used as a microprocessor model for in-circuit emulation of ARM SoC designs (don't worry if you don't understand this yet, you will by the end of the book). After working in the simulation world for some time, it was exciting for me to take a shot at a real hardware design project, even if it was only one programmable device. I diligently created the necessary VHDL source files for the FPGA and constructed a mixed-language Verilog and VHDL simulation environment of the CPU board including a Verilog model for the ARM CPU and my VHDL code for the FPGA and another CPLD on the board. I connected my simulated board to a couple of test designs to make sure the whole thing worked together. After fixing a couple of bugs in the FPGA code, I found the necessary synthesis and FPGA place-and-route tools to turn the VHDL code into a suitable bitstream file for programming the FPGA. After checking and rechecking, the moment of truth had come. It was time to try out the design in the lab. The result would be either disappointment or tremendous satisfaction. As luck would have it, the design worked on the first try. I was successfully able to use the CPU board for in-circuit emulation of ARM designs.

Contrast this experience to one of my previous jobs at a startup that set out to build a complex multiprocessor server of up to 32 Intel processors completely out of small programmable logic devices (PLDs). At that time, all programmable logic devices were small (by today's standards), and the smaller they were, the faster they went. The engineer's rally cry was "plug and debug." The management's mindset was that since the system was constructed totally out of programmable devices it was most expedient to do a bit of design, build a couple boards, and hit the lab. If it did not work on the first try, engineers could simply change the programmable logic and try again. The recipe included iteration until the system finally worked. The only thing standing in the way of a shipping product was one last tweak of a PLD. Then again, considering the primitive logic analyzers being used and the nearly infinite combinations for programming a PLD, the system might never work. Even if this was the way to get a product working in the shortest time, it was difficult on morale. Now I know why they had a spinning siren in the lab that was activated whenever something worked, for it was a bit of a miracle every time it happened. By the way, the system did work eventually (most of the time), though the high part count made reliability a constant problem.

Most engineering projects probably fall somewhere inbetween these two extremes; not everything works on the first try, but it does have a very high chance of working, eventually. The purpose of this book is to increase the experiences of great satisfaction and minimize the disappointment. The best way to increase the odds of success in the embedded system world is to verify that the hardware and software work well together even before the project hits the lab. This book will demonstrate how planning and patience, combined with proven engineering techniques, can help ensure your next embedded system project is a success.

This chapter provides the necessary embedded system background to form a foundation to discuss time and money-saving co-verification techniques.

What's an Embedded System?

There is no formal definition of an ***embedded system***, but it is generally accepted to be dedicated computer hardware with software designed to solve a specific problem or task. This is in contrast to a general-purpose computer, such as a personal computer (PC) or workstation, designed to run any software application programmers create and users choose to install. Marketing campaigns like "Intel Inside" have taught even non-technical people that there is a microprocessor inside every PC. In contrast, embedded systems utilize "hidden" microprocessors. Product literature may not even list how many or what kind of microprocessors are used in a product. As one who is always curious about what is inside a particular product or chip, I am often puzzled as to why this information is not readily available in product brochures and datasheets. This is especially true for products where the software content can be added or changed. As confirmation, ask yourself, "What kind of microprocessor is used in my mobile phone?" Few people have any idea what is inside the phone.

Embedded systems typically use a microprocessor, combined with other hardware and software, to solve a specific computing problem. Microprocessors range from simple (by today's standards) 8-bit microcontrollers to the world's fastest and most sophisticated 64-bit microprocessors. At a minimum some random access memory (RAM) or read-only memory (ROM) is required to store the software. Flash memory is commonly used as nonvolatile memory to hold the software and still allow for field upgrades when defects are fixed or other software enhancements are made. In addition to the microprocessor and memory, embedded systems generally have a mix of hardware functions such as timers, interrupt controllers, UARTs, general-purpose input and output (GPIO) pins, direct memory access (DMA) controllers, real time clocks, and liquid crystal display (LCD) controllers. The mix of hardware peripherals varies greatly in embedded systems and is tailored specifically for the requirements of the product.

Embedded system software can be divided into two main classes, the operating software and the application software. Operating software ranges from a small executive to a large real-time operating system (RTOS). The function of the operating software is to provide a set of services to the application software without forcing the application software to learn about the details of the hardware implementation. The application software implements the specific tasks the embedded system is designed to perform. An example of application software is the graphical user interface (GUI) that is used to configure the product. The different types of embedded system software will be further classified in the next chapter. Figure 1-1 illustrates the basics of an embedded system.

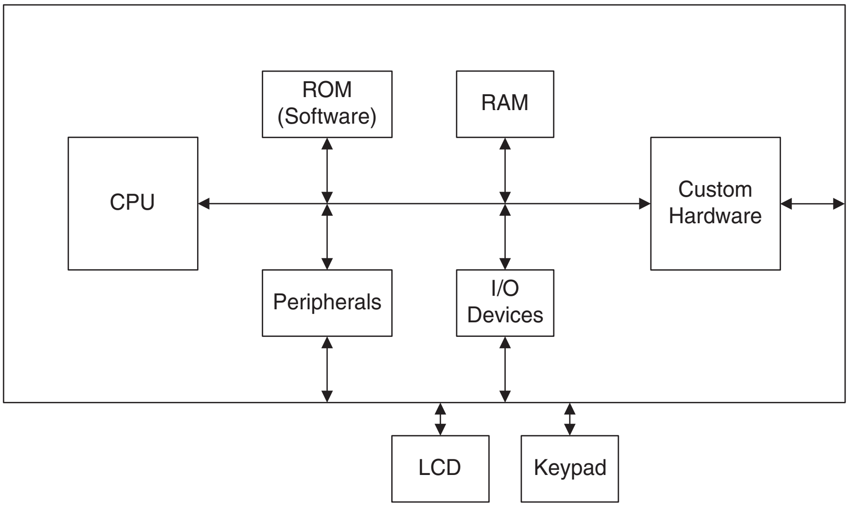


Figure 1-1: Basic diagram of an embedded system

Embedded Systems Are Everywhere

The embedded system landscape is as diverse as the world's population; no two systems are the same. Embedded systems range from large computers, such as air traffic control systems, to small computers, such as a handheld computer that fits into your pocket. Following are just a few of many products we experience each day.

Consumer Electronics

Products include PDAs, MP3 players, DVD players, digital cameras and set-top boxes.

Wireless

Products include mobile phones, base stations and wireless networking products.

Medical

Products include implantable pacemakers and magnetic resonance imaging (MRI) machines.

Networking

Products include routers, switches and gateways.

Security

Products include encryption processors and biometric identification systems.

Imaging

Products include printers, scanners and fax machines.

Storage

Products include disk drives, tape drives and memory cards.

Automotive

Products include engine control systems, anti-lock brakes and navigation systems.

Connectivity and integration are the current focus of many products. No longer are networking and wireless a class of products by themselves, but instead they are features of almost every embedded system. The convergence of products such as the PDA, mobile phone, and MP3 player is occurring. We are quickly approaching the point where everything is connected and there are more microprocessors than people. Try to identify all of the microprocessors that are embedded in the products you use everyday. Your car is likely to have the most microprocessors of anything you own. For example, I recently read an article indicating that the current 7-Series BMW and S-Class Mercedes each have about one-hundred processors. Even a basic non-luxury car has about twenty-five processors.

Design Constraints

Embedded system designs have different constraints depending on the target application.

Cost

Expected product volume directly impacts cost constraints. Low volume products with high margins can afford trade-offs that may increase the manufacturing cost in exchange for useful benefits such as flexibility or the ability to update the product at a later time. High volume products are more restrictive since saving a small amount of money per unit leads to a major cost savings down the road.

Memory

One area for cost savings in high volume designs is memory size or memory type. Some embedded microprocessors have developed a special mode that allows a 32-bit architecture to run 16-bit instructions. This technique offers the performance of a 32-bit processor and the memory requirements of a 16-bit processor for software storage. Of course, there is some overhead required to process 16 bit instructions in a 32-bit CPU. Examples include the ARM Thumb and MIPS16 instruction sets.

Power

Portable products with batteries require a design that is optimized for low power consumption. To minimize power, both static and dynamic power dissipation must be considered. Static power dissipation (when the system is not active) can be minimized using sleep modes and special circuits with low leakage current. Dynamic power dissipation (when the system is running) can be minimized by reducing operating voltages and controlling the clock frequency. Clock gating techniques save power by shutting off the clock to parts of the design not being used. The system clock may be slowed down or stopped as a way to insert wait states on the microprocessor bus instead of using a wait signal to indicate the insertion of wait states.

Real-Time Response

Response time is a critical constraint in embedded systems. There are specific constraints for how long the system can take to respond to critical events. This is also referred to as interrupt latency. There are two kinds of real-time computing systems, hard real-time and soft real-time. Systems with hard real-time constraints must have a guaranteed response time with potentially tragic results if these requirements are not met. Factory automation is an example of hard real-time system. If the software cannot perform specific tasks following events such as external interrupts, machines shut down or malfunction. Soft real-time systems also have constraints on response time to external events, though the penalty for failing to meet the requirements is less severe. A mobile phone is one example of a soft real-time system. If the phone software cannot respond quickly, the call in progress is lost.

Performance

Performance is one of the most important constraints and one of the most difficult to predict and verify. Ideally, performance can be computed or predicted before the system is constructed. In reality, only after the hardware is designed and the software is run on the hardware can the architects be certain that the design meets the performance requirements. It is crucial to have tools that provide visibility and the ability to quantify expected performance.

System Size

Product size requirements may restrict the possible solutions. A mobile phone that can run any protocol from anywhere in the world and play 3D games at 1 GHz may not succeed because of its excess size and weight. In contrast, customers may perceive more value from a product that is larger. A bigger car must cost more than a smaller car because it takes more material to make it.

Reliability

Embedded systems are viewed differently than general-purpose computers. It has become common for users to accept that general-purpose computers such as desktop PCs often crash and need to be restarted. In critical applications, such as servers, techniques such as ECC (error correcting codes) and RAID (redundant array of inexpensive disks) are used to provide higher reliability for memory and disk storage. Most embedded systems are not allowed to crash. Reliability is accomplished using both hardware and software to detect problems and correct them. A watchdog timer is an example of a software technique used to correct a problem. Embedded systems also have a higher probability of being reliable since the software is tightly controlled and thoroughly tested. Generally, users cannot add or change the embedded system software.

Time-to-Market

Time-to-market is one of the most talked about concepts in embedded system design. Being first to deliver a new and unique product can propel it into becoming the de facto standard and produce higher revenue than if there are two similar products available. Much of the high-tech world of electronics and software revolves around the delivery of unique technology that provides valuable benefits to users. In the world of electronic design automation (EDA) and embedded development tools (EDT), every company is trying to deliver products that help engineers get products to market faster with fewer problems. Related to time-to-market concerns is total development cost. In poor economic times, companies try to maintain cash and stay alive. In such an environment, saving money is more important than time-to-market since consumers may not have the resources to adopt every new emerging technology.

Embedded Systems Decomposition

There are endless topics related to embedded systems and many good books on the subject. The aim of this book is not to cover all of the related topics such as hardware design, writing embedded software, porting operating systems, debugging software, and writing diagnostics to test hardware systems. The goal is to understand the necessary background to help engineers verify that the software works with the hardware and to validate that the system meets the design requirements. To reach this goal, some background on microprocessor history is useful to identify the classes of embedded systems that are the most at risk in the area of hardware/software interaction.

Microprocessors, Chips and Boards

In 1989, Andy Grove announced at the Embedded Systems Conference that by the end of the century Intel would have a microprocessor chip numbered 80786 with 100 million transistors and a 250 MHz clock rate. As it turned out, Intel dropped the 80x86 numbering system so the marketing people could put on a better campaign using terms like Pentium, but Grove's performance prediction was exceeded when Intel produced a chip running at more than 1 GHz. Intel's dominance in PC microprocessors and strong marketing gives them higher name recognition than companies making processors for embedded systems. Major companies providing microprocessors for embedded systems are ARM, Hitachi, IBM, MIPS, Motorola, and Texas Instruments. Engineers can choose from somewhere around 100 different microprocessors for a specific embedded system design project. Shipments of embedded microprocessors greatly outnumber microprocessors used in PCs and workstations.

Microprocessor vendors target design wins in two ways. First, some sell chips for use on a printed circuit board (PCB). This market is referred to as system-on-board (SoB). The microprocessor company is selling chips to companies making boards and systems. The second way microprocessor vendors operate is to sell microprocessors for use in an application specific integrated circuit (ASIC) or application specific standard product (ASSP). This market is referred to as system-on-chip (SoC). The microprocessor vendor is selling to companies making chips and potentially also using those chips in systems. No actual chips are being sold for use in the SoC, but the design files for the microprocessor are being sold. The design files are referred to as intellectual property (IP) and the companies that sell them are labeled "IP Companies" since they sell files, more like selling software.

The term SoC has no formal, agreed upon definition, but SoC has generally come to mean a single chip that includes one or more microprocessors, application specific custom logic functions, and embedded system software.

Some view a SoC as a system where a single chip provides all of the required digital logic, and the existence of a microprocessor is optional. Japanese companies often call this integrated device a System LSI (large scale integration). SoC is always a bit misleading since a product or system always needs more than one single device. External memories and connections to hardware such as display and keyboard are usually required. After all, a chip by itself is not much use unless it is put on a board and connected to something more meaningful. There is also the world of analog design. Even when a single digital chip is used, analog components are not always included in the chip for technology and cost reasons. The world of embedded system design continues to integrate more and more functionality into larger semiconductor devices, and the inclusion of microprocessors presents new challenges for SoC hardware and software engineers.

Vendors selling chips to the SoB market provide value and differentiation in one of two areas, either high-performance or high-integration. High performance chips are the latest and greatest chips with the fastest clock rates. Many embedded systems require maximum processor performance to meet design requirements. Examples of high-performance microprocessors used in embedded systems are the Motorola MPC7455 and the PMC-Sierra RM9000. These are 32- and 64-bit devices capable of clock speeds in excess of 1 GHz.

The second area where chip vendors deliver value and differentiation is by providing less than leading edge performance yet maximizing integration. These devices typically start with a CPU that has already been proven and add to it additional peripherals so that nearly all of the required functionality is available in a single chip. This lowers the total component count and provides a solution requiring less custom hardware to be designed from scratch. Examples of high-integration microprocessors used in embedded systems are the Motorola PowerQUICC family. This family of chips uses proven PowerPC cores combined with nearly every peripheral needed for any networking application.

The recent growth of the SoC market is a result of the natural progression in electronic design driven by increased performance (faster), higher integration (smaller) and lower cost (cheaper) products. Embedded products that started as SoB products built with an off-the-shelf microprocessor chip and other components have evolved. Over time, it became advantageous to integrate the microprocessor and the custom logic into a single device. For example, the Nokia mobile phone started as a set of discrete components for the CPU, DSP, memory and supporting hardware. This was one of the first applications to merge the CPU, DSP and other logic into what we now call an SoC.

Increased integration brings increased risk and decreased flexibility. Integration decisions can be difficult because of the risk involved. The Nokia phone had many choices for discrete CPUs, though choices became limited once the CPU and DSP were integrated into a more complex custom chip. System engineers focus on choosing the best microprocessor and system architecture to satisfy the project requirements within the given constraints.

Embedded System Classifications

Classification of embedded systems may be based on the profile of the hardware design. Hardware solutions that satisfy the design constraints can be obtained in many different ways, ranging from assembling off-the-shelf boards and components to designing full-custom integrated circuits. Following is a rather simplistic categorization of embedded system hardware to help identify the types of systems that are the best candidates for improving collaboration and integration of hardware and software.

Little or No Custom Hardware Design

Some systems are designed using little or no custom hardware. The solution is built using off-the-shelf boards or even complete systems. Appropriate software development solves the problem and satisfies the design requirements without any hardware modifications. Minor modifications may be made to an off-the-shelf board or reference design from a microprocessor vendor to add some custom logic to satisfy the design requirements. The custom hardware often takes the form of programmable logic. Again, software is the primary differentiator of the product. These projects have the following hardware design characteristics:

- Hardware design is minimized (to get the product to market quickly)
- Hardware uses high-integration microprocessors or off-the-shelf boards

An example of this type of design is a piece of hospital equipment I recently saw for monitoring brain waves. The solution used a Windows PC with some custom hardware to connect the PC to a set of probes. Likely, most of the work to produce the product was in the software that was running on the PC to produce useful results for doctors and nurses.

A Lot of Custom Hardware – SoB Design

Board designs that make use of high-performance microprocessors have performance as a top priority. Complex custom logic in the form of FPGAs and ASICs is often required to meet performance objectives. Integration is used primarily as a means to achieve performance requirements. These designs have the following hardware design characteristics:

- High-performance microprocessor, such as PowerPC and MIPS chips are utilized
- Boards include large custom logic, FPGAs or ASICs
- Boards are often very large and may utilize multiple processors
- Systems can be comprised of multiple boards

An example of this type of design is the high-performance routers produced by Cisco. To achieve the required performance, a combination of custom ASICs and the fastest MIPS microprocessor chips are used. Size and power are less of a factor in router design. A large effort for both hardware and software is required to produce a final product.

A Lot of Custom Hardware – SoC Design

Applications that require small product size and low power often turn to SoC design. SoC can also be used to meet performance requirements that could not be achieved by limitations imposed by printed circuit board technology. For the purpose of this section, SoC is defined as an ASIC or ASSP that includes one or more microprocessors on a chip. Microprocessor companies create designs to sell high-integration chips. Numerous other fabless semiconductor companies also participate in SoC design by licensing microprocessor IP to develop and sell chips targeted at specific industries. These designs have the following hardware design characteristics:

- One or more microprocessors on a chip
- Incorporates microprocessor IP such as ARM, MIPS or Tensilica
- DSP cores are often included
- Integration, performance and power are crucial
- Cost to develop is very high
- Contains high custom hardware content

Good examples of this type of design are readily found in consumer electronics such as MP3 players and digital cameras. In consumer electronics the volume is higher and die size and power are important.

Custom hardware requires custom software to provide system specific diagnostics, initialization software, and device drivers. By understanding the nature of the hardware design it becomes easier to identify the risks in a specific embedded system design project and formulate a strategy to increase the chances of project success.

Embedded System Design Process

The process of embedded system design generally starts with a set of requirements for what the product must do and ends with a working product that meets all of the requirements. Following is a list of the steps in the process and a short summary of what happens at each state of the design. The steps are shown in Figure 1-2.

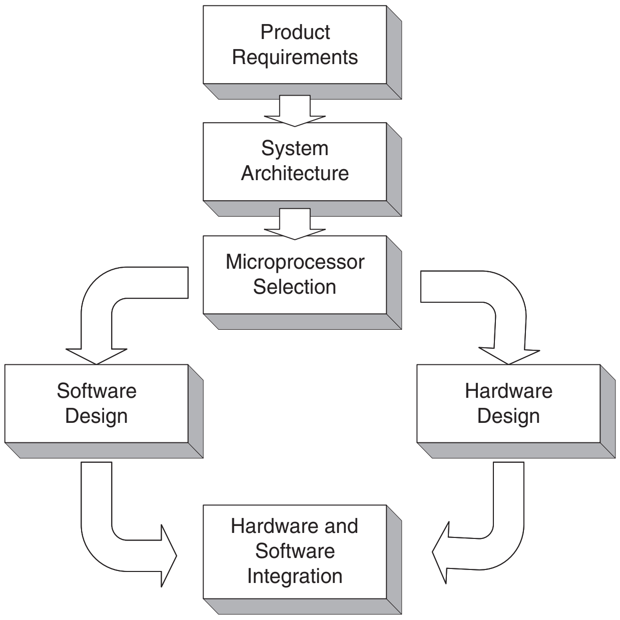


Figure 1-2: Embedded system design process

Requirements

The requirements and product specification phase documents and defines the required features and functionality of the product. Marketing, sales, engineering, or any other individuals who are experts in the field and understand what customers need and will buy to solve a specific problem, can document product requirements. Capturing the correct requirements gets the project off to a good start, minimizes the chances of future product modifications, and ensures there is a market for the product if it is designed and built. Good products solve real needs, have tangible benefits, and are easy to use.

System Architecture

System architecture defines the major blocks and functions of the system. Interfaces, bus structure, hardware functionality and software functionality are determined. System designers use simulation tools, software models, and spreadsheets to determine the architecture that best meets the system requirements. System architects provide answers to questions such as, "How many packets/sec can this router design handle?" or "What is the memory bandwidth required to support two simultaneous MPEG streams?"

Microprocessor Selection

One of the most difficult steps in embedded system design can be the choice of the microprocessor. There are an endless number of ways to compare microprocessors, both technical and non-technical. Important factors include performance, cost, power, software development tools, legacy software, RTOS choices, and available simulation models. Benchmark data is generally available, though apples-to-apples comparisons are often difficult to obtain. Creating a feature matrix is a good way to sift through the data to make comparisons.

Software investment is a major consideration for switching the processor. Embedded guru Jack Ganssle says the rule of thumb is to decide if 70% of the software can be reused; if so, don't change the processor. Most companies will not change processors unless there is something seriously deficient with the current architecture. When in doubt, the best practice is to stick with the current architecture.

Hardware Design

Once the architecture is set and the processor(s) have been selected, the next step is hardware design, component selection, Verilog and VHDL coding, synthesis, timing analysis and physical design of chips and boards.

The hardware design team will generate some important data for the software team such as the CPU address map(s) and the register definitions for all software programmable registers. As we will see, the accuracy of this information is crucial to the success of the entire project.

Software Design

Once the memory map is defined and the hardware registers are documented, work begins to develop many different kinds of software. Examples include boot code to start up the CPU and initialize the system, hardware diagnostics, real-time operating system (RTOS), device drivers and application software.

During this phase, tools for compilation and debugging are selected and coding is done.

Hardware and Software Integration

The most crucial step in embedded system design is the integration of hardware and software. Somewhere during the project the newly coded software meets the newly designed hardware. How and when hardware and software will meet for the first time to resolve bugs should be decided early in the project. There are numerous ways to perform this integration. Doing it sooner is better than later, though it must be done smartly to avoid wasted time debugging good software on broken hardware or debugging good hardware running broken software.

Verification and Validation

Two important concepts of integrating hardware and software are verification and validation. These are the final steps to ensure that a working system meets the design requirements.

Verification: Does It Work?

Embedded system verification refers to the tools and techniques used to verify that a system does not have hardware or software bugs. Software verification aims to execute the software and observe its behavior, while hardware verification involves making sure the hardware performs correctly in response to outside stimuli and the executing software. The oldest form of embedded system verification is to build the system, run the software, and hope for the best. If by chance it does not work, try to do what you can to modify the software and hardware to get the system to work. This practice is called *testing* and it is not as comprehensive as verification. Unfortunately, finding out what is not working while the system is running is not always easy. Controlling and observing the system while it is running may not even be possible. To cope with the difficulties of debugging the embedded system many tools and techniques have been introduced to help engineers get embedded systems working sooner and in a more systematic way. Ideally, all of this verification is done before the hardware is built. The earlier in the process problems are discovered, the easier and cheaper they are to correct. Verification answers the question, "Does the thing we built work?"

Validation: Did We Build the Right Thing?

Embedded system validation refers to the tools and techniques used to validate that the system meets or exceeds the requirements. Validation aims to confirm that the requirements in areas such as functionality, performance and power are satisfied. It answers the question, "Did we build the right thing?" Validation confirms that the architecture is correct and the system is performing optimally.

I once worked with an embedded project that used a common MIPS processor and a real-time operating system (RTOS) for system software. For various reasons it was decided to change the RTOS for the next release of the product. The new RTOS was well suited for the hardware platform and the engineers were able to bring it up without much difficulty. All application tests appeared to function properly and everything looked positive for an on-schedule delivery of the new release. Just before the product was ready to ship, it was discovered that the applications were running about 10 times slower than with the previous RTOS. Suddenly, panic set in and the project schedule was in danger. Software engineers who wrote the application software struggled to figure out why the performance was so much lower since not much had changed in the application code. Hardware engineers tried to study the hardware behavior, but using logic analyzers that are better suited for triggering on errors than providing wide visibility over a long range of time, it was difficult to even decide where to look. The RTOS vendor provided most of the system software and so there was little source code to study. Finally, one of the engineers had a hunch that the cache of the MIPS processor was not being properly enabled. This indeed turned out to be the case and after the problem was corrected, system performance was confirmed. This example demonstrates the importance of validation. Like verification, it is best to do this before the hardware is built. Tools that provide good visibility make validation easier.

Human Interaction

Embedded system design is more than a robotic process of executing steps in an algorithm to define requirements, implement hardware, implement software, and verify that it works. There are numerous human aspects to a project that play an important role in the success or failure of a project.

The first place to look is the organizational structure of the project teams. There are two commonly used structures. Figure 1-3 shows a structure with separate hardware and software teams, whereas Figure 1-4 shows a structure with one group of combined hardware and software engineers that share a common management team.

Separate project teams make sense in markets where time-to-market is less critical. Staggering the project teams so that the software team is always one project behind the hardware team can be used to increase efficiency. This way, the software team always has available hardware before they start any software integration phase. Once the hardware is passed to the software engineers, the hardware engineers can go on to the next project. This structure avoids having the software engineers sitting around waiting for hardware.

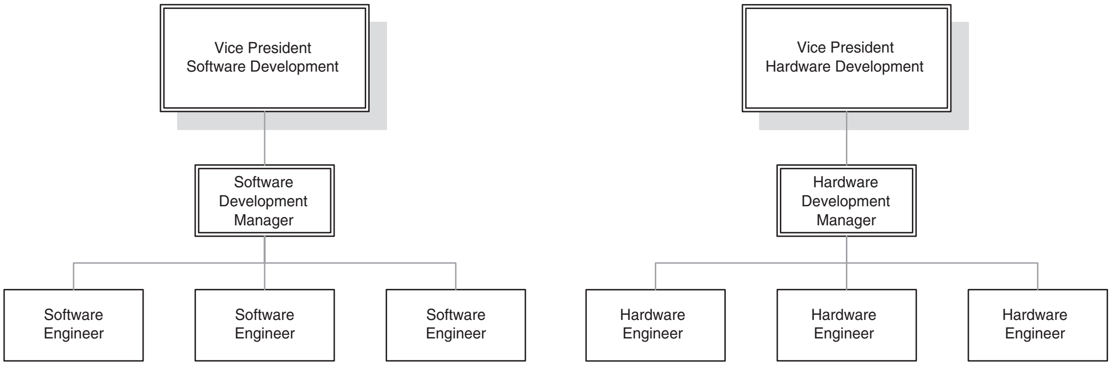


Figure 1-3: Management structure with separate engineering teams

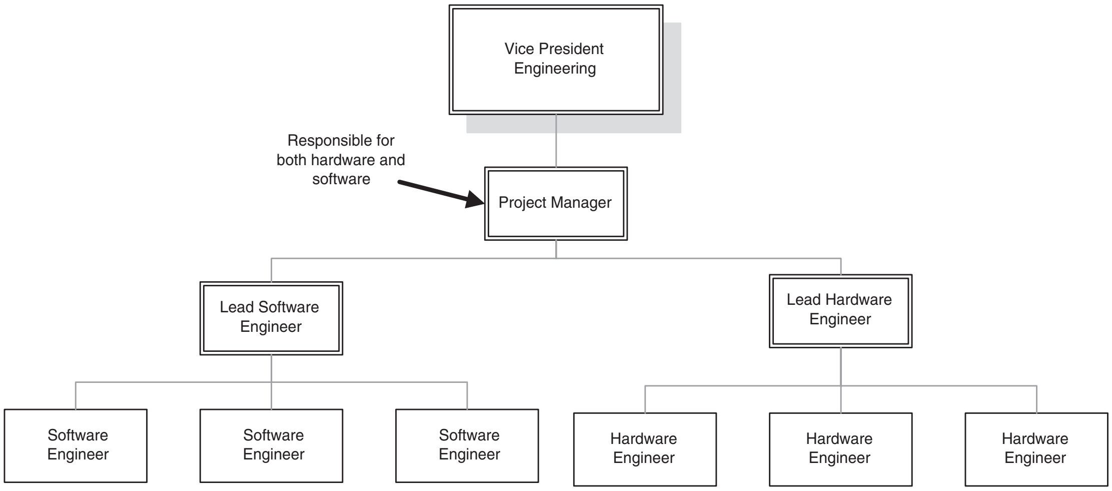


Figure 1-4: Management structure with combined engineering teams

A combined project team is most efficient for addressing time-to-market constraints. The best situation to work under is a common management structure that is responsible for project success, not just one area such as hardware engineers or software engineers. Companies that are running most efficiently have removed structural barriers and work together to get the project done. In the end, the success of the project is based on the entire product working well, not just the hardware or software.

I once worked in a company that totally separated hardware and software engineers. There was no shared management. When the prototypes were delivered and brought up in the lab, the manager of each group would pace back and forth trying to determine what worked and what was broken. What usually ended up happening was that the hardware engineer would tell his manager that there was something wrong with the software just to get the manager to go away. Most engineers prefer to be left alone during these critical project phases. There is nothing worse than a status meeting to report that your design is not working when you could be working to fix the problems instead of explaining them. I don't know what the software team was communicating to its management, but I also envisioned something about the hardware not working or the inability to get time to use the hardware. At the end of the day, the two managers probably went to the CEO to report the other group was still working to fix its bugs.

Everybody has a role to play on the project team. Understanding the roles and skills of each person as well as the personalities makes for a successful project as well as an enjoyable work environment. Engineers like challenging technical work. I have no data to confirm it, but I think more engineers seek new employment because of difficulties with the people they work with or the morale of the group than because they are seeking new technical challenges.

What is this Book About?

This chapter provided an introduction into the interaction of hardware and software for embedded system projects. The purpose of the remaining chapters is to document and teach important information about the verification technique known as Hardware/Software Co-Verification. It is clear that many projects are not completed on time and those that are may be able to shrink schedule and lower cost even more.

A recent survey into embedded systems projects found that more than 50% of designs are not completed on time. Typically those designs are 3 to 4 months off the pace, project cancellations average 11–12% and average time to cancellation is 4-and-a-half months. Jerry Krasner of Electronics Market Forecasters June 2001.

Hardware/software co-verification aims to verify embedded system software executes correctly on a representation of the hardware design. It performs early integration of software with hardware, before any chips or boards are available.

The primary focus of this book is on SoC verification techniques. Although all embedded systems with custom hardware can benefit from co-verification, the area of SoC verification is most important because it involves the most risk and is positioned to reap the most benefit. The ARM architecture is the most common microprocessor used in SoC design and serves as a reference to teach many of the concepts presented in the book.

If any of the following statements are true for you, this book will provide valuable information:

1. You are a software engineer developing code that interacts directly with hardware.
2. You are curious about the relationship between hardware and software.
3. You would like to learn more about debugging hardware and software interaction problems.
4. You desire to learn more about either the hardware or software design processes for SoC projects.
5. You are an application engineer in a company selling co-verification products.
6. You want to get your projects done sooner and be the hero at your company.
7. You are getting tired of the manager bugging you in the lab asking, “Does it work yet?”
8. You are a manager and you are tired of bugging the engineers asking, “Does it work yet?” and would like to pester the engineers in a more meaningful way.

9. You have no clue what this stuff is all about and want to learn something to at least sound intelligent about the topic at your next interview.

Scope and Outline

This book presents practical techniques to verify integration of SoC hardware and software. It provides detailed information and plenty of examples for the most common SoC being designed today, those using ARM microprocessors. The examples most directly relate to the SoC/ASIC/ASSP market where the risk is greatest and mistakes equal money. It is most relevant to software engineers developing code that deals with hardware operation.

The book is not a design book to create a new chip or software; it has no grand and glorious top-down schemes for system-level design and hardware/software partitioning. There are many other books about design practices using Verilog and VHDL and associated simulation and synthesis tools.

Chapter 2 provides information about the separate yet related hardware and software design processes. It documents the tools and techniques commonly used in each discipline. The boundaries where hardware meets software and the relationships at these boundaries are described. The view of the world for both the hardware engineer and the software engineer is presented.

Chapter 3 gives an overview and some detailed information about the ARM architecture used throughout the book. For engineers involved in ARM projects it provides both a tutorial on the architecture and bus protocols as well as details about important areas related to co-verification.

Chapter 4 provides the foundation for hardware/software co-verification including the definitions, techniques, benefits and examples.

Chapter 5 covers advanced topics on hardware/software co-verification that most engineers don't typically learn except from on-the-job training while trying to use co-verification on an embedded system project.

Chapter 6 explains the relationship between co-verification and the testbench. Traditional hardware verification techniques and how they relate to the microprocessor portion of the design are covered.

Chapter 7 puts all of the previous information into practice and presents a methodology for SoC verification using the techniques covered in the early chapters on a real ARM SoC design example.

This page intentionally left blank

CHAPTER 2

Hardware and Software Design Process

To understand hardware/software co-verification, it is necessary to understand the tools and the process used to develop hardware and software. Until recently, the integration of software with hardware was performed in a lab environment by constructing the hardware and running the software. Debugging was done using equipment such as in-circuit emulators (ICE), logic analyzers, and oscilloscopes. Debugging late in the design cycle, when the pressure of the project schedule is the greatest, is a tedious and stressful task. Co-verification changes this environment by using a virtual prototype of the hardware to execute the software well before prototypes are available. With this in mind, let's dig into the technical aspects of how hardware and software engineers work and identify the boundaries where they meet.

Three Components of SoC Verification

Before getting into the details of hardware and software design and examining the intersection of the two, it is worthwhile to review of the three components of SoC verification. To build a cohesive methodology for hardware and software verification, engineers must understand specific tools in each area as well as the interoperability between them. The three components are:

1. Verification platform
2. Hardware verification tools and techniques
3. Software debugging tools and techniques

Verification Platform

The verification platform is the method used to execute a description of the hardware design. It has other common names such as execution engine or virtual prototype. The hardware design process consists of describing the hardware using one of the two common hardware description languages (HDLs), Verilog or VHDL. This HDL representation of the hardware design can be executed using any number of platforms or execution engines.

Four distinct methods have been identified and used for the execution of the hardware design:

- Logic simulation
- Simulation acceleration
- Emulation
- Hardware prototyping

Each hardware execution method has specific debugging techniques associated with it, each with its own set of benefits and limitations. The methods range from the slowest execution method, with the most flexibility and best debugging, to the fastest, with less flexibility and debugging. The following definitions are used to describe the types of platforms and the way they operate.

Hardware Description Language (HDL) refers to a dedicated language designed to describe hardware. The two languages used today are Verilog HDL and VHDL. These languages are used to specify the behavior of a chip or a board in the same way a software program specifies the behavior of a microprocessor or embedded system. HDLs contain the keywords, syntax and semantics to model hardware circuits. Software tools can then use these models to simulate hardware behavior and to synthesize HDL models into structural representations that can be used to build hardware. HDLs are used to specify the implementation of ASIC and FPGA devices. HDLs are also used to model off-the-shelf electronic components such as memories and other digital logic, as well as to model the stimuli to and from the interfaces of the chip or system. Following is a short history of Verilog and VHDL. Numerous books and resources are available for those wishing more information on the uses and details of Verilog and VHDL.

Verilog HDL originated at Automated Integrated Design Systems (later renamed Gateway Design Automation) in 1985. Verilog HDL was designed by Phil Moorby, who later became the Chief Designer for Verilog-XL and the first Corporate Fellow at Cadence Design Systems. Gateway Design Automation grew rapidly with the success of Verilog and was acquired by Cadence Design Systems, San Jose, CA in 1989.

Cadence Design Systems decided to open the Verilog language to the public in 1990. The standards body created to oversee Verilog was called OVI (Open Verilog International). When OVI was formed in 1991, a number of small companies began work on Verilog simulators. The first of these came to market in 1992. The most successful simulator to follow Verilog-XL was VCS, the Verilog Compiled Simulator, from Chronologic Simulation. VCS was a compiler as opposed to an interpreter like Verilog-XL. As a result, compile time was longer, but simulation execution speed was much faster. Today there are Verilog simulators available from several sources including vendors such as Cadence, Mentor Graphics, and Synopsys.

An IEEE (Institute of Electrical and Electronic Engineers) working group was established in 1993 under the Design Automation Sub-Committee (DASC) to produce the IEEE Verilog standard 1364. The IEEE maintains standards for a variety of engineering fields. Verilog became IEEE Standard 1364 in 1995. The latest version of the IEEE 1364 Verilog HDL and PLI (programming language interface) standard is called Verilog 2001. The Verilog PLI provides a programming interface that allows engineers to customize and extend the capabilities of Verilog simulators. Engineers use the PLI for a variety of programming tasks during simulation. There are actually three generations of PLI. The first generation PLI routines (starting with tf_) work only with the parameters passed to them. It was soon apparent that all design parameters could not be passed to C functions as parameters. As a result, more sophisticated PLI functions emerged. Second generation functions are known as access routines (starting with acc_), and cover a variety of design objects while keeping the user interface as simple as possible. Access routines did a good job keeping the user interface simple, but often they are inconsistent. In 1995, Cadence came up with the third generation of PLI for Verilog, the Verilog Procedural Interface (VPI). All three generations of PLI are part of IEEE Standard 1364-1995.

VHDL, the very high-speed integrated circuit (VHSIC) hardware description language, is the second language used by hardware engineers to describe hardware. Unlike Verilog, which was conceived by a private company, VHDL is a product of the VHSIC program funded by the Department of Defense in the 1970s and 1980s. It was intended to serve as a way to document circuits as well as a modeling language for simulation. VHDL was first approved as the IEEE 1076 standard in 1987 and was updated in 1993 to the IEEE 1164 standard. While Verilog tends to look more like C, VHDL tends to look more like Ada, due to its roots in the Department of Defense.

For several years the industry has debated about which HDL is better, Verilog or VHDL. This language war is pretty much over today. Neither language was ever declared the winner. Both are widely used today. Verilog is loosely typed and considered less verbose, although this can lead to unexpected behavior. VHDL is strongly typed so it is more difficult to learn, but some would argue that saves time in the long run. There are no hard rules about who uses Verilog and who uses VHDL, but some generalizations are possible. Verilog is more common for ASIC design and VHDL is more common for FPGA design. It is generally accepted that Verilog is more widely used in North America, VHDL is more widely used in Europe, and Japan is a pretty even mix of Verilog and VHDL.

Both Verilog and VHDL continue to evolve. The next version of Verilog being developed is System Verilog 3.X (the 3rd generation of Verilog) and it will likely result in a new IEEE 1364 standard. VHDL also saw the formation of a working group in 2003 to start work on next version of the IEEE 1076 (language) and 1164 (packages) standards. Both languages are adding more support for testbench features, assertions, better modeling and an expanded synthesis subset.

Since parts of a chip or system may be designed by different groups in a company or purchased from other companies, many designs use a combination of both languages. This situation of using both Verilog and VHDL is called *mixed-language design*, and simulators that can simulate Verilog and VHDL in the same simulation are known as mixed-language simulators. Most logic simulators today are mixed-language capable and some can run additional languages, such as SystemC, beyond Verilog and VHDL.

For the purposes of this book, **Software Simulation** refers to an event-driven logic simulator. Software simulators run on workstations and use Verilog and VHDL as simulation languages to describe the design and the testbench (stimulus and response checking).

The most common type of digital simulator is an event-driven simulator. When the value of a signal changes, the time, the signal and the new value are collectively referred to as an event. The event is scheduled by putting it in an event queue or event list. When the specified time is reached, the logic value of the signal is changed. The change affects other signals that have this signal as an input. All of the affected signals must be evaluated, which may add more events to the event list. The simulator keeps track of the current time, the current time step and the event list that holds future events. For each signal, the simulator keeps track of the logic state and the strength of the source or sources driving the signal. The logic simulator is the most common tool used to simulate the behavior of hardware designs. Logic simulators follow one of two models, interpreted-code or compiled-code.

An interpreted-code simulator uses the HDL model as data, compiling an executable model as part of the simulator structure, and then executes the model. This type of simulator usually has a short compile time but a longer execution time compared to a compiled-code simulator. An example of an interpreted-code simulator is Verilog-XL. A compiled-code simulator converts the HDL model to an intermediate form (usually C) and then uses a separate compiler to create executable binary code (an executable). This results in a longer compile time but shorter execution time than an interpreted-code simulator. Most simulators today are categorized as native-code-compiled since they bypass the intermediate representation (such as C) and convert the HDL directly to an executable for the workstation. Native-code compiled simulators offer the fastest execution time. There are also hybrid simulators that can be configured to run in interpreted mode or compiled mode depending on user preference.

There are many more aspects to learn about the workings of a modern logic simulator, and gurus in this area will debate about the details, but these descriptions will suffice for the purpose of understanding the role of the logic simulator in SoC verification.

Simulation Acceleration refers to the process of mapping the synthesizable portion of the design into a hardware platform specifically designed to increase performance by evaluating HDL constructs in parallel. Verilog and VHDL are inherently parallel languages since they are used to describe hardware operations that involve many concurrent operations. This parallelism is what makes them different from the sequential nature of most software programming languages. Simulation acceleration takes advantage of this parallelism. Since logic simulators runs on a workstation, the parallel constructs end up being serialized into the CPU instruction set of the workstation. Over the years, many people have tried to use multiple processors or multiple workstations to take advantage of the parallelism of Verilog and VHDL simulation, but with little success; however, simulation acceleration using custom hardware instead of general-purpose processors has had good success in providing faster simulation. In simulation acceleration there are two components that contribute to the overall simulation time; the portion of the simulation that can be mapped into the custom hardware, and the remaining portions of the simulation that are not mapped into hardware. The later portions run in a software simulator and work in conjunction with the hardware platform to exchange simulation data. Simulation acceleration provides higher performance because it removes simulation events from the logic simulator and evaluates them using parallel processing hardware. This removal of simulation events increases performance. The easy way to think about acceleration is to say that whatever can be removed from the simulator executes in zero time. The final performance is determined by the percentage of the simulation that is left running in the logic simulator. For example, if 50% of the simulation events can be removed from the logic simulator and evaluated in zero time, then the total simulation time is cut in half. The simulation acceleration setup is shown in Figure 2-1.

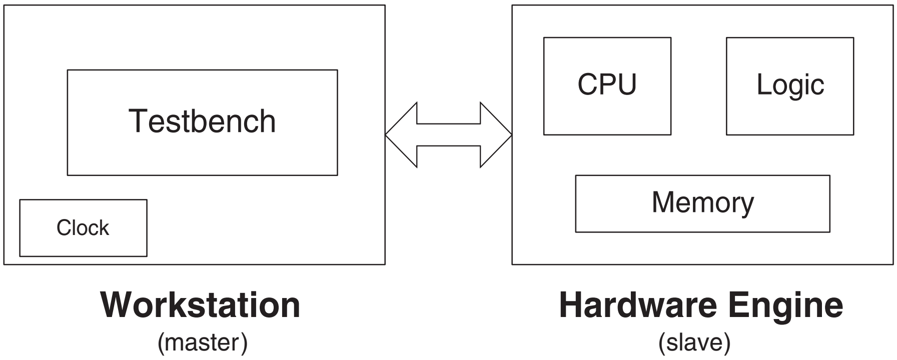


Figure 2-1: Simulation acceleration

The goal of acceleration is to increase performance. The final performance is based on the speed of the acceleration platform and the percentage of the simulation that can be run inside the acceleration hardware. In a typical acceleration situation, some of the simulation is left on the workstation. The ratio of the percentage of the simulation on the workstation versus the percentage of the design in the accelerator determines the final performance. An example of profile output from a simulator is shown in Figure 2-2. Due to testbench and PLI programs, this example shows that only a 5X speedup can be expected from simulation acceleration.

```
C8 > $finish at simulation time 27386280 ns
-- Simulation execs 2738628 time steps in 128.17 sec (46.80 us/ts,  21367.15
ts/sec)
-- Profile: TB=5.87%  UTF=9.76%  RCC=0.00% DUT=80.51% SYS=3.86%
TB   3.20% in TBplatform.uPlatform.uProcSubSys.uProcCoreMod.uARM926EJS
TB   2.14% in clkgen.sw_clks
TB   0.53% in TBplatform.uTrickWrapper.uAHBTube


UTF 9.76% in TBplatform.uPlatform.uProcSubSys.uProcCoreMod.uARM926EJS
Simulation: cpu time = 128.17 secs, elapsed time = 285 secs, heap memory size =
453.05M bytes
```

Figure 2-2: Example simulation profile

Emulation refers to the process of mapping an entire design into a hardware platform designed to increase performance. There is no constant connection to the workstation during execution, and the emulator receives no input from the workstation. By eliminating the connection to the workstation, the hardware platform now runs at its full speed and does not need to wait for any communication.

There are three commonly used techniques for simulation acceleration and emulation. One uses an array of custom processors to execute the design and other two use an array of FPGAs to achieve parallelism. In the past, acceleration and emulation technology consisted of performing a wire-for-wire and gate-for-gate mapping of an RTL design into a gate level representation for FPGAs just as is done in prototyping. Re-timing issues, glitches, and race conditions often made such technology difficult to use and led to its demise. More recent advances use either ASICs or FPGAs to implement many processing elements that evaluate only a small portion of the design. These computing elements are scheduled in a similar way to how a software logic simulator works to produce the same simulation results at much higher speed due to parallel processing.

There are many different definitions of what the term emulation means and how it relates to simulation acceleration. Following are some of the characteristics that define emulation:

- There is no testbench running on the workstation
- The emulator is the master and the workstation is the slave
- All clocks are generated by the emulator, not the workstation

Simulation acceleration has the opposite characteristics:

- There is some testbench running on the workstation (hopefully not much, to get good performance)
- The workstation is the master and the emulator/accelerator is the slave
- All clocks are generated by the testbench running on the workstation

In-Circuit refers to the use of external hardware coupled to an emulator for the purpose of providing a more realistic environment for the design being simulated. This hardware commonly takes the form of printed circuit boards, sometimes called *target boards* or a *target system*, and test equipment cabled to the emulator. **Targetless emulation** refers to running with no testbench input from the workstation, but also no target system. With targetless emulation all stimulus generation (testbench) is synthesizable and runs in the emulator. There are always gray areas in these definitions since current generation products perform both simulation acceleration and emulation and can switch between these modes by issuing a simple command. For example, when running in emulation mode a user may press Ctrl+c to stop the emulator so there must be some connection back to the workstation. Some emulators also allow synthesizable testbenches to use print statements such as $display in Verilog to print messages on the workstation for debugging purposes; another sign of a loose connection back to the workstation. Diagrams for targetless emulation and in-circuit emulation are shown in Figures 2-3 and 2-4, respectively.

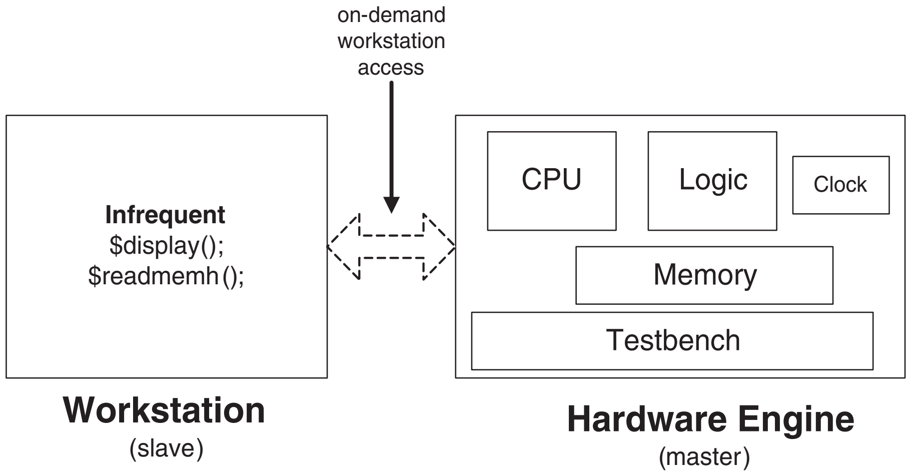


Figure 2-3: Targetless emulation

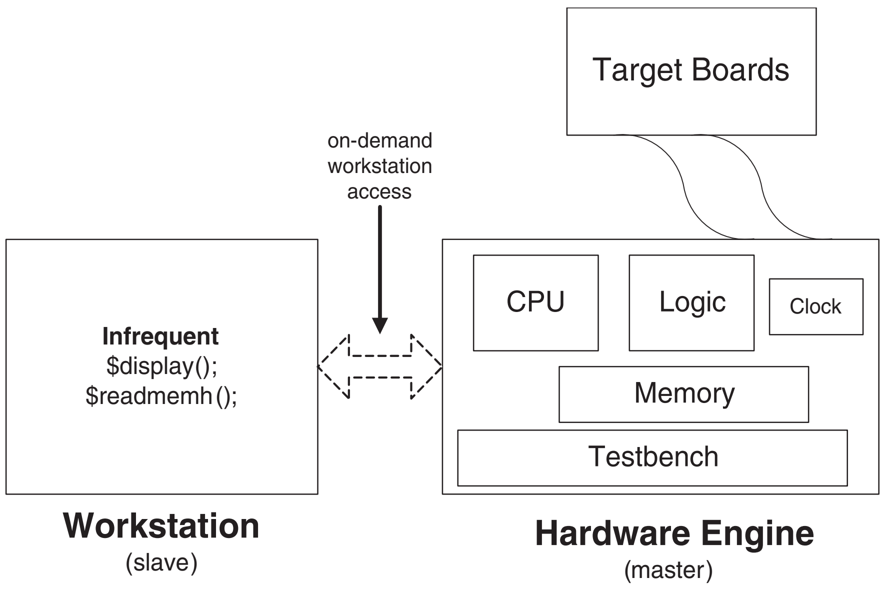


Figure 2-4: In-circuit emulation

Hardware Prototype refers to the construction of custom hardware or the use of reusable hardware (breadboard) to construct a hardware representation of the system. Prototype is a representation of the final system that can be constructed faster and available sooner than the actual product. This is achieved by making tradeoffs in product requirements such as performance and packaging. A common tradeoff for a prototype is to save time by substituting programmable logic for ASICs. This allows the design to be available sooner, but in a representation that runs slower and is much larger.

Software Engineer's View of the World

To the software engineer the entire world revolves around the programming model of the embedded system. Here is a computer scientist's definition:

A programming model is a model used to provide certain operations to the programming level above and requiring implementations of all of the architectures below.

Practically, the programming model for a microprocessor consists of the key attributes of the CPU that are necessary to abstract the processor for the purpose of software development. As an example of a programming model, consider the ARM9E-S CPU from ARM.

From the technical reference manual (TRM) we find that the ARM9E-S implements the ARM v5TE instruction set that includes the 32-bit ARM instruction set and the 16-bit Thumb instruction set. The details of the instruction set are an important part of the programming model. Also covered by the programming model are details related to the operating modes of the CPU, memory format, data types, general purpose register set, status registers, and interrupts and exceptions. All of these details of the microprocessor are important to the software engineer.

Beyond the microprocessor, software engineers are interested in the memory map for the embedded system. For a 32-bit address space, there is 4 GB of physical memory that can be accessed. Embedded systems use only a subset of this physical address space and the memory map defines where in the address space various types of memory and other hardware control registers are located. The memory map may also define what happens if addresses where no physical memory exists are accessed. Figure 2-5 shows an example of a memory map.

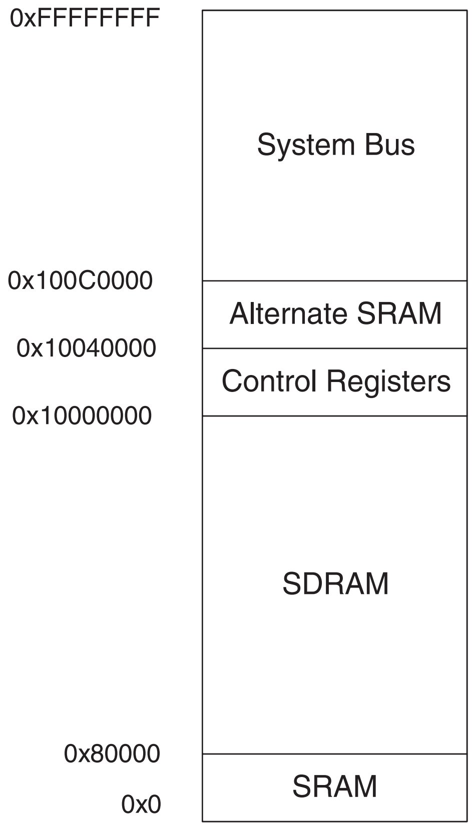


Figure 2-5: Example memory map

Common types of memory in an embedded system are ROM to hold the initial software to run on the CPU, flash memory, DRAM, SDRAM, or DDR memory, SRAM for fast data storage, and memory mapped peripherals. Peripherals can be any dedicated hardware that is software programmable. These can range from small functions such as a UART or timer to more complex hardware like a JPEG encoder/decoder.

The combination of the microprocessor programming model, the memory map, and the individual hardware control registers form the software engineer's view of the embedded system. This information becomes the law to follow for all software development and is available in the form of technical manuals on the microprocessor combined with the system specifications supplied by the hardware engineers or system architects.

Hardware Engineer's View of the World

Hardware engineers have a different view of the embedded system. We saw how the internal operation of the microprocessor is important to software engineers. The internal workings of the CPU are much less important to hardware engineers, and the bus interface is what matters most. For the hardware design to work correctly, the logic connected to the microprocessor must obey all of the rules of the bus protocol. If the rules of the bus protocol are obeyed, the details of what the software is doing are not important. To hardware engineers, the microprocessor is nothing more than a bus transaction generator.

All microprocessors use some type of protocol to read and write memory. At the hardware engineer's level, the microprocessor is viewed as a series of memory reads and writes. These reads and writes are used for fetching instructions, accessing peripherals, doing DMA transfers, and many other things, but in the end they are nothing more than a sequence of reads and writes on the bus.

Example

To demonstrate the differences in how software and hardware engineers view the world, look at the following example. Consider a register that is programmable from software. The register is the definition of a control register from the ARM926EJ-S MMU. It is accessed using coprocessor read and write instructions. The bit definitions of the register are shown in Figure 2-6. Software engineers writing low-level code typically have tens or hundreds of such registers to program. They can be accessed using coprocessor instructions or be memory mapped and accessed using data load and store instructions.

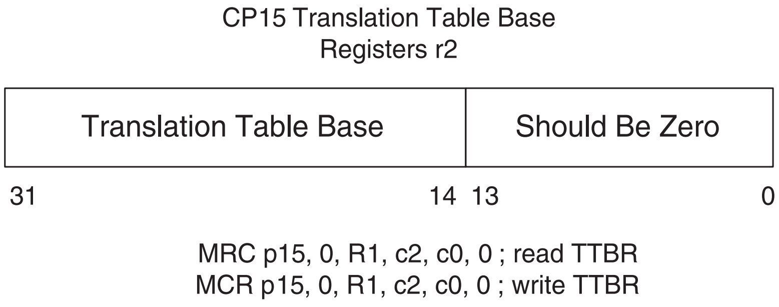

CP15 Translation Table Base
Registers r2
Translation Table Base
Should Be Zero
31
14
13
0
MRC p15, 0, R1, c2, c0, 0 ; read TTBR
MCR p15, 0, R1, c2, c0, 0 ; write TTBR

Figure 2-6: Example programmable register

The hardware design looks much different. The particular block of code implementing the register will not use all 32 bits of address, only a subset of the address. Also, not all bits of the register may be easily located as a 32-bit wide register in verilog. The path from the CPU bus to this register is not so easy to trace. A fragment of how a configuration register for a memory controller is implemented is shown in Figure 2-7. For this 32-bit register only 8 bits are meaningful and the register is implemented as multiple smaller registers that are concatenated together to form the register value when read from software.

```
// ---------------------------------------------------------------------------
// Offset | Register Name | R/W | Valid Bits   | Reset | Description
// ---------------------------------------------------------------------------
// 0x00     MPMCStConfig    R/W   20:19,8:6,     0x0     Static Memory
//                                3,1:0                  configuration

assign MPMCStConfig  = {HWDataReg20to19Q, 10'b0000000000, HWDataReg8to6Q[8:6],
                        2'b00, HWDataReg4to0Q[3], 1'b0, HWDataReg4to0Q[1:0]};
```

Figure 2-7: Example register implementation

Software Development Tools

Editor

OK, maybe an editor is not really a software development tool, but most software engineers spend more time with their editor than any other program. It's a bit of a paradox to consider the power and complexity of desktop applications available today to computer users (think of the feature bloat of Microsoft® Word for example), but at the same time engineers that make a living with computers still use the same editors that were used thirty years ago, programs like vi and emacs. Experience has shown that software engineers are more likely to have better editors than their hardware counterparts, but there is usually room for productivity improvements.

Source Code Revision Control

Once some software exists, the next step is to make sure it doesn't get lost or otherwise broken by accident. Revision control is a means of recording incremental steps during software development. Using revision control, it becomes possible to undo changes that have been made if the changes cause problems. It can also provide a means to limit who can modify particular files as well as to identify who made a particular change.

This is potentially very useful in the development of software where there are multiple developers who may be working in different locations. Without it, projects operate in an environment where control of the source code is maintained entirely by word-of-mouth.

There are many such revision control systems ranging from free to expensive. Concurrent versions system (CVS) is the de facto standard in the world today. It runs on most every platform and provides an easy client/server model for people in any location. It may not be the best, but it is the most commonly known. Some examples of how to use CVS to checkout files and view file history are given in Figure 2-8.

```
x8:46 % cvs update Makefile
U Makefile

x8:44 % cvs status Makefile
===================================================================
File: Makefile          Status: Locally Modified

   Working revision:    1.20
   Repository revision: 1.20    /home/tools/x-cvs-repos/arm/Makefile,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

sp8:47 % cvs log Makefile
RCS file: /home/tools/x-cvs-repos/arm/Makefile,v
Working file: Makefile
head: 1.20
branch:
locks: strict
access list:
symbolic names:
        release_2002_1_1: 1.17
        release_2002_1_1: 1.16
        release_2001_3_2: 1.9
        start: 1.1.1.1
keyword substitution: kv
total revisions: 21;    selected revisions: 21
description:
----------------------------
revision 1.20
date: 2002/11/19 18:37:30;  author: toolman;  state: Exp;  lines: +1 -3
Fixed gtk target for Linux platform.
```

Figure 2-8: Using CVS

Compiler

A compiler is a tool to translate C/C++ and assembly language text files into a program the computer can run. This format is called *machine language* and is stored in a file called an *executable file*.

Embedded system projects normally use a cross compiler. A cross compiler runs on a computer with a different processor type than the one used in the embedded system. This means the executable file produced by the compiler cannot be run on the machine that created it and must be transferred to the embedded system to be run.

Debugger

Unfortunately, not all programs run correctly the first time. It may be more accurate to say that 100% of all programs do not work correctly. Even if a program seems to work correctly, it is impossible to test it under every situation and circumstance, so it could never be proven to always work correctly. A debugger helps software engineers find enough problems so that the code runs correctly for most of the situations it faces. A debugger allows the software engineer to inspect the sequence of the code, the CPU registers and memory to figure out what is happening.

Common debugger functions include:

- View and change registers
- View and change memory
- Display function call stack (backtrace)
- Set breakpoints
- Set watchpoints (data breakpoint)
- Single step by C statement or assembly instruction
- Interrupt the running CPU

Simulator

For a software engineer, a simulator models the internal workings of the CPU. CPU simulators are often provided along with software development tools used to compile software. Another name for this type of simulator is an instruction set simulator (ISS). One application of the simulator allows compiler writers to test compiler output without having a CPU chip available. When a new processor is designed the compiler must be completed early so programs can be run immediately on the first implementation of the processor. The ISS is also used by microprocessor designers to cross-check the CPU design against the model to spot differences. Using the simulator as a golden reference model is a good way to verify and debug microprocessor operation. Software engineers may also use a simulator to test code without having a chip or board available. For early software design it is easy to run the initial boot code on a simulator to make sure the details of the CPU setup are correct.

Development Board

A development board contains a CPU, memory and peripherals, and a way to download and debug programs. The development board is the de facto standard for software engineers. When a new project starts or a new processor is introduced software engineers will immediately get a development board and start trying to build and run programs to make sure the compilation and debugging tools are working. Microprocessor vendors provide these boards for CPU evaluation and to show the CPU is real, it works, and software engineers can try it out very easily.

Integrated Development Environment (IDE)

The IDE combines all necessary functions and tools for the software engineer into a single, integrated product that normally includes project editing, source code search, file navigation, file editing, and project building. The idea is that learning a single tool makes life easier than having different applications that must be run manually for each function. Efficiently navigating source code saves time and makes it easier to understand how the software works.

Software Debugging Connections

Software debuggers connect to embedded systems in many different ways. Even though the debugging concepts are the same for all connection methods, it is useful to understand the underlying mechanism the debugger is using to communicate with the microprocessor and the memory. Debuggers work by sending commands to the embedded system to perform operations on the CPU registers and memory. The two most used functions are reading CPU registers and reading memory. With this information the debugger can determine the context of software execution. Almost all debugger commands require memory to be read; examples include printing variables and viewing disassembled software instructions. Other debugger functions are the less often used register and memory write and control operations to start and stop software execution. Following are some of the ways software debuggers exchange information with embedded systems to provide debugging functions.

Unlike debugging a program on a PC or workstation, the embedded system software debugger runs on another machine, not the embedded processor. This concept is often called *remote debugging*. The target processor may not work correctly and may not have any keyboard and monitor anyway.

JTAG

One way debuggers can connect to a CPU is JTAG. JTAG was developed as the IEEE 1149.1 standard for boundary scan testing of printed circuit boards. It uses five wires to send and receive serial commands and data from devices that implement the JTAG standard. Microprocessors use the same JTAG protocol not to test PCB connections, but instead to send and receive information from the embedded system. JTAG debugger connections provide a way to link a machine running the debugger to a connector on the embedded system that implements the required JTAG signals. Since the serial JTAG sequences can be very long, a small box that converts commands into the serial data signals is used to provide good performance. To use JTAG debugging, the microprocessor must implement special hardware for this purpose.

Stub

For processors that have no hardware support for debugging, the software debugger can communicate with the embedded system using a stub. A stub is some special code that the user adds to the embedded software for the sole purpose of communicating with the debugger. The stub software communicates with the debugger using a communication channel such as a serial port or Ethernet connection. The stub performs debugger requests such as reading registers and memory and sends the results to the debugger via the communications channel. It also uses an interrupt service routine to gain control when the debugger wants to stop, such as when the user hits Ctrl+c to stop execution. A stub is somewhat intrusive since special code that is not required for product operation must be run on the embedded system, but when the stub is working it is a very flexible way to debug software.

Direct Connection

When using an ISS there is no need for either a JTAG connection or a stub between the software debugger and the embedded system. Since the ISS is a software model, the debugger can easily access all required information by making function calls to the ISS. A direct connection is the best connection between debugger and embedded system since it does not require any special code to be inserted into the embedded software and does not require any hardware connection or cables. All debugger requests can be satisfied immediately, without the need to modify the state of the CPU or enter any special debug mode. This provides the most accurate and realistic picture of how the embedded system operates.

Types of Software

Five distinct types of embedded system software have been identified, with software design proceeding in this order. The software content (lines of code) increases with each step:

- System initialization software and hardware abstraction layer (HAL)
- Hardware diagnostic test suite
- Real-time operating system (RTOS)
- RTOS device drivers
- Application software

System Initialization and HAL

The first software-coding task is to write and test the microprocessor initialization software. This code includes configuring the operating modes and peripherals (things like cache configuration, memory protection unit or MMU programming, interrupt controller configuration, timer setup, and DRAM initialization).

The hardware abstraction layer is the next layer of software that works with the initialization code to provide a common interface for higher-level software to use for hardware-specific functionality after the system is initialized. The HAL abstracts the underlying hardware of the processor architecture and the platform to a level sufficient for the RTOS kernel to be ported.

Diagnostic Suite

Once the initialization software and HAL are stable, the next phase of software development consists of developing a detailed test suite for the hardware design. In the past this usually took the form of a hardware testbench. While testbenches are still necessary to provide stimulus for external interfaces such as networking protocols, the software now serves as the testbench for the CPU core bus. A comprehensive set of diagnostic tests are developed to verify each subsystem and peripheral. This starts with the memory subsystem, progresses to interrupt testing, then moves to other IP blocks like timers, DMA controllers, video controllers, MPEG decoders, and other specialty hardware. Most of these tests do not see their way into the final product, but they are very important because they build the case for a solid hardware design. Creating the programs gives the software engineers a very good understanding of the hardware and serves as a chance to learn about the hardware specifics in a more secluded environment. Diagnostics can also be reused to verify operation when the hardware arrives.

Real-Time Operating System (RTOS)

The first assumption of the RTOS engineer is that the hardware is stable. This is true if the diagnostic suite was done well. The initial RTOS work consists of just getting it to boot on the platform. How big a task the RTOS is depends on how standard the platform is and how well the HAL was designed. During the initial porting phase, device drivers for application specific hardware may be missing, but the RTOS can still boot.

Device Drivers and Application Software

Once the platform is complete, it is time to test device drivers and application software. At the application level the hardware is assumed correct, and the task becomes more like workstation programming. If the application crashes, it can be safely assumed it is a software problem, not something wrong with the hardware.

Applications usually want to interface to real network traffic, see things on the screen, and use the pointer or mouse. During application development, hardware and lower-level software bugs are few and far between and the software engineer is focused on providing robust applications with differentiating features for end users.

In the coming chapters, methods to verify all types of software will be discussed, but the primary focus will be on hardware dependent software. The software that interacts most with the specifics of the hardware design is the critical place in the project where hardware and software integration is most important.

Software Development Process

Writing embedded software is different from writing a program for a workstation or PC. Programming aspects that are not important on a workstation suddenly become important when working in the constrained environment of an embedded system. As an example, consider the memory constraints of an embedded project. Memory layout is important, how much memory used is important, how to get the right data into the memory is important. This is different from a workstation program running on a modern operating system with large amounts of virtual memory.

For the last few years, I have been following the developments of an open source tool that is used to organize software projects, analyze the code, make it easier to navigate, build and debug. In the software world, it would be classified as an integrated development environment (IDE). I took interest in it because I had extended it to work with the Verilog language and used it for my daily development tasks. Like most open source projects (including the Linux kernel) the development effort was all volunteer and driven by e-mail lists. One of the e-mails on the list reminded me again how different embedded software development is from writing applications for PC or workstation. An embedded software engineer started out by reminding everybody that he does not really like these kind of graphical IDE tools and he really doesn't need them since he can do everything using the command line, but in this case it saved him. He went on to tell how his product had run out of memory and he was required to add one more feature. Programming the feature was trivial, but squeezing it into memory was not. Using the tool, he was able to analyze his project and free up 1.1k bytes of unused code. In the workstation world where a web browser chews up 20 MB of memory, 1.1k bytes may not seem like much, but it was enough for this guy to add the feature and finish the project.

Software engineers write code in a text editor using the programming language of choice. The next step is to compile the source code into object files, then link those object files together, usually with some libraries provided by the compiler and operating system. The result is an executable file that can be directly executed on the computer. This is the same process for developing software that runs on a PC or workstation.

One file format often used in both workstation programs and embedded programs is ELF (executable and linking format). ELF files are object files produced by the compiler and linker that are binary representations of programs intended to be executed directly on a processor. The standard for ELF defines the file format including a header and different sections that allow the files to be easily identified and understood in a machine independent way. Understanding the details of the file format is not usually required to design and develop software for embedded systems, but some familiarity is useful.

The workstation program can easily use pre-compiled libraries to access common functions provided by the operating system to access graphics displays, network interfaces and input devices like keyboard and mouse. Workstations also use dynamic linking to avoid duplication of data by loading libraries at runtime, when the program actually needs them. Dynamic linking does not include all of the libraries inside the final executable, but includes only references about where to find them. This keeps executable file size small and the operating system can load the libraries when they are actually needed.

An example of running "hello world" on a Linux workstation is shown in Figure 2-9.

```
% cat hello.c
main()
{
   printf("Hello World\n");
}

% gcc -o hello hello.c
% ldd hello
        libc.so.6 => /lib/libc.so.6 (0x4001e000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
% hello
Hello World
```

Figure 2-9: Example workstation program

Contrast this software development process to that used when writing a diagnostic program for an embedded processor. The diagnostic program has no operating system and no standard libraries to communicate with input and output devices (if there are any). Engineers will often call this operating environment *bare hardware*. The code usually goes something like the following:

- Software starts execution from the reset vector. It is different for each architecture, but in the case of ARM it is usually address 0.
- Setup interrupt vectors by programming addresses of interrupt handlers into the table of vectors. As an example, in the ARM architecture address 0x18 contains the address of the handler for the nIRQ interrupt (normal interrupt) signal.
- Configure and enable other hardware. This can include enabling the instruction and data cache and programming the memory controller.
- Setup a stack and call __main to prepare for program execution.
- Start execution from the main() C function.

Everything up to the final step is done in assembly language to prepare the CPU to run even the simplest C program starting from main. This flow provides a look into what is necessary to work in the world of embedded systems compared to workstation programming.

Now let's take a look at the compile process for an embedded program to understand how it differs from a workstation program. I saw the message shown in Figure 2-10 in a newsgroup related to embedded software development.

Hi,

I am new to this, but I understand that I need a certain amount of assembly code to "boot" the arm720t processor. Now this should be a standard thing for you guys, so although I have been unable to dig up any info about this so far, I was hoping you could either point me in the right direction, or perhaps even help me aquire the needed code. Or do I really need to "re-invent the wheel" so to speak!? The arm7 must be set up properly before it can execute my c-program. As all arm720t cpus are equal (?!) so should their init code be (!?), so why can't I just download this code from *www.arm.com* !? ...or from any site dealing with arm development—kinda hard to develop without even getting the thing running. I'm sure this is obvious to you guys, but please help me out.

In danger of having misunderstood how things work, I am open to all input on the matter!

Figure 2-10: An engineer trying to understand embedded software development

Once the diagnostic program is coded in C and assembly language it is compiled into object files for the target processor. This is the same as a workstation program. Figures 2-11 and 2-12 show the compilation commands for both C and assembly language required for a small test program.

```
armasm -32 -bigend -checkreglist -CPU 5T -keep -apcs /inter -g \
  -i include init.s -list init.lst -o init.o

armcc -ansi -c -cpu 5T -zo -bigend -fy -g -O2 -I include main.c -o main.o
```

Figure 2-11: Example commands to compile C and assembly language into object files

The first command is an example of an assembly language file that is used to initialize the CPU. The second command is a C file that contains main(). The next step is to take all of the object files and link them together to form an ELF file named test.elf.

```
armlink -debug -scatter link.txt -remove -noscanlib -info  \
sizes,totals,veneers,unused -map -symbols -xref main.o cp15init.o \
/tools/ads11/common/lib/armlib/c_a__un.b   \
/tools/ads11/common/lib/armlib/f_a_m.b     \
/tools/ads11/common/lib/armlib/m_a_mu.b -list test.map -o test.elf
```

Figure 2-12: Linking object files to produce an ELF file

For embedded systems, the memory layout is extremely important. On a workstation the software engineer does not care where in memory the program is loaded or about anything related to the physical addresses of the hardware. In embedded programs, the software engineer must use a link map as shown in Figure 2-13 to specify the location of the memory and where to put ROM, RAM, and stack data.

```
LR_1 0x0
{
        E_INIT_VEC 0x00
        {
                int_vectors.o(+RO)
        }
        E_RO +0
        {
                .ANY(+RO)
        }


        E_TABLE +0 UNINIT
        {
                init.o(image)
        }
        E_RW 0xFFFE0300
        {

                .ANY(+RW)

        }
        ER_ZI +0
        {

                *(+ZI)

        }
}
```

Figure 2-13: Example link map

Once the ELF file is complete and located correctly in memory, the final step is to translate the ELF file into a format that can be used to program a flash memory or otherwise load the data into the embedded system. Since this book is focused on the verification of hardware and software before any chips or boards are constructed, the ELF file must be translated into a format that is suitable for a verification platform such as a logic simulator or emulator.

A logic simulation environment will contain memory models for the various types of memory in the design. Common practice is to load the code into these memories at the start of simulation. One way to do this is to generate hexadecimal memory files that can be read directly into the memory models as in Figure 2-14.

```
% fromelf -vhx -16x2 test.elf -output rom0.dat
```

Figure 2-14: Command to translate ELF file into Verilog memory format

Verilog provides a system task, $readmemh, that will read the memory data from a file into the memory array. The task takes two arguments; the filename and the memory array name to load the data into. A code fragment is shown in Figure 2-15.

```
// Memory
reg  [15:0] data [0:16383];

initial $readmemh(“rom0.dat”, data);
```

Figure 2-15: Loading memory data

The above example shows all of the steps to go from C and assembly language source code to a simulation memory model and finally to a complete simulation of the ARM design. When the reset of the microprocessor is completed it will begin to fetch instructions from memory and execute the program. Understanding how embedded software is developed and simulated is crucial to understanding the world of hardware/software co-verification.

Hardware Development Tools

There are two primary classes of tools used by hardware engineers; those dealing with design implementation and those dealing with design verification. Implementation is the process of describing a design and transforming the design description into necessary formats to manufacture a chip or a board. Verification is the process of making sure the design works, it does not crash, and it does this according to the specification. For our purposes design verification is the focus, so let's review some of the most common verification tools.

We have already discussed the verification platform as the method used to execute a description of the hardware design. The most common platform is the Verilog and VHDL simulator running on a PC or workstation. We also discussed other platforms that provide features and performance beyond the simulator for the purposes of simulation acceleration and in-circuit emulation. An entire industry has sprung up to provide products that augment the verification platform to help engineers develop tests and interpret the results. There are special hardware verification languages that have been designed to improve efficiency and verification quality. Other commonly used tools are code coverage, lint tools, and debugging tools to visualize results.

Another technique quickly gaining popularity is the use of assertions as a way to document the designer's assumptions and the properties of the design. Assertions are a powerful tool to crosscheck the design's actual versus intended behavior. Assertions are also valuable to verification and system engineers to specify the intended behavior of the system formally and to make sure it is acting according to specification.

Editor

Just like software engineers, most hardware engineers enter a design description and testbench using a text editor. In the case of hardware engineers, the editing techniques used are even more primitive than software engineers. One reason is that most hardware engineers involved in complex design are working in a UNIX or Linux computing environment. The primary focus is on simulation performance and the ability to share a network of workstations for most tasks. Another reason for the use of primitive editing tools is that the relative size of the market for engineers developing in Verilog and VHDL is much less than for a software development language like C++ or Java. This smaller market does not get as much attention from potential tool providers, especially when you consider that most of the market is for tools on UNIX and Linux.

Source Code Revision Control

Another area where hardware engineers are following software engineers is the area of revision control. From the beginning of time, software development has always involved managing a large number of text files that are produced by many engineers and compiled into one or more executable programs. With the advent of Verilog and VHDL design flows, hardware engineers are dealing with the same type of environment and must use revision control tools to manage projects. Many revision control tools do not really care if the files being checked in and out are C++, Java, Verilog, or VHDL; after all, a file is a file in any language.

One area where revision control may differ between hardware and software is the amount of data generated and stored in hardware design projects. Compiled software projects consist of many object files and some number of executable files. Intermediate versions of the binary files are not usually archived permanently because even a very large project can be rebuilt in a matter of hours. Hardware projects are different in that it may take many days to create the proper binary files and databases to go from HDL source code to a layout of a chip or board. This process involves many different tools and intermediate files. This leads to much more data and a higher importance to save this data using revision control tools. For these applications, it is important to find a tool that is capable of handling large amounts of binary data efficiently. There are even some tools that are specifically targeted for revision control and synchronization of these hardware databases across design sites in different locations.

Lint Tools

Lint tools provide an easy way to analyze code for common mistakes by simply reading the source code. Lint originated in the early days of C programming as a tool to find errors before runtime. Common lint checks include unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall through cases.

The concept of lint has been extended to Verilog and VHDL. HDL lint tools provide syntax checks and coding style checks for simulation and for synthesis. They can also detect race conditions and analyze properties of finite state machines. The benefit of lint tools is to analyze the code before the compile and simulation process. There is nothing worse than starting a compile of a large project only to find out the compiler reports a coding style problem 15 minutes into the compile, or to find a race condition after spending 45 minutes to compile, run simulation, and use a waveform to examine the result. The goal of lint tools is to achieve higher quality code and catch bugs earlier by analyzing the design before simulation.

Code Coverage

Code coverage is another technique borrowed from software engineering. Coverage measures areas of code exercised when tests are run. Based on coverage results, additional tests can be developed to increase coverage. Indirectly, increasing coverage will increase product quality. For HDL design there are different types of coverage metrics that are measured, such as statement coverage, branch coverage and FSM coverage.

Statement coverage, also called line coverage, measures which statements were executed. Statement is the most basic type of coverage metric that identifies unexecuted lines of code for which either a test should be developed or maybe the code can be removed.

Branch coverage identifies conditions that were satisfied in branch statements such as if-then-else and case statements. It provides more detail than simple statement coverage and will identify areas where not all possible conditions were exercised during testing.

FSM coverage identifies design behavior by identifying a set of statements as a finite state machine and analyzes the set of FSM transitions that occur, to make sure all behavior is tested.

Code coverage is a good way to make sure HDL code is tested, but doesn't guarantee the design is correct. Even if an incorrect design is well tested, this does not help in the end since coverage provides metrics on the implementation and has no correlation back to the design specification.

Debugging Tools

Unlike some of the previous tools, which were adopted by hardware engineering from software engineering, hardware debugging is very different from software debugging. Hardware engineers operate primarily in a batch environment. Whether they are working on large simulations and using a farm of computers to run the jobs or working on a single module and simulating it on a local workstation, the debugging is often done using post-processing techniques. This means that the step of running the simulation and capturing results is separate from the step of interpreting results and debugging problems.

The most common tool for hardware debugging is a waveform viewer. Hardware engineers operate by viewing signal values across a span of simulation time. A particular simulation time is referenced by the simulation timestamp. Today's waveform tools save data in compressed formats for maximum performance and minimal file size. The IEEE Verilog standard does specify a format for waveform files, known as value change dump (VCD). This text format is fine for small simulations with small amounts of data to store, but becomes impossible for very large simulations. For large simulations, proprietary formats with compression must be used.

Verification Languages

As design complexity has increased, it has become more difficult to use simulation languages to verify a design adequately. New languages have been proposed and developed that are specifically tuned for verification. Although there are many languages, one of the most popular is the *e* language invented by Yoav Hollander, founder of Verisity, which is currently in committee to become IEEE standard 1674.

To improve verification of HDL designs, additional capabilities are required than what is currently available in Verilog, VHDL or C. Some of the useful aspects of a verification language are:

- Easy to reuse code for multiple tests
- Constraints for directed-random test generation
- Built in assertions and functional coverage support
- Data types that provide both high-level functionality and interface well to HDL designs

The idea of a language that is specific for functional verification is appealing to many projects that are struggling with complex verification and the manual work required to develop testcases.

Assertions

The use of assertions to specify design intent and the properties of the design is rapidly growing in popularity. Assertions are a powerful tool to crosscheck the design's actual versus intended behavior. They are also valuable to verification and system engineers to formally specify the intended behavior of the system and to make sure it is acting according to specification. Recently, much has been written about assertions and even though the concept has existed as an ad hoc technique for years, there is still much confusion over their use. Some of the details of how assertions are specified and implemented are presented in Chapter 6. For now, let's take a look at the goals and benefits of assertions.

Assertions provide a common format for multiple tools. Without an agreed upon method of specifying assertions, there is no chance to automate activities such as functional coverage metrics, error reporting and severity levels. A common assertion methodology lends itself to increased automation in both formal verification and simulation. Assertions can also benefit from a "write-once, run-anywhere" characteristic. Once inserted into a design, they can be used by many tools and are quite portable for design reuse.

Assertions are better than a paper test plan. Verification involves deriving a list of features to be tested from the design specification. Most projects write a test plan that dictates how the features of the design are to be verified. These features are then documented, and a testcase is developed to verify each feature. Testcases usually take the form of a testbench that will cause the desired feature to be exercised. This directed approach may be augmented with random testing. Assertions help to automate the manual process of running a testcase; visually verifying the test has covered the feature, and adding the test to the regression suite. Without using assertions, there is often no way to guarantee the testcase still exercises the feature as the design evolves. Assertions provide a mechanism to measure functional coverage and quantify test results.

Assertions ease debugging and reduce simulation time. One of the main motivations for assertions is reduced debug time. Since assertions are a white-box verification technique they provide increased visibility and controllability of the design under test. Assertions will detect design errors as soon as they occur without waiting for the effects to be propagated to the device or system boundaries. Having an immediate indication of a problem can save hours of trying to look backward to get to the root cause of the problem. The use of assertions has been reported to reduce debug time by as much as 50%. Assertions clearly have the potential to reduce debugging time. A runaway or meaningless simulation can waste valuable simulation time if it occurs during an overnight regression run when more tests could be given a chance to run.

Assertions verify interfaces between blocks. During integration phases, assertions act as watchdogs at the module or block interface boundaries, making sure each block obeys the agreed upon protocol. This can be crucial, since it is possible for a testcase to pass based on the data observed at the chip or system boundaries even when there is a violation of a protocol inside the design. Assertions help eliminate such situations.

Assertions are a good RTL coding practice. Assertions are far more valuable than comments alone in the design. Well-commented code makes it easier to maintain and to understand the intended functionality. Assertions go one step further by documenting exactly what the code is expected to do in a way that can be verified using tools instead of a human reading the comments and attempting to understand if the code is working correctly. Assertions are also a good way to sanitize the code before check-in. Tools can read a design file and its assertions and indicate possible problem areas. In the same way a lint program checks for errors in the code's syntax and structure, assertions can be used to check for errors in the design's behavior.

Assertions provide many benefits to system-level engineers, design engineers doing RTL coding, and verification engineers. Each group brings to the table different knowledge about the design and its operation, but a common assertion methodology allows all parties to benefit.

Debugging Defined

What most engineers call "debugging" is actually a combination of two separate activities: *detection* and *debugging*.

Detection is the process of determining that there is a problem in a design or test. Most projects use simulation as the primary means to detect problems. More recently, formal verification tools have been used to detect design problems. In simulation, there are many ways to communicate the existence of a problem. Two common ways to communicate problems are using $display (print) statements and comparing memory values with expected results. The lack of a $display statement may also indicate a problem with a test result. Assertions are a better way to formalize ad hoc detection methods.

To increase problem detection, engineers can apply more computers and simulators or use other ways to increase performance, such as simulation acceleration or emulation. In addition to the brute-force methods of adding more performance, engineers can develop additional or smarter tests that will uncover the problems faster or detect them sooner. Examples include developing directed tests that target a specific area, or using feedback from code coverage to order tests in a better way.

Once a problem is detected it must be debugged. *Debugging* is the process of finding the root cause of the problem and changing the design or test to correct it. Debugging is a manual process when compared to detection, because it requires an engineer to spend time to figure out what is right and what is wrong. Determining the correct behavior requires good knowledge of the design and how it should work. More or faster computers do not help the debugging process. Tools can help improve the process by helping to improve the understanding of the design and by providing better views of simulation results.

The primary means of debugging is viewing waveforms. Using the detection information, engineers must look at logic signals at different times during simulation and find out what is not working correctly. This process typically consumes many hours of trying to figure out how a design is supposed to work and why it is not working correctly. Waveform dumping can also drastically slow down simulation performance and result in very large data files. Engineers spend time trying to decide **before simulation** when to generate waveform files and which parts of the design to capture.

In embedded system verification there are two important areas that must be well understood and will be reviewed in the following sections; memory models and microprocessor models.

Memory Models

Memory models for discrete memory components as well as embedded on-chip memories are used in logic simulation. As we have discussed, software must be translated from binary formats like ELF into something suitable for memory models to load. There are many kinds of memory models ranging from Verilog and VHDL models to C models that use the logic simulator's C interface to communicate with an HDL wrapper. An example of a simple memory model is shown in Figure 2-16.

The IEEE VCD format for storing waveform data does not have any provision for handling the contents of memories. To better understand memory activity, special handing must be done. One alternative is to use a proprietary waveform file format that is capable of storing memory history. Most hardware debugging tools included with logic simulators as well as independent debugging tools provide this provision. For example, Debussy from Novas Software provides the $fsdbDumpMem to dump

```
// Simple read-only memory
module ROM(A, OE, D);

input [13:0] A;   // Address
input        OE;  // Output enable
output [7:0] D;   // Data output

reg      [7:0]  D;
reg [7:0] Mem [0:(16 * 1024)]; // memory array

// Read from memory
always @(OE or A)
begin
  if (OE)
    D = Mem[A];
  else
    D = 8'bz;
end
endmodule
```

Figure 2-16: Simple memory model

memory contents into its proprietary file format and the Synopsys VCS simulator offers a similar task $vcdplusmemon to dump memory into its proprietary file format. Both tools provide GUI functions to display memory contents as saved in the waveform files. These work well for memories defined using a Verilog array. The second alternative is to use memory models, such as those from Denali Software, that are written in C and have an API to read and write memory contents. This allows graphical tools to view memory history both interactively during simulation and after simulation in a post-processing mode. It also allows memory to be changed during simulation using a GUI.

Understanding how memory models load data and where they are in the design's memory map is crucial because memory is one of the key areas where hardware and software meet. As we will see, memory is one of the most important areas in hardware and software co-verification.

Microprocessor Models

Another tool used by hardware engineers is the microprocessor model. The two primary models used for hardware design are the bus functional model (BFM) and the full-functional model (FFM). A full-functional logic simulation model is necessary for system-on-chip designs. It is impossible (or at least risky and impractical) to build an ASIC containing a microprocessor without a full chip simulation. Software is necessary to use a full-functional model. The software must be cross-compiled for the target and loaded into the logic simulation memories. The processor will then fetch and execute instructions, thus verifying the system design. Full-functional models come in a variety of formats. The fastest are written in C and use only a wrapper or shell to communicate with the logic simulator. Sometimes RTL or gate level descriptions of the microprocessor are used as full-functional models. This can be the actual design database (sometimes encrypted) that is used to manufacture the device. While full-functional models are necessary, many hardware engineers do not want to become software engineers just to test the hardware design. For this reason, hardware engineers also use bus functional models.

In the high performance system-on-board area, a full-functional model is not usually available for high performance microprocessor chips such as the PowerPC or MIPS. To fill this need, hardware engineers have turned to the BFM and the hardware model for verification. The bus functional model is less complex than the full-functional model. It just implements the bus interface portion of the microprocessor. Bus transactions like memory read, memory write, instruction fetch and interrupt acknowledge are simulated at a task level. A testbench is used to generate a sequence of possible transactions the processor may perform. This testbench-driven BFM may or may not reflect the transaction sequence that will occur when the embedded system software is run in the target system, but is much easier to use for a hardware engineer. Bus functional models are usually written in C using the logic simulator's C interface or in behavioral Verilog or VHDL. Recent advances in the microprocessor bus protocols are making BFMs harder to write. Techniques such as bus pipelining, bursting, out-of-order transaction completion, cache snooping, and cache interventions make can make it impractical to construct a home-brew BFM.

Another technique used for board simulation is the hardware model. A technology that has existed for over 15 years, the hardware model uses actual device silicon mounted in a socket that interfaces with a logic simulator. The device serves as its own model. Hardware models can be used for most any digital device for which a simulation model is needed. Things like microprocessors, digital signal processors, bus interface chips, network processors, and system controllers are common. Hardware models are usually deployed for those devices for which a software model would be too complicated to write and full functionality is needed.

All of these microprocessor models are being used to help solve the hardware and software integration problem. By themselves they are useful for hardware verification, but do not naturally involve the software engineers and software debugging. We will see in Chapter 4 that some of these models will serve as a base technology for building more useful ways to solve the hardware and software integration problem that are suitable for both hardware and software engineers and don't require software to be debugged using simulation log files and waveforms.

Hardware Design Process

The hardware design process consists of deciding on the architecture and behavior of the design, proceeding with implementation using Verilog or VHDL at the register transfer level (RTL), and taking the RTL code through a set of steps to produce chips and boards. The set of steps from RTL to chips and boards is well defined. Activities at the architecture level are much less defined and vary greatly.

From a verification viewpoint, the process is first to verify each RTL block as it is produced, then accumulate the blocks into subsystems and systems performing verification along the way until the entire design is verified all together.

The key issue in SoC design is when to spend time on hardware and software integration. Some projects advocate doing this very early, in the architecture stage, to make sure there are no performance issues that will show up later. At this early phase an abstract model of the design can also be used to provide a high-performance way to run software. Others advocate delaying integration until the RTL code of the design is ready and all of the blocks have been verified together and all major subsystems are working. Regardless of the specific methodology, it is important to do the hardware and software design in parallel and meet for integration sometime before the hardware is committed for fabrication.

Microprocessor Review

As we have seen, hardware and software engineers have a different view of the microprocessor. It is convenient to break down the functions of the microprocessor into two parts: the internal operation and the external interface. Internal functions include:

- *Instruction Set*. The format for the instructions that the processor can run. Instructions can be encoded in many ways. For RISC processors the instructions are a fixed length such as 32 bit or 16 bit instructions.
- *Registers*. Some combination of general-purpose registers, status registers, and program counter.
- *Cache*. Special memory to store frequently used data. Deciding which data to cache and maintaining cache coherency is typically done by hardware.
- *Pipeline*. The different stages of instruction processing to provide increased performance. Embedded RISC processors may use a 3 or 5 stage pipeline.
- *Memory Management Unit (MMU)*. The MMU provides address translations necessary to implement virtual memory and is used by operating system to provide protection from programs or tasks crashing the system.

To get outside the microprocessor there are load and store instructions (RISC) and a memory map that defines which memories or registers will be accessed. The external interface of the microprocessor is of primary interest to hardware engineers. This interface consists of a set of signals that follow a defined protocol. This external interface includes:

- *Memory Bus Interface*. The rules for bus arbitration, master and slave to obey in reading and writing data on the bus. This is how the microprocessor gets data in and out.

- *Coprocessor Bus Interface*. Sometimes a separate, dedicated interface is used for communication with a hardware function that requires special attention to performance.
- *Interrupts*. A set of signals used to tell the microprocessor that service is needed. Typically there are multiple interrupt sources with different priority levels.

To become proficient in embedded systems a good understanding of computer architecture and microprocessor operation is required. Readers wishing more detail in this area should consult one of the many textbooks available on computer architecture.

Hardware and Software Interaction

To set the stage for the techniques used to verify hardware and software, consider the debugging methods used by engineers.

Software Debugging Characteristics

- Completely interactive
- Historically done in a lab with a board running at 25 MHz or faster
- Use printf( ) statement to trace execution and variables
- Trace software execution with source-level debugger
- Use breakpoints to stop execution and inspect memory (variables and data structures), call stack and register contents
- Iteratively reboot the system and adjust breakpoints until bugs are found

Hardware Debugging Characteristics

- Run logic simulation at speeds of 10–100 Hz
- Use print statements (or lack of print statements) to detect errors
- Use of waveform dumps to examine signal values until bugs are found

This list of characteristics demonstrates the challenge of verifying a design with both hardware and software. We have seen that the primary interaction between hardware and software comes at the microprocessor bus boundary and in the contents of programmable registers and memory. Memory is probably the most important place where hardware and software meet, and yet it is one of the most misunderstood. For example, hardware engineers are accustomed to loading memory data into a simulation model at the start of simulation using commands such as $readmemh. Once the simulation starts up and reset is completed, the memory data is ready to be read from or written to. Software engineers have a different concept. The most common way to write a test program for an embedded system is to compile it into an executable file format such as ELF, go to a lab, power-on a board with the microprocessor, connect a software debugger using a JTAG cable connected between the board and the parallel port of a PC and use the debugger to download the ELF file into memory on the board and start running the program. Of course, the JTAG connection uses the microprocessor to perform a long series of memory writes on the CPU bus to get the data into memory instead of just pre-loading the data like in the simulation case. I'm constantly reminded of these differences between hardware and software.

Not long ago, I was talking to an engineer about these concepts of memory and explaining the connection between the operations of a software debugger and the simulated hardware design. All at once, a light seemed to go on and he asked if he could try an experiment. I just got out of the way and gave him the keyboard. He asked where in the CPU memory map was an SRAM. I told him there is SRAM at address 0x10000000. He used the software debugger to write one word into the memory at this address. He then proceeded to stop the logic simulator by hitting Ctrl+c and asked for the Verilog hierarchy path to one of the 16-bit wide SRAM models. I told it to him and proceeded to issue a command to dump the contents of the SRAM into a file. He then went to another xterm and opened the file with the trusty vi editor and amazingly enough the data value he had written into this memory was there! He seemed almost in shock that he now understood the connection between the software debugger and the simulation memory models. The connection between the software debugger and the memory in the logic simulation is shown in Figure 2-17.

```
From the software debugger command line write a new data value to
address 0:

arm9sd: ex 0,1
0x00000000: 0xea000012                                    "...."
arm9sd: let 0 = 0x12345678
arm9sd: ex 0,1
0x00000000: 0x12345678                                    "xV4."
arm9sd:

From the Verilog command prompt dump the new memory contents to a file:

C1> $dumpmemh("mem.dat",top.mem.Rom);

xp8:4 % more mem.dat
@000000 12345678 ea0005ef ea0005f6 ea00062a ea000621 eafffffe ea000611 ea000602
@000008 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
@000010 00000000 00000000 00000000 00000000 e3a00000 e3a01000 e3a0dd84 e92d0003

Sure enough, address 0 has the data value written from the debugger.
```

Figure 2-17: The memory connection

As we move into the details of hardware and software co-verification, let's review the key points. Hardware simulation always has the notion of a simulation timestamp to mark the place in time when events occur. This is the key reference point that is used to identify when things happen. Software generates bus transactions on the CPU bus to access memory and control registers. Hardware peripherals generate interrupts to request help or service from the software. These are the important areas where the hardware world meets the software world.

Learning about the relationship between hardware and software engineering is like the old joke about a mechanical engineer, electrical engineer, and software engineer.

A mechanical engineer, an electrical engineer and a software engineer are in a car that breaks down. The mechanical engineer says: "Maybe it's a stuck valve in the engine." The electrical engineer says: "Maybe the battery is dead or a fuse is blown." The software engineer says: "I know. Let's all get out of the car and get back in again and see if the problem goes away."

Understanding the tools and techniques of hardware and software engineers and the concepts of embedded systems and microprocessors is critical to verifying designs made up of both hardware and software.

CHAPTER 3

SoC Verification Topics for the ARM Architecture

Understanding many of the concepts of hardware/software co-verification is best done by example. Understanding a specific microprocessor architecture is useful for discussion of co-verification topics. In today's SoC projects, the ARM architecture holds the largest market share for 16/32 bit processors. The goal of this chapter is not to teach every detail of the architecture, but to provide some background on ARM and to teach the things that are directly related to verifying a hardware design with software running on an ARM microprocessor. Engineers currently involved in ARM projects will find the material directly applicable to their project, while those not yet involved in an ARM project can use the material to become familiar with the architecture and should have little trouble relating the concepts to other processor architectures.

ARM Background

ARM started as a branch of Acorn Computer in Cambridge, England, with the formation of a joint venture between Acorn, Apple and VLSI Technology. A team of twelve employees produced the design of the first ARM microprocessor between 1983 and 1985. The ARM was the first RISC processor developed for commercial use.

In 1990 a new company, Advanced RISC Machines Ltd., spun out from Acorn, to focus on creating microprocessors. The ARM acronym originally stood for Acorn RISC Machine, but was changed to Advanced RISC Machine to reflect the spin out from Acorn. Today the ARM name is not an acronym, just ARM Ltd. The company is still headquartered in Cambridge and employs about 800 people worldwide. It operates design centers in Sheffield, Maidenhead, around Europe and the United States, and sales offices throughout the world. ARM currently holds a 75–80% market share in 32-bit microprocessors.

Besides pioneering the first RISC microprocessor for commercial applications, ARM has pioneered a new business model in the electronics industry. The company makes no chips. Almost all of the company revenue is realized by licensing its technology (primarily microprocessor designs) to semiconductor partners. As of this writing, ARM counted 118 semiconductor partners utilizing ARM technology and the number grows every quarter. These partners use ARM intellectual property (IP) to develop chips and products for the electronics market. ARM revenue comes from three primary sources:

1. License fees for microprocessors and other IP blocks.
2. Per-chip royalties on shipments of chips using ARM IP.
3. Tools and boards to support development and debugging of ARM applications (development systems).

ARM is currently the largest and most successful IP company. It offers customers time-to-market advantages by offloading some of the design work. Due to the global acceptance of the ARM standard, users are never locked into a single semiconductor partner, as many sources of ARM technology are available. The large network of partners also makes the ARM architecture popular with engineers, as there are many supporting tools to choose from. As we will see, co-verification is no exception. ARM is number one in terms of partner support for modeling, co-design, and co-verification tools.

ARM Architecture

All ARM processors use RISC (reduced instruction set computer) principles. The other type of microprocessor design is called CISC (complex instruction set computer). The goals of ARM RISC processors are simplicity, high instruction throughput, excellent real-time response, and a small, low-power design. ARM has historically been known as the leader in low-power design. All ARM processors are static designs. This means the clock can be stopped at any time to save power. These low-power features make the architecture popular in portable, battery-operated applications.

RISC design started in IBM when engineers noticed that many instructions in the CISC instruction set were never or rarely used. In the early 1980s engineers in IBM and HP, as well as researchers in Stanford, began to develop microprocessors that provided higher performance by simplifying the instruction set. The availability of an operating system (UNIX) written in C that could be easily ported and compiled for RISC architectures made the combination of UNIX and RISC the standard for engineering workstations. This is still true today, although it may be changing with the widespread adoption of Linux on Intel PC hardware as the standard engineering workstation. Today, RISC processors dominate the world of embedded systems. RISC shipments for various architectures are shown in Figure 3-1. ARM has continued to gain market share in the last couple of years.

Merchant RISC Microprocessor Shipments (1000s)

	thru 1994	1995	1996	1997	1998	1999	2000	2001
ARM/StrongARM	2,170	2,100	4,200	9,800	50,400	152,000	414,000	402,000
MIPS	3,254	5,500	19,200	48,000	53,200	57,000	62,800	62,000
Hitachi SH	2,800	14,000	18,300	23,800	2,600	33,000	50,000	45,000
PowerPC	2,090	3,300	4,300	3,800	6,800	8,300	18,800	23,000
Total	30,499	33,830	58,480	98,220	149,080	262,820	556,800	538,860

Source: Andrew Allison

Figure 3-1: RISC shipments from 1994 to 2001

ARM Architectures, Families, and CPU Cores

RISC processors employ fixed length instructions. The ARM instruction set is defined by a particular version of the ARM architecture. Each specific ARM CPU core implements a specific architecture version. The ARM architecture dates all the way back to version 1 that corresponds to the first design developed at Acorn. Here are the primary architecture versions used in recent processor families:

- ARMv4T
- ARMv5T
- ARMv5TE
- ARMv5TEJ
- ARMv6

There are four families of ARM processors currently being used in new SoC designs:

- ARM7 family uses ARMv4T
- ARM9 family uses ARMv4T, ARMv5TE and the ARMv5TEJ
- ARM10 family uses ARMv5T, ARMv5TE and the ARMv5TEJ
- ARM11 family uses ARMv6

The ARM7 family uses a von-Neumann architecture (a single bus for instruction and data) while the ARM9 family uses a Harvard architecture (different buses for instruction and data).

ARM cores are available in both hard macro form or in soft form. A hard macro is a design that has been taken all the way to silicon implementation. It is provided to the user as a layout object to be included with the rest of the design at the physical design stage. Hard cores provide the highest performance and smallest die area because they have been fully optimized by ARM. Hard cores are not limited by any design restrictions of the user's front-end chip design flow. The drawback of a hard core is the portability between different silicon processes. Since the hard core is already committed to a specific semiconductor process, a user is forced to this process instead of a different one that may be more suitable for the rest of the chip.

Soft cores are delivered in Verilog RTL format. The user is responsible for performing synthesis and physical implementation of the CPU. Soft cores offer more flexibility for the user, related to the silicon process, but cannot provide performance as good as the hard core. Over time, ARM has shifted more and more into soft cores. Today's front-end EDA tools are capable of mixed language (Verilog and VHDL) designs and advanced semiconductor processes reduce the need for total optimization. We will see in future chapters that the implementation used for the ARM core is important for determining what kind of models are available for co-verification.

There are many ARM processors that are actively being used in new designs today. Following is a list of the most common variants and the highlights of each:

- **ARM7TDMI**: Developed in 1995, it is still one of the most popular today. Uses a three-stage pipeline and the v4T architecture to provide good performance and very low power and small size. It's available as both a hard and soft macrocell.
- **ARM720T**: Includes the ARM7TDMI core plus an 8k unified (instruction and data) cache, MMU, and write buffer and can run operating systems such as Windows CE and Symbian OS. Available as a hard macrocell.
- **ARM9TDMI**: An upgrade to the ARM7TDMI that uses the same v4T architecture, but moves to a five-stage pipeline and Harvard architecture. Available as a hard macrocell.
- **ARM940T**: Includes the ARM9TDMI core plus dual 4k caches (separate instruction and data) and an MPU that supports most real-time operating systems. Available as a hard macrocell.
- **ARM920T / ARM922T**: Includes the ARM9TDMI core plus dual 16k caches for ARM920T and dual 8k caches for ARM922T. Other than cache size they are identical. Both support operating systems like Windows CE, Symbian OS, PalmOS and Linux. Available as hard macrocells.
- **ARM9E-S**: Uses a five-stage pipeline to support the v5TE architecture that includes an enhanced multiplier for improved DSP performance. Available as a soft macrocell. Represents the first shift to synthesizable CPU cores.
- **ARM966E-S**: Includes an ARM9E-S CPU and instruction and data tightly coupled memory (TCM). Targets "hard" real-time embedded applications, no caches or MPU/MMU. Available as a soft macrocell.
- **ARM946E-S**: Adds user configurable instruction and data caches and MPU to the ARM966E-S. Supports popular real-time operating systems.

- **ARM9EJ-S**: Adds Jazelle Java technology to the ARM9E-S core to support the v5TEJ architecture. Available as a soft macrocell.
- **ARM926EJ-S**: Includes the ARM9EJ-S core and instruction cache, data cache and MMU. Supports operating systems such as Windows CE, Symbian OS, PalmOS and Linux. Available as a soft macrocell. The most popular ARM9 core.
- **ARM1020E / ARM1022E**: Combines the ARM10E integer core with six-stage pipeline with DSP extensions and supports the v5TE architecture with instruction and data caches and MMU. Supports operating systems such as Windows CE, Symbian OS, PalmOS and Linux. First design to use 64-bit internal data paths and 64-bit external data buses. ARM1020E and ARM1022E are identical except for cache sizes. ARM1020E is 32k/32k and ARM1022E is 16k/16k. Available as a hard macrocell.
- **ARM1026EJ-S**: Combines the ARM10EJ integer core with six-stage pipeline, DSP instructions, Jazelle technology and supports the v5TEJ architecture with instruction and data caches, MMU and instruction and data TCM. Fully synthesizable design that allows cache sizes and external bus widths to be configured by the user. Capable of running all operating systems. Available as a soft macrocell.
- **ARM1136J-S**: Represents the first implementation of the v6 architecture (some 80+ new instructions) with eight-stage pipeline. The highest performance processor with multiple 64-bit external busses. Design is available as a soft macrocell.

Because of the unique semiconductor partner relationships maintained by ARM, the cores that are available primarily in soft form may also be available as hard macrocells from specific semiconductor partners. This allows the partners to provide optimized designs for specific processes and offer them directly to users. For example, most projects use the ARM946E-S as Verilog RTL, but some may use a foundry hardened ARM946E.

A brief explanation of some of the naming conventions used by ARM is useful. One of the most common questions is "What does 7TDMI stand for? Does it mean anything?" Actually, it does, and here is the meaning:

7 = ARM7 family.

T = Thumb extensions specified in the v4T architecture.

D = debug hardware.

M = enhanced multiplier.

I = embedded ICE for JTAG debugger connections.

Other commonly used conventions are the E for the further enhanced multiplier to improve DSP performance (v5TE architecture) and the J to specify the Jazelle technology that allows the direct execution of Java bytecodes. S indicates a synthesizable or soft core.

Thumb Instruction Set

All of the above CPU cores are 32-bit architectures. 32-bit architectures provide the best performance to operate on 32-bit data. The registers are 32 bits wide as are the datapaths inside the CPU. One of the drawbacks of a 32-bit RISC architecture when compared to a CISC architecture is the amount of memory required to hold the software instructions. The measure of how much memory is required in the embedded system to hold instructions is called the ***code density***. In embedded systems, there is often a constraint to minimize memory size. This is especially true for on-chip memory. On-chip memory such as cache usually takes up more space on the chip than the CPU and the other random logic of the SoC. To address memory sensitive applications, the ARM architecture includes a mode that allows it to run 16-bit instructions called *Thumb*. The Thumb instruction set is a group of 16-bit instructions that are a subset of the most popular 32-bit instructions. Special purpose hardware is used to "decompress" the 16-bit instructions into 32-bit instructions. Of course, the extra step required to handle 16-bit instructions incurs some performance penalty. Each application has the freedom to trade off performance for memory size. The use of Thumb instructions may shrink the required memory by 1/3 and decrease performance by about 1/3. ARM uses a very innovative approach that allows the CPU to change dynamically between 32-bit instructions and 16-bit instructions. This gives the software engineer the ability to run performance critical software using 32-bit instructions and less critical software in 16-bit mode to save memory. The Thumb instruction set was introduced in the ARMv4T architecture and first made popular in the ARM7TDMI.

ARM microprocessors use a pipeline to increase instruction throughput. The use of a pipeline allows multiple operations to be done in parallel. As the ARM architecture has advanced the length of the pipeline has increased from three stages (ARM7) to five stages (ARM9) to six stages (ARM10) to eight stages (ARM11). A diagram of the ARM7 pipeline and how it is used to increase performance is shown in Figure 3-2.

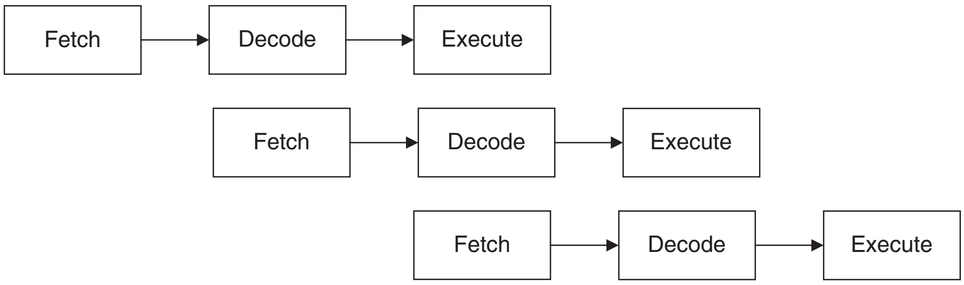


Figure 3-2: ARM7 three-stage pipeline

Programming Model

The ARM programming model consists of 16 general-purpose 32-bit registers known as R0 through R15. The architecture allows for two operating modes, user and supervisor. The register definitions are slightly different depending on the mode of operation. The CPU will start operation in supervisor mode and can be switched to user mode. Applications will run in user mode and operating system or other system software will normally operate in supervisor mode.

Engineers focused on verification of ARM SoC designs are probably not extremely interested in all of the details of the architecture, the instruction set and pipeline. However, there are a few things that are important to know. The first thing is to be able to identify the program counter (PC) as R15. Most simulation models list the registers as R0 through R15 with no special distinction of the program counter. Being able to find the PC in the simulation waveforms is useful to figure out where the software is executing. The other register that is sometimes worth looking at is R14, the link register (LR). This register holds the return address that will become the program counter when the software leaves the current subroutine. This can also help identify where the software is executing. R13 is commonly used as the stack pointer (SP). The CPSR register stands for Current Program Status Register and holds the condition code bits, interrupt flags and operating mode bits. Figure 3-3 shows the basic ARM registers. There is a lot to learn to understand the ARM register set in the different modes, including both ARM state and Thumb state, but familiarity with the basic registers is useful for verification.

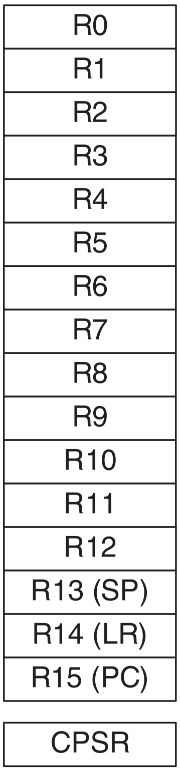


Figure 3-3: Basic ARM register set (ARM state, system/user mode)

Instruction Set

Details of the instruction set are not extremely important for verification. However, some instructions that have a direct relationship to hardware operation are useful to understand for the purposes of verification.

Data Transfer Instructions

Memory loads and stores are instructions that move data from memory to registers (load) and from registers to memory (store). These instructions create a read (load) or a write (store) on the CPU data bus. The instruction set allows data to be accessed in different sizes. Although some ARM10 and ARM11 cores have 64-bit buses, we will focus on the standard 32-bit buses here. One interesting note is that early versions of the ARM architecture did not support half-word loads/stores or signed byte loads and stores, only word and unsigned byte accesses. This made things interesting for C programmers. Recent versions have added support for half-words as well as signed bytes. Some examples of load and store instructions are shown in Figure 3-4.

```
STR r0,[r1],#4

LDR  pc,[r4,r6, LSL #2]     ; this is actually a function call

LDRH    r0, [r0]

STRH    r4, [r2], #3

STRB    r4, [r2]

LDRBT   r3, [r6], #0
```

Figure 3-4: Examples of Load and Store Instructions

```
MRC     p15, 0, r3, c1, c0, 0   ; read from coprocessor

MCREQ   p15, 0, r12, c7, c14, 1  ; write based on condition

MCR     p15, 0, r0, c9, c1, 1   ; write to coprocessor
```

Figure 3-5: Examples of coprocessor Instructions

Coprocessor Instructions

The ARM architecture allows hardware coprocessors to be connected to the processor and provides special coprocessor instructions to access this hardware. The coprocessors are also accessed via 32-bit load and store instructions. Up to 16 coprocessors are supported. Some examples of coprocessor instructions are shown in Figure 3-5. Not all processors support all coprocessor instruction types. The most common are MCR and MRC. Normally, coprocessor access is also restricted to privileged mode to keep application software from changing settings.

Some CPUs, like the ARM7TDMI, provide a set of pins that serve as the coprocessor interface. Users are free to connect special hardware to this interface to provide a dedicated, direct connection between the CPU and the coprocessor without sharing the memory bus. The most common use of coprocessors is coprocessor 15. ARM has designated coprocessor 15 to be used to control and configure caches, memory management units, memory protection units, buffers, and any other configurable hardware related to the CPU. Recent designs such as the ARM926EJ-S use coprocessor 15 extensively for configuration and control. Some of the things provided by coprocessor 15 in the ARM926EJ-S are:

- CPU identification
- Details of cache such as size and other cache attributes
- Ability to set cache attributes
- Ability to invalidate entire cache or specific cache lines
- Ability to lock specific data in the cache

- Presence of tightly coupled memory (TCM), its size, and the ability to enable/disable TCM
- Ability to configure TCM address ranges
- Configuration of address translation, protection, and other MMU functions

Exceptions and Interrupts

Exceptions are events that break the normal execution of CPU instructions and cause it to do something else. There are generally many things that fall under the category of exceptions. Interrupts are one type of exception and are defined as events that originate outside the CPU and signal to the CPU to take an exception. Interrupts play a key role in all embedded systems. While software normally communicates with hardware by writing and reading registers and memory, hardware initiates communication with software using interrupts.

Learning about exceptions and exception vectors is also very useful for ARM SoC verification. ARM processors start execution after reset by fetching the first instruction from address 0 (the reset vector). Reset is an interrupt since an external signal tells the CPU to start over. Some processors have a provision to use an alternate address for the reset vector, called the high-reset vector, located at address `0xffff0000`. An example of this is the ARM926EJ-S. If the signal **VINITHI** is low during reset then the exception vectors start at address 0. If **VINITHI** is high during reset then the exception vectors start at `0xffff0000`. This alternative location enables the use of a different (usually faster) type of memory to handle exceptions. The alternate exception address is also useful for dual processor designs. The addresses for the exception vectors are given in Figure 3-6. If the high exception vectors are used the offsets are not from 0, but from `0xffff0000` instead.

Reset	0x0
Undefined Instruction	0x4
Software Interrupt	0x8
Instruction Fetch Abort	0xc
Data Abort	0x10
Interrupt (nIRQ)	0x18
Fast Interrupt (nFIQ)	0x1c

Figure 3-6: Exception addresses

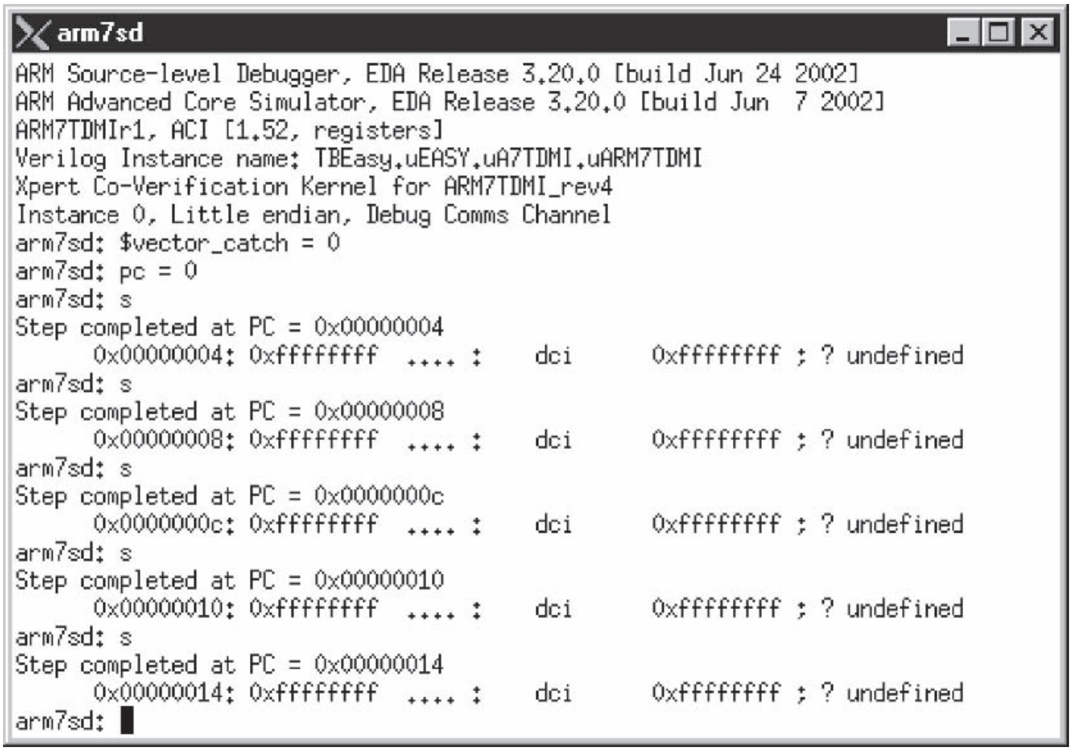
The priority for the CPU to handle exceptions is:

1. Reset
2. Data Abort
3. Fast Interrupt
4. Normal Interrupt
5. Instruction Fetch Abort
6. Software Interrupt

All ARM CPUs use two interrupt signals on the bus, **nIRQ** and **nFIQ**. **nIRQ** is the normal interrupt request and **nFIQ** is the fast interrupt request. The bus signals for these two interrupts are active low signals, so driving the signal low indicates an interrupt.

Another pair of exceptions that are important for co-verification are the abort exceptions. There are two kinds of aborts, instruction prefetch and data aborts. If the CPU tries to access memory that cannot be handled by the memory system, an abort will be signaled to the CPU. The **ABORT** signal is used for some ARM CPUs such as the ARM7TDMI. For CPUs using the AHB interface (described below), **HRESP** is used to signal abort.

The remaining two exceptions in the exception table are the undefined instruction and software interrupt (SWI). These tend to play less of a role in co-verification. The software interrupt is generated by a software instruction and has no special connection to the hardware design. Seemingly the Undefined Instruction exception would be important if the memory system provided bad instruction values to the CPU in a situation where there was a bug in the hardware design, but more often than not whatever bad data is provided to the CPU is usually interpreted as some instruction and the CPU continues to run with no clear indication that disaster has already occurred. For example, if the memory system is somehow not working correctly and provides an instruction of `0xffffffff` the CPU still runs forward incrementing the program counter by 4 forever. Figure 3-7 shows a screen shot of an ARM debugger running a program where all data is `0xffffffff` and no exception occurs.



```
ARM Source-level Debugger, EDA Release 3.20.0 [build Jun 24 2002]
ARM Advanced Core Simulator, EDA Release 3.20.0 [build Jun  7 2002]
ARM7TDMIr1, ACI [1.52, registers]
Verilog Instance name: TBEasy.uEASY.uA7TDMI.uARM7TDMI
Xpert Co-Verification Kernel for ARM7TDMI_rev4
Instance 0, Little endian, Debug Comms Channel
arm7sd: $vector_catch = 0
arm7sd: pc = 0
arm7sd: s
Step completed at PC = 0x00000004
      0x00000004: 0xffffffff  .... :    dci      0xffffffff ; ? undefined
arm7sd: s
Step completed at PC = 0x00000008
      0x00000008: 0xffffffff  .... :    dci      0xffffffff ; ? undefined
arm7sd: s
Step completed at PC = 0x0000000c
      0x0000000c: 0xffffffff  .... :    dci      0xffffffff ; ? undefined
arm7sd: s
Step completed at PC = 0x00000010
      0x00000010: 0xffffffff  .... :    dci      0xffffffff ; ? undefined
arm7sd: s
Step completed at PC = 0x00000014
      0x00000014: 0xffffffff  .... :    dci      0xffffffff ; ? undefined
arm7sd: █
```

Figure 3-7: Running a meaningless program doesn't generate any exception

Memory Layout and Byte Order

Different microprocessors use different ways to order the bytes of data on the bus. The computer term **endian** is used to describe the order of bytes of a multibyte data value. There are two ways to order the bytes:

- Big-endian means the most significant byte has the lowest address.
- Little-endian means the most significant byte has the highest address.

Historically, Intel microprocessors have used little-endian byte order and RISC microprocessors have used big-endian. The ARM microprocessors discussed in this chapter allow the user to choose the byte order. Although it is probably not a word, the byte order chosen is called *endianness*. Figure 3-8 and 3-9 show how to assign addresses for a 32-bit word for each byte order.

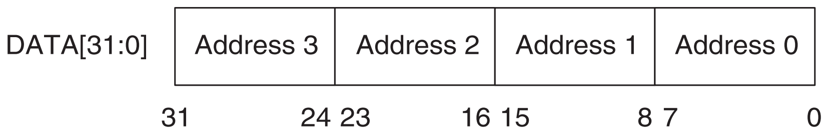


Figure 3-8: Little endian byte order

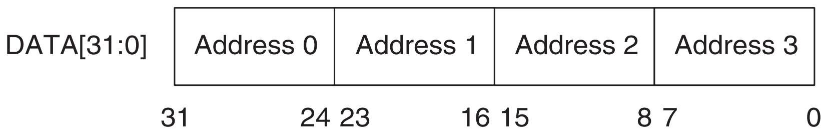


Figure 3-9: Big endian byte order

Endianness is one of the most misunderstood topics in co-verification. One misunderstood fact that is that for ARM cores with a 32-bit bus, as long as the CPU is transferring word values on the bus, endianness does not matter. When data transfers are a full word, the data on the bus is identical for both big-endian and little-endian byte order. Endianness becomes important when data transfers on the bus are for 1 or 2 bytes. In these cases, both master and slave must know which bytes are valid and which are not. Incorrect interpretation of the data by either master or slave leads to certain data corruption.

The second misunderstanding is that how a bus is declared defines endianness. In Figure 3-8 the data bus is declared as [31:0], but this does not say anything about byte order.

One of the first questions I always ask for a new design is, "What is the byte order of the design, big or little endian?" Working in co-verification requires me to talk with application engineers about projects using co-verification. When I ask about endianness I may get a blank stare, or maybe they figure they have a 50-50 chance so they just take a guess. One response was: "The only thing I know is that the bus is from "31 downto 0" which, to me, is little endian, correct?" Understanding byte order is important when trying to find problems related to incorrect data transferred on the bus.

ARM Bus Interface Protocols

The bus protocol used by the CPU is an important aspect of co-verification since this is the main communication between the CPU, memory, and other custom hardware. ARM processors use different bus protocols depending on when the core was designed. This section will cover the protocol used for the ARM7TDMI and the CPU bus protocols covered by the advanced microcontroller bus architecture (AMBA) specification. The ARM7TDMI utilizes an older bus protocol but it is still widely used today without the AMBA specification so it is worth learning. The goal of this section is not to teach every detail about the protocols, but to give a good overview such that engineers dealing with verification will have the necessary skills to understand what is happening on the bus and how the bus activity relates to software execution. For detailed information on how the bus protocols operate, the appropriate specification should be consulted.

ARM7TDMI Bus Protocol

The ARM7TDMI uses a single 32-bit address and 32-bit data bus to access memory. The protocol has two characteristics that make it non-trivial to understand, bus pipelining and dual edge clocking. The core contains a very large number of signals that at first seem overwhelming, but most of these are not important for normal operation and certainly not needed in the context of co-verification.

The primary signals used by the ARM7TDMI bus protocol are described below. A good understanding of this subset is sufficient for most situations. A diagram of the ARM7TDMI signals is shown in Figure 3-10. A complete description of all signals and functionality is available from ARM in the ARM7TDMI Technical Reference Manual (TRM).

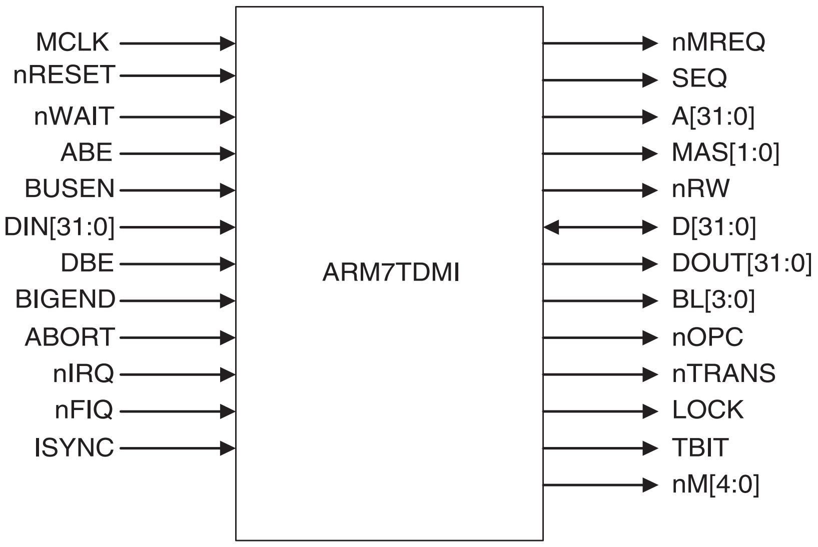


Figure 3-10: ARM7TDMI signals

MCLK is the main input clock to the core. Since the ARM7TDMI is a fully static design, the clock does not have to be regular. In fact, stopping the clock is a commonly used way to insert wait states on memory accesses.

nRESET is the processor reset. Asserting it low for at least 2 clocks and then releasing it will cause execution to start from the reset vector (address 0). While **nRESET** is low the CPU will perform idle transactions, incrementing the address on each transaction to indicate it is in the reset state.

nWAIT provides another way to insert wait states on the bus. The core is clocked by the logical AND of **MCLK** and **nWAIT**. **nWAIT** must change only when **MCLK** is low.

nMREQ is low when the processor starts to perform a memory access. **SEQ** and **nMREQ** together define the type of bus cycle being performed. The cycle types are shown in Figure 3-11.

SEQ indicates a memory access with an address related to the previous address. Depending on the CPU mode, the new address is an increment of 2 or 4 bytes from the previous address.

A[31:0] is the 32-bit address bus.

ABE must be driven high to enable the address bus.

MAS[1:0] indicates the size of a data transfer on the 32-bit data bus. The values are 00 for byte, 01 for halfword (16-bit), and 10 or word (32-bit). The value of 11 is not used.

nRW indicates a read (low) or a write (high).

BUSEN selects either the bi-directional data bus (low) or the uni-directional data busses (high).

D[31:0] is the 32-bit bi-directional data bus used when **BUSEN** is low.

DIN[31:0] is the 32-bit input data bus used when **BUSEN** is high.

DOUT[31:0] is the output data bus used when **BUSEN** is high.

DBE must be driven high to enable the data bus.

BIGEND selects the byte order, high for big endian and low or little endian.

ABORT indicates that a memory access is not allowed. It will cause the processor to take the data abort exception (vector address 0x10).

BL[3:0] provides a way to connect to memory systems that are less than 32-bit. **BL** is used to latch data onto the 32-bit data bus. This signal is rarely used, since most designs use external latches to interface to memories that are less than 32-bits wide.

nOPC is low to indicate an instruction fetch and high to indicate a data access.

nTRANS is low to indicate User Mode and high to indicate Privileged Mode.

LOCK is high to indicate an atomic read/write operation such as the SWP or SWPB instructions.

TBIT is high when operating in ARM state and low when operating in Thumb state.

nM[4:0] indicates the processor mode. These bits mirror the least significant 5 bits of the CPSR and are intended for debugging only. Values are listed in Figure 3-12.

nIRQ is the normal interrupt (lower priority) request. A low value causes the processor to take the **nIRQ** exception (vector address 0x18).

nFIQ is the fast interrupt (high priority) request. A low value causes the processor to take the **nFIQ** exception (vector address 0x1c).

ISYNC high tells the processor to sample **nIRQ** and **nFIQ** with the rising edge of **MCLK** (synchronous interrupts).

Cycle Type	nMREQ	SEQ
Non-Sequential Cycle	0	0
Sequential Cycle	0	1
Internal Cycle	1	0
Coprocessor Cycle	1	1

Figure 3-11: ARM7TDMI bus cycle types

Mode	nM[4:0]
System	00000
Unidentified	00100
Abort	01000
Supervisor	01100
IRQ	01101
FIQ	01110
User	01111

Figure 3-12: ARM7TDMI processor mode values

One of the difficulties with the ARM7TDMI memory bus is figuring out when the various signals are valid on the bus. Since the bus uses pipelining and both edges of **MCLK** it is not always easy. In the normal pipelined mode (**APE**=1 and **ALE**=1), the cycle type indicators, **nMREQ** and **SEQ**, are sampled on the rising edge of **MCLK**. The address related signals are sampled on the next falling edge of the clock, and assuming no wait states (via **nWAIT**) the data is sampled on the next falling edge of the clock. A waveform of the ARM7TDMI memory bus after reset is shown in Figure 3-13.

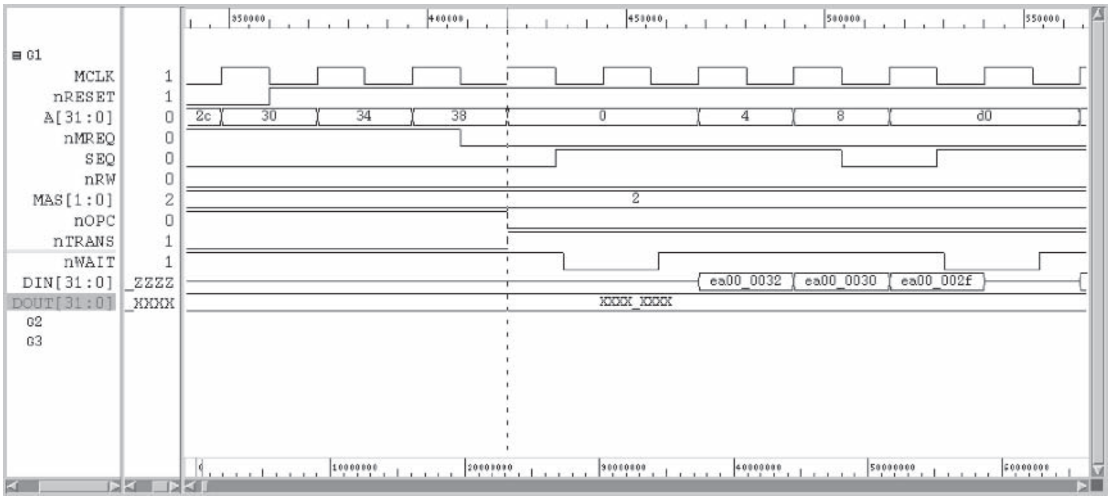


Figure 3-13: ARM7TDMI waveform after reset

AMBA Specification

The evolution of the ARM7TDMI bus interface led to the creation of the advanced microcontroller bus architecture (AMBA). Since ARM is focused on selling intel-lectual property that includes both microprocessors and other peripherals, it became useful to propagate a standard bus structure that is open for engineers to use in designing with ARM IP. Issues such as design reuse and modular system design are important to ARM. AMBA enables even more IP that works together with ARM IP and provides a common set of protocols for bus infrastructure that has become the de facto standard used in SoC design today. AMBA has become so ubiquitous that even designs not based on ARM microprocessors are using AMBA.

The primary specification in use today is version 2.0 of AMBA. This specification defines three busses: two high-speed busses and one peripheral bus. The high-speed busses are advanced system bus (ASB) and advanced high-performance bus (AHB). The peripheral bus is advanced peripheral bus (APB). It's pure speculation, but the term "advanced" in all three specifications may stem from one of the meanings of the ARM acronym, Advanced Risk Machines, or it may have been chosen just to form a nice acronym. Either way, none of the busses are particularly "advanced" as bus protocols go, but nevertheless they are all useful for most classes of SoC designs.

The typical design using AMBA is a diagram that is seen in the AMBA specification and over and over again throughout ARM literature. It contains a single ARM processor and important hardware functions such as a memory controller for fast memory and DMA attached to the high-speed bus (ASB or AHB) and a bridge to the slower peripheral bus (ABP) that connects slower peripherals that are optimized for power instead of speed. A block diagram of a typical AMBA system is shown in Figure 3-14.

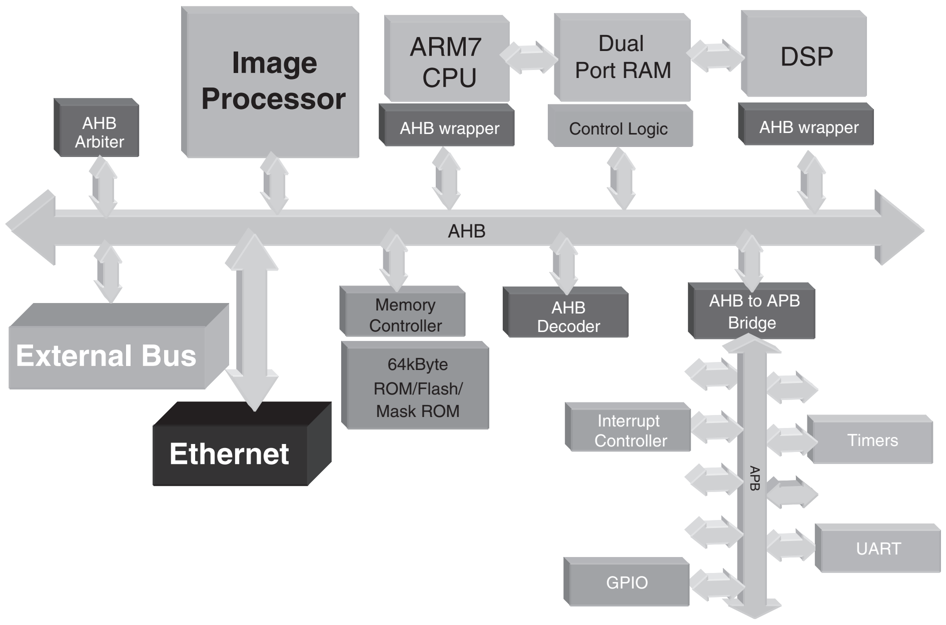


Figure 3-14: Example "typical" AMBA system

Introduction to AMBA Protocols

This section provides a short introduction to the three AMBA protocols. The next section provides more details of the most common protocol used for ARM microprocessors (AHB) since this is the protocol that is most important for co-verification.

AMBA ASB

ASB is the first-generation system bus that evolved from the ARM7TDMI bus protocol. It supports pipelining, bursts, and multiple bus masters. The four bus agents in ASB are:

- *Arbiter*: Implements a simple request/grant structure to support multiple bus masters.
- *Decoder*: Centralized address decoder to determine which slave is responsible for servicing a bus transaction.
- *Master*: Initiates reads and writes on the bus.
- *Slave*: Responds to master initiated reads and writes.

As we will see in the next section there are two primary drawbacks of ASB that led to the development of AHB; double-edged clocking and the bi-directional data bus. ASB uses both edges of the bus clock, which makes it not only more difficult to understand and design with, but also imposed increased complexity for most ASIC design flows and synthesis tools that are based on using only the rising edge of clock. Similarly, the bi-directional data bus (and tri-state signals in general) is not possible under many ASIC design rules. Even if tri-state signals are possible, bus turnaround times always cause some performance penalty. AMBA AHB corrects these issues.

For highest performance, typical designs based on ASB use an ARM processor with a write-back cache. A write-back cache is a cache algorithm that allows data to be written into cache without updating the system memory. Since ASB does not have any provisions for maintaining cache coherency of multiple caching bus masters only one processor can be used on ASB.

AMBA AHB

AHB is the second-generation system bus that evolved from ASB and improves upon its bus clocking and bus handover. Like ASB, AHB also supports pipelining, bursts, and multiple-master operation. It uses the same four bus agents: Arbiter, Decoder, Master, and Slave. All bus clocking is single-edge and uses the rising edge of the bus clock to ease the synthesis flow. All AHB signals are uni-directional, which guarantees implementation on all ASIC design processes and enables single-cycle bus master handoff. The data bus width of AHB is specified as 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide, but practically it is implemented as 32, 64, or 128 bits wide. AHB also supports split transfers in which the address transfer is separated from the data transfer. Slaves can use a split response when it will take many cycles to complete the bus transaction. Overall bus utilization is improved when slower slaves do not stall the bus for long periods of time.

AMBA APB

APB is a peripheral bus that is optimized for low power and interfacing to slower peripherals. APB is normally encapsulated behind a bus bridge that connects APB to either ASB or AHB. The bridge is a slave on ASB or AHB. Like AHB, APB has been improved to use only the rising edge of the bus clock. The APB protocol does not support any pipelining (address and control signals remain on the bus for the entire transfer).

AMBA 3.0 and AXI

At the end of 2002, ARM announced AMBA 3.0 and a large list of partners for this new version of AHB. The specification was later released under the name Advanced Extensible Interface (AXI) and it is the next generation of high-performance AMBA. AXI is also an open specification that is easily downloadable from the ARM website. Some of the features of AXI are:

- Separate address/control and data phases
- Support for unaligned data transfers using byte strobes
- Burst-based transactions with only start address broadcast

- Separate read and write data channels to enable low cost direct memory access (DMA)
- Issuing of multiple outstanding addresses
- Out-of-order transaction completion

AXI has recently been incorporated into new ARM11 CPU cores such as the ARM1156T2-S and the ARM1176JZ-S. These cores were announced at Microprocessor Forum in October 2003 and will be available for use in mid-2004. AXI is the next evolution of the ARM system bus and will likely be used on all new high-performance designs.

Summary of ARM CPU Bus Interfaces

Below is a summary of the bus protocols used by the popular ARM CPU cores. When referencing a CPU technical reference manual (TRM), sometimes the signal names are not exactly the same as those listed in the AMBA Specification, but the behavior is usually equivalent. TRMs that were updated after AMBA 2.0 was released may provide information on the equivalence of the AMBA signal names and the CPU signal names. An example of this is the ARM920T Rev 1 TRM. Other TRMs that were released before AMBA 2.0 will not have this information, but the reader will be able to make the connections since the signal names are usually very similar. Each CPU has additional memory bus signals beyond those listed in the AMBA specification, so it is best to first learn the AMBA protocols and then use the TRM as a superset of AMBA to learn the details of a CPU memory bus interface. Most TRMs assume this AMBA knowledge. For example, the ARM926EJ-S Rev 0 TRM has only seven pages of information on the bus interface. These pages are used only to qualify which functionality of AHB is used by the ARM926EJ-S, so it assumes complete knowledge of AHB.

- **ARM7TDMI** uses non-AMBA (ARM7TDMI native) bus interface. It can be converted to AHB using HDL wrappers.
- **ARM720T** started with ASB interface but rev4 has migrated to an AHB interface.

- **ARM9TDMI** uses non-AMBA (ARM9TDMI native) bus interface. Can be converted to AHB using HDL wrappers.
- **ARM940T** uses an ASB interface. It includes extra signals to enable conversion to AHB using an HDL wrapper.
- **ARM920T** and **ARM922T** both use an ASB interface. They include extra signals to enable conversion to AHB using an HDL wrapper.
- **ARM9E-S** uses a non-AMBA Harvard architecture (separate instruction and data buses).
- **ARM966E-S** uses a single AHB interface.
- **ARM946E-S** uses a single AHB interface.
- **ARM9EJ-S** uses a non-AMBA Harvard architecture (separate instruction and data buses).
- **ARM926EJ-S** uses a dual AHB interface, one AHB for instructions and one AHB for data.
- **ARM1020E** and **ARM1022E** both use dual AHB interfaces.
- **ARM1026EJ-S** uses a dual AHB interface.
- **ARM1136J-S** uses four 64-bit AHB interfaces plus a 32-bit AHB interface as a peripheral port.

AHB Tutorial

This section presents a tutorial on AMBA AHB from the view of co-verification. The purpose is not to teach how to implement a design for AHB, but rather to be able to understand the protocol and to be able to diagnose problems in the context of design verification. The goal is not to repeat all of the information that is provided in the AMBA specification, but to summarize the important points and offer some insight into interesting information that has been learned by experience using real ARM designs and various types of ARM models. A diagram of the signals of an AHB master is shown in Figure 3-15.

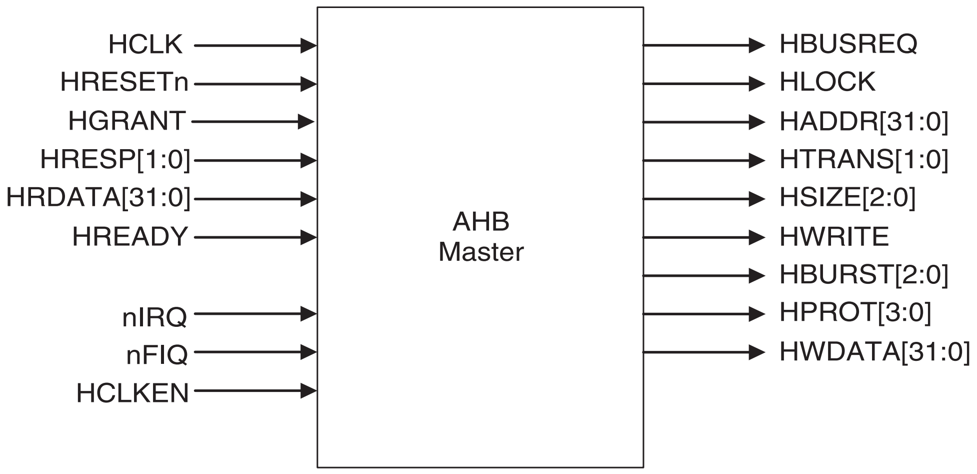


Figure 3-15: AHB master signals

AHB as implemented on ARM7 and ARM9 CPU cores is a single 32-bit address and 32-bit data bus to access memory. All signals, with the exception of the reset, are active high. All activity uses the rising edge of the bus clock and all signals are unidirectional (no tristate values). Interrupt signals are also synchronous. Previous CPU cores such as ARM7TDMI provided the option for asynchronous interrupts via the **ISYNC** signal, but with AHB cores the interrupt signals must synchronized before presenting them to the CPU. The primary AHB signals are described below:

HCLK is the main bus clock. All signals are sampled on and driven from the rising edge of **HCLK**. On most pre-AHB CPU cores such as ARM920T and ARM940T, there were two clocks, one for the CPU core and one for the bus interface. On newer AHB cores such as ARM946E-S and ARM926EJ-S there is only one clock input **CLK** that clocks the CPU core directly. The AHB interfaces can be run at a slower rate by gating **CLK** with **HCLKEN** to produce the AHB clock **HCLK**.

HRESETn is an active low signal used to reset the bus and CPU. When asserted it will cause the CPU to restart from the reset vector. For many ARM cores using AHB, reset is also used to sample configuration signals for endianness and the location of the exception vectors.

HBUSREQ is the bus request signal from a bus master that wants to use the bus. Each bus master will have its own **HBUSREQ**. For ARM CPU cores with multiple AHB interfaces such as ARM926EJ-S, each AHB interface will have its own **HBUSREQ**.

HGRANT is the bus grant from the arbiter to the master that signals the master can use the bus. The master is granted the bus when **HGRANT** and **HREADY** are both high. Failing to also understand that **HREADY** must also be high to indicate a bus grant is one of the most common mistakes engineers make in trying to understand AHB operation.

HLOCK indicates the master is performing a locked access (read-modify-write sequence) and tells the arbiter not to grant the bus to another master during this sequence.

HADDR[31:0] is the 32-bit address bus.

HTRANS[1:0] indicates the current transfer type, either IDLE, BUSY, NONSEQUENTIAL, or SEQUENTIAL. Values for **HTRANS** are shown in Figure 3-18.

HSIZE[2:0] indicates the size of the transfer. Using three bits allows for sizes up to 1024 bits to be transferred, but with the 32-bit data bus only values of 8, 16, and 32-bits are used and only 2 bits of **HSIZE** are used. For example, the ARM946E-S has **HSIZE[2]** permanently tied low. Values for HSIZE are shown in Figure 3- 20.

HWRITE specifies a read (low) or a write (high).

HBURST[2:0] indicates whether or not the transfer is part of a burst. Using 3 bits, there are eight different burst values that are supported. Individual ARM cores will not use all of the burst types. For example, ARM946E-S uses SINGLE, INCR, INCR4, and INCR8. The ARM926EJ-S uses SINGLE, INCR4, INCR8, and WRAP8. All of the values for **HBURST** are listed in Figure 3-19.

HPROT[3:0] is driven by the master to provide more information about the type of transfer. It can be used to implement protection control. Information provided relates to data or instruction fetch (bit 0), privileged or user access (bit 1), bufferable or not bufferable (bit 2), cacheable or not cacheable (bit 3). Bus masters that are not processors will not usually provide this information. Values for **HPROT** are shown in Figure 3-21.

HRDATA[31:0] is the 32-bit read data bus.

HWDATA[31:0] is the 32-bit write data bus.

HREADY is driven by a slave to indicate that its data transfer is finished; the slave can use it to insert wait states. Since the bus is pipelined, **HREADY** does not only pertain to the data phase of a transfer, it also has implications related to the termination of the address phase and arbitration. When the last transfer of a data phase is completed it means the transfer currently in the address phase now moves to the data phase. **HREADY** is also used to determine the bus grant for the next master that will gain access to the bus. **HREADY** can be viewed as advancing the bus pipeline to the next stage. Slaves must both sample **HREADY** on the bus to track the bus protocol and also drive **HREADY** for transfer where it is the slave. ARM recommends calling the **HREADY** input to a slave **HREADY** and the output **HREADYOUT**.

HRESP[1:0] is the slave response indicating if the transfer was completed successfully, had an error, or needs to be completed using the split or retry protocol.

The following signals are not part of a bus master such as an ARM CPU, but are included in a slave or arbiter:

HSEL selects the slave that is responsible for the transaction. It is generated from the decoder based on the master supplied address.

HMASTER[3:0] is generated by the arbiter to indicate which master is using the bus. It is generated with the address phase of a transfer. Up to 16 bus masters are possible with the 4 bits of **HMASTER**.

HMASTLOCK is generated by the arbiter and indicates the master is performing a locked transfer.

HSPLIT[15:0] is generated by a slave that is now ready to complete a previously postponed transfer using the protocol for split transfers. The arbiter will see the **HSPLIT** from the slave and inform the master to retry the transfer.

Configuration at Reset

At the end of reset (rising edge of **HRESETn**) ARM CPU cores sample some inputs to set the operating configuration. Each CPU has different configuration information. Some of the common configuration information determined at reset is the location of the exception vectors, memory organization (endianness), and enabling of tightly coupled memory (TCM). Following are some examples for processors with AHB interfaces.

The ARM946E-S and the ARM926EJ-S sample the signal **VINITHI** to determine the location of the exception vectors. If **VINITHI** is low, the vectors are located at address 0, but if **VINITHI** is high then the vectors are located at address 0xffff0000.

As mentioned previously, endianness is one of the most misunderstood issues in co-verification. To add to the confusion, ARM cores have used different ways to configure endianness. The ARM7TDMI started by using an input pin that was sampled at reset to set the endianness. Later, cores like the ARM946E-S moved to software configuration. The core always starts off in little endian mode and designs that wish to use big endian mode should read only words (since endianness doesn't matter if you read and write words) until such time as the software changes a register bit to specify big endian. There is also an output pin named **BIGENDOUT** that reflects the state of the register bit. Once the switch to big endian mode is made the core can begin to access data using byte and half-word transfers.

The debate between hardware versus software control must not have resulted in a clear winner since the ARM926EJ-S adopted both methods. The ARM926EJ-S uses the same setup as the ARM946E-S except the original value of the register is set by an input signal named **BIGENDINIT** to determine the memory organization. If **BIGENDINIT** is high the CPU will start in big endian mode and if low it will start in little endian mode. After the initial value is set by **BIGENDINIT** software may change to a different byte order by changing the register value. For ARM926EJ-S the output signal that reflects the register value is called **CFGBIGEND**. It appears software wins out in the end since it has the last chance to determine endianness, unless of course hardware decides to become difficult and issue another reset to change endianness yet again. Explaining endianness can definitely cause confusion. After reading this can you remember the difference between the signals **BIGEND**, **BIG-ENDOUT**, **BIGENDINIT**, and **CFGBIGEND**? I use them often and I still must look them up much of the time.

Another configuration option sampled at reset is the enabling of the tightly coupled memory (TCM). TCM details are covered in a future section, but TCM can optionally be enabled for the purpose of booting the CPU from code located in TCM.

Phases of AHB Transfer

There are three phases of the AHB transfer: arbitration, address phase, and data phase. To increase bus utilization, all three of these phases can be performed in parallel. The technique of performing the three phases in parallel is known as bus pipelining. When wait states are inserted into the data phase, it has the side effect of delaying the advancement of the address phase and arbitration phase. Not unlike a CPU pipeline, the bus pipeline is shown in Figure 3-16.

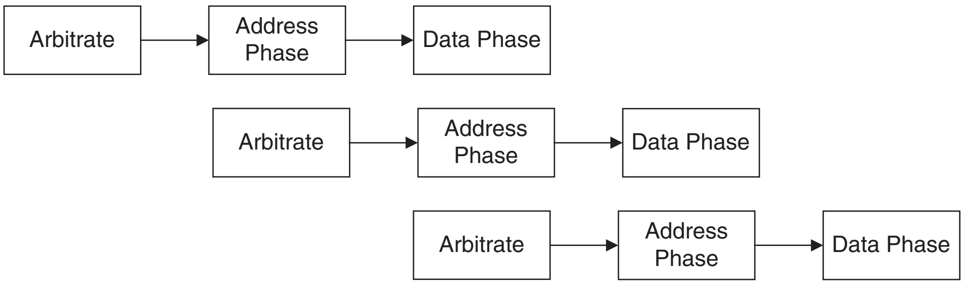


Figure 3-16: Three phases of AHB transaction

AHB Arbitration

The basic arbitration mechanism is straightforward to understand, but I have learned that it is more complicated than it looks. Each bus master has its own request and grant signal connected to the arbiter. Bus masters will assert **HBUSREQ** when they desire to use the bus. The arbiter will grant the bus by asserting **HGRANT**. The master has been granted the bus when it samples **HGRANT and HREADY** high on the rising edge of the clock. The requirement for **HREADY** results from the possibility that wait states can be inserted into a data phase. While the data phase is stalled, the address phase is also stalled, and the arbitration is also stalled.

LOCK is also part of arbitration, and is used when a bus master wants to do a sequence of transfers that must not be interrupted. Examples include read/modify/write operations used to implement semaphores and other atomic operations. The arbiter must honor **LOCK** and make sure no other bus master is granted the bus until the locked sequence finishes. The arbiter also asserts **HMASTLOCK** to indicate to slaves that a locked sequence is in progress.

Split transfers were introduced in the AHB protocol to improve bus utilization by decoupling the address and data phases for those cases where slaves require many cycles to complete a transfer. The signals involved are **HSPLIT** and **HRESP**. The split protocol allows the data phase to be postponed into the future and allows other transfers to be started on the bus. In concept, the split protocol sounds very useful, but feedback from engineers indicates it is not often used (although almost everybody claims they will use it in the future). In designs with slow peripherals or interfaces where the time to complete a transfer is variable, most designs find other ways to interface these peripherals to AHB. For this reason, no more detail about the split protocol is provided here.

Interesting situations occur when either the arbiter or the default slave uses **HREADY** to influence the arbitration. The AMBA specification makes it necessary to assert **HREADY** when there is no data phase in progress. Take for instance the first instruction fetch after reset. Since this is the first non-IDLE transfer, **HREADY** must be asserted so the CPU can receive a bus grant. This situation expands the meaning of **HREADY** beyond the basic use of inserting wait states in the address phase. Once it is realized that **HREADY** does more than just insert wait states, it can be used creatively to influence arbitration. For example, the first fetch after reset is usually from a slow memory such as flash or other off-chip memory. Since **HREADY** must be asserted to enable the first bus grant, why not assert **HREADY** for 2 clocks to provide the second bus grant as well and get the second address being accessed (address 4). This situation is shown in Figure 3-17. Notice address 0 is only on the bus for 1 clock.

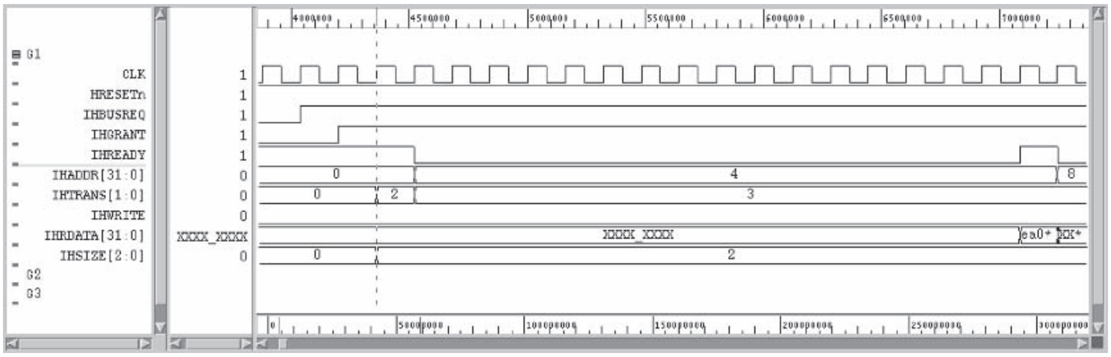


Figure 3-17: Use of HREADY in arbitration

AHB Address Phase

Once a master is granted the bus, the next phase is the address phase. During the address phase, the master will put the address on the bus, along with the other attributes that define the transfer. The address phase is only 1 clock long, but as mentioned wait states inserted into the data phase by the use of **HREADY** have the side effect of extending the next address phase since it is stalled from using the data bus. Slaves will sample the address phase signals and will prepare the response signals for the next clock cycle. The address phase signals driven by the master are: **HTRANS**, **HWRITE**, **HSIZE**, **HBURST**, and **HPROT**. Values for some of these signals are given in Figures 3-18, 3-19, 3-20, and 3-21.

Cycle Type	Description	HTRANS[1:0]
IDLE	No bus activity	00
BUSY	Master inserting wait states	01
NON-SEQUENTIAL	Transfer with address not related to the previous	10
SEQUENTIAL	Transfer with address related to the previous transfer	11

Figure 3-18: Transfer type values

Cycle Type	Description	HBURST[2:0]
SINGLE	Single Transfer	000
INCR	Incrementing Burst (length unknown)	001
WRAP4	Burst length 4 Wrapping Address	010
INCR4	Burst length 4 Incrementing Address	011
WRAP8	Burst length 8 Wrapping Address	100
INCR8	Burst length 8 Incrementing Address	101
WRAP16	Burst length 16 Wrapping Address	110
INCR16	Burst length 16 Incrementing Address	111

Figure 3-19: Burst values

Size	HSIZE[2:0]
8 bits (byte)	000
16 bits (half word)	001
32 bits (word)	010
64 bits	011
128 bits	100
256 bits	101
512 bits	110
1024 bits	111

Figure 3-20: Size values

Description	HPROT[0]
Opcode Fetch	0
Data Access	1

Description	HPROT[1]
User Access	0
Privileged Access	1

Description	HPROT[2]
Not bufferable	0
Bufferable	1

Description	HPROT[3]
Not Cacheable	0
Cacheable	1

Figure 3-21: Protection values

A basic AHB transfer is shown in Figure 3-22.

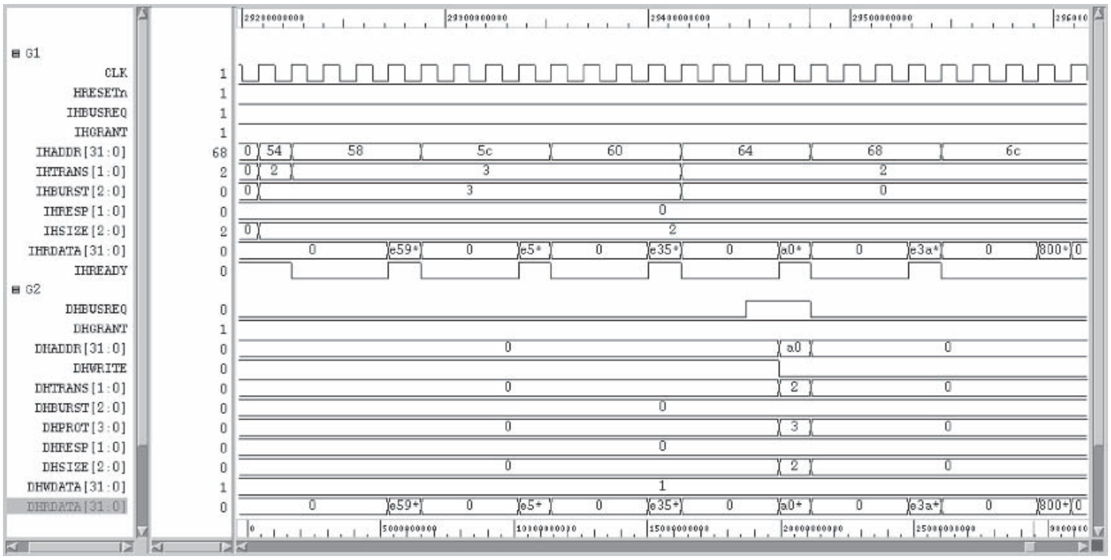


Figure 3-22: Basic AHB transfer

AHB Data Phase

The data phase is used to transfer data on the bus between master and slave. The master can insert wait states using **HTRANS** = BUSY. The slave may insert wait states by bringing **HREADY** low. The slave also uses **HRESP** to provide the status of the transfer. The possible values for HRESP are shown in Figure 3-23.

Description	HRESP[1:0]
Completed Sucessfully	00
Error occured	01
Master should retry	10
Perform Split Protocol	11

Figure 3-23: Slave transfer response values

For CPU cores, an **HRESP** of ERROR from the slave maps to an abort in the CPU exception table. There are two kinds of aborts, data and instruction prefetch as already discussed in the interrupt and exception section of this chapter.

AHB uses separate data busses for read data and write data. This avoids the need for tri-state signals. When the master performs a write **HWDATA** is used by the master for the data being written, and when the master performs a read **HRDATA** is used by the slave for the read data. Data transfers occur on the rising edge of **HCLK** when **HREADY** is high.

The data bus uses all 32 address bits along with the transfer size, **HSIZE**, to determine which portion of the data bus is being used on a particular transfer. The address is aligned to the transfer size and the portions of the data bus used on half-word and byte transfers are determined by endianness. The master must put data in correct byte lanes on writes, and the slave must do the same on reads. Unaligned memory accesses were not part of the ARM architecture until the recently introduced v6 extensions.

Some masters replicate data on writes, but it is not required. Data replication applies to transfers that are less than the bus width. In the case of a byte write, the master will put the write data on all bytes of the data bus, regardless of the address. The same is true for 16-bit writes on a 32-bit bus; the master will put the data on both halves of the data bus. If the master does replication the endianness doesn't matter, since the slave will sample the correct data no matter which bytes or half word it takes from the bus. This practice is not a great idea, since it has been shown to mask problems in slaves. If a slave is tested with a CPU core that performs data replication on writes and then is later used with a master that does not do replication, a problem may occur. This is especially true for slaves that are designed to work with both big and little endian configurations, since there is no guarantee the slave is handling endianess correctly. It is not common for slaves to do any data replication for read transfers. The ARM7TDMI always replicates data on writes to make it easier for slaves to take data from any place. This data replication is shown in Figure 3-24. The CPU is writing address `0x20000000` with data value of `0x61` which is driven on all 4 byte lanes.

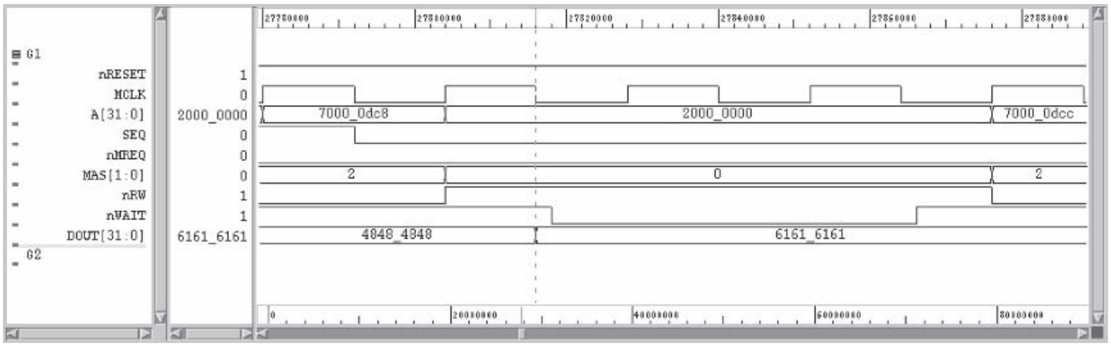


Figure 3-24: Data replication on writes

AHB does not use any byte enables to specify which byte lanes are active. This means all masters and slaves must know and understand the byte order (endianness) on the bus to interpret the portion of the data bus being used on half-word and byte transfers. This endianness agreement was perfectly fine until the ARM926EJ-S added a write buffer. The write buffer required a new signal named **DHBL** be added to specify which byte lanes are active during a write. The reason is that endianness can be dynamic, so the slave needs to know more about the transaction. Software can change endianness. This will change the value of the **BIGENDOUT** signal. Delaying writes by using a write buffer has the potential of **BIGENDOUT** changing to a new endianness value while writes of the old endianness are still in the write buffer. When these writes make it to the bus they will be inconsistent with the value of **BIGENDOUT**, hence the need for **DHBL**.

AHB-Lite

Another variety of AHB that is not covered in the AHB specification, but is covered in a separate document, is called AHB-Lite. There are many situations where AHB is deployed and only one bus master is present on the bus. In these cases there is no need for an arbiter to take care of request and grant. There is also no benefit to the split or retry protocol since there is no other master that can use the bus anyway when a slave would require a long time to complete a transfer. AHB-Lite removes the **HBUSREQ** and **HGRANT** signals and specifies that slaves cannot use the split and retry response. Any AHB master is automatically an AHB-Lite master by simply not connecting **HBUSREQ** and connecting the **HGRANT** to 1. All AHB slaves that don't implement split or retry (almost all slaves in existence today) are automatically AHB-Lite slaves. AHB slaves that do implement split or retry can be fitted with an HDL wrapper to handle the split and retry with no changes to the slave.

Single-Layer and Multilayer AHB

For an AHB system with multiple masters, there are two different styles of implementation. The traditional bus structure where there are multiple masters and slaves on a shared bus controlled by an arbitration protocol using requests and grants is called single-layer AHB. With single-layer AHB all masters share the bus bandwidth and apart from pipelining there is only one datapath between the active master and slave at any time. For SoC design where extra signals don't impact the number of pins and package size, it is possible to allow for multiple connections between masters and slaves to be active in parallel. This increases the bus bandwidth and increases performance. AHB includes a specification on how to do this called multilayer AHB. Multilayer AHB replaces the shared bus with an interconnection matrix using a multiplexing scheme to connect masters and slaves together. In addition to performance benefits, multilayer AHB also removes the requirement for arbitration on the masters and moves arbitration to the interconnection matrix. The interconnection matrix consists of a set of multiplexer functions. Each master and slave in multilayer AHB can use the AHB-Lite protocol instead of the full AHB protocol that includes arbitration.

ARM926EJ-S Example

This section uses the ARM926EJ-S as an example of different AHB bus architectures. The ARM926EJ-S utilizes a Harvard architecture with two AHB interfaces, one for the instruction bus and one for the data bus. From the above descriptions of single-layer AHB, multilayer AHB, and AHB-Lite there are many different combinations of how the ARM926 can fit into an AHB subsystem. Three possible alternatives are discussed:

- Both masters connected to single-layer AHB
- Each master connected to a different AHB
- Both masters connected to a multilayer AHB interconnect matrix

The first implementation for the ARM926 is connecting both masters to single-layer AHB. Each AHB master has a bus request (**IHBUSREQ** and **DHBUSREQ**) and a grant (**IHGRANT** and **DHGRANT**). An arbiter will determine which bus master is allowed to use the bus. The arbiter normally gives a higher priority to the data AHB. This structure is shown in Figure 3-25. For single-layer AHB, both masters must run at the same clock speed since they are on a shared bus.

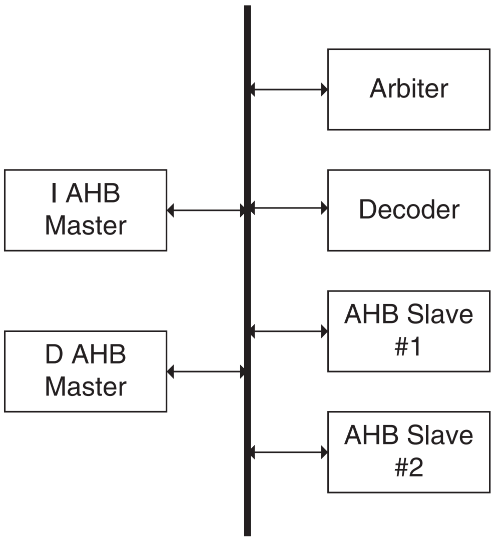


Figure 3-25: ARM926 using single-layer AHB

The second alternative is to connect each master to a separate AHB. Since the masters are (almost) independent, they do not need to use a shared AHB bus or even run at the same clock frequency. Internally, the ARM926 will use a single clock, but each AHB interface has separate clock enables (**IHCLKEN** and **DHCLKEN**). Effectively this allows the AHB bus interface units to run at different clock speeds that are multiples of each other. Figure 3-26 shows a diagram of connecting each AHB master to separate busses.

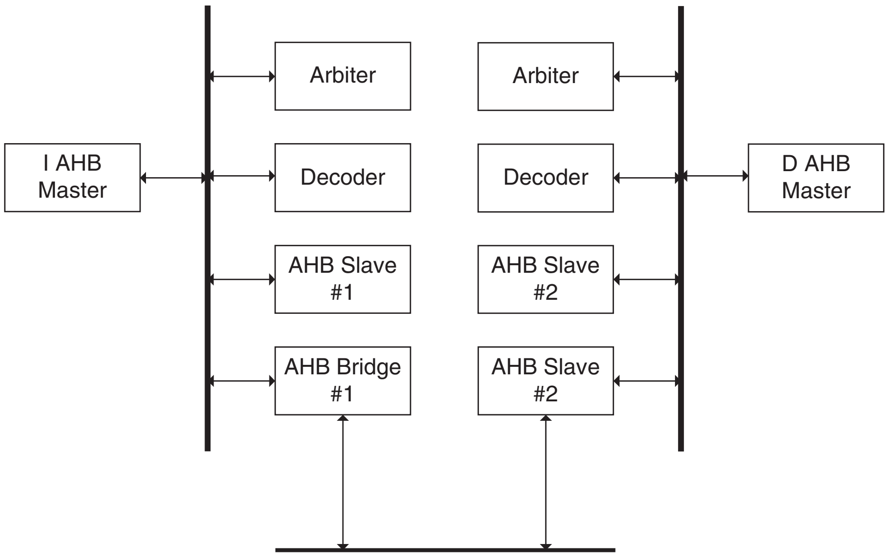


Figure 3-26: Each AHB on separate busses

The third alternative is to use multilayer AHB and connect each master to an interconnection matrix. This moves the arbitration responsibility to the interconnection matrix and can offer higher performance. The use of multilayer AHB is shown in Figure 3-27. This alternative allows concurrent transactions between multiple masters and slaves to improve performance.

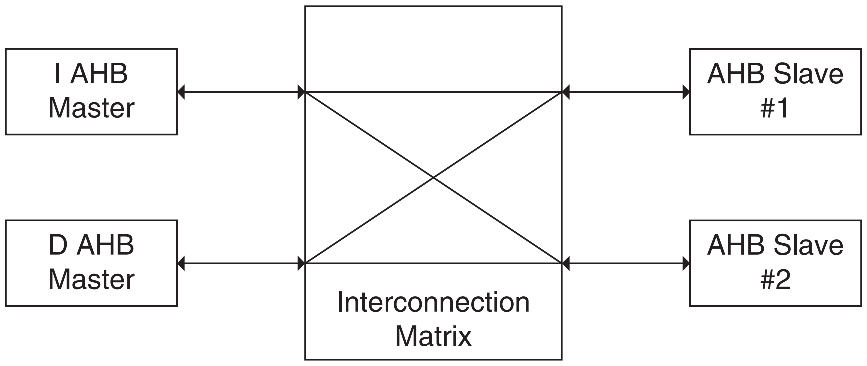


Figure 3-27: Multilayer AHB

In addition to the impact on the hardware design, software impact must also be considered when determining bus architecture. Although there is usually no dependency between instruction memory and data memory, there are situations when such dependencies exist. Some examples of dependencies between instructions and data are loading or copying instructions to a different memory, such as copying critical code from flash to SDRAM. Other examples are self-modifying code, where software explicitly modifies other software instructions. Although self-modifying code is not common practice, one example of this is the use of software breakpoints. One technique for debuggers to set breakpoints is to overwrite memory with a new instruction that will cause execution to stop at the breakpoint. Since the write to modify the instruction will take place on the data AHB and the instruction will be fetched from the instruction AHB it is possible that the new instruction may not be fetched if the write did not reach memory before the instruction fetch occurred. There are many possible reasons for the data write not reaching memory. The data could have been written into the data cache (which is a write-back cache) and may be sitting in the cache in the modified state. The data could be in the write buffer. The data write may also not be complete due to speed differences between the two AHB interfaces (the D-AHB is running very slow and the I-AHB is running very fast). These and other issues are taken care of in software using common functions to invalidate caches, making sure write buffers and prefetch buffers are flushed.

Interrupt Signals

Although interrupts are not part of the formal AMBA specifications, ARM CPU cores use two interrupt signals to indicate a request for service, **nFIQ** (fast interrupt request) and **nIRQ** (normal interrupt request). An interrupt will cause the processor to fetch and execute the interrupt service routine. This behavior on the AHB is shown in Figure 3-28.

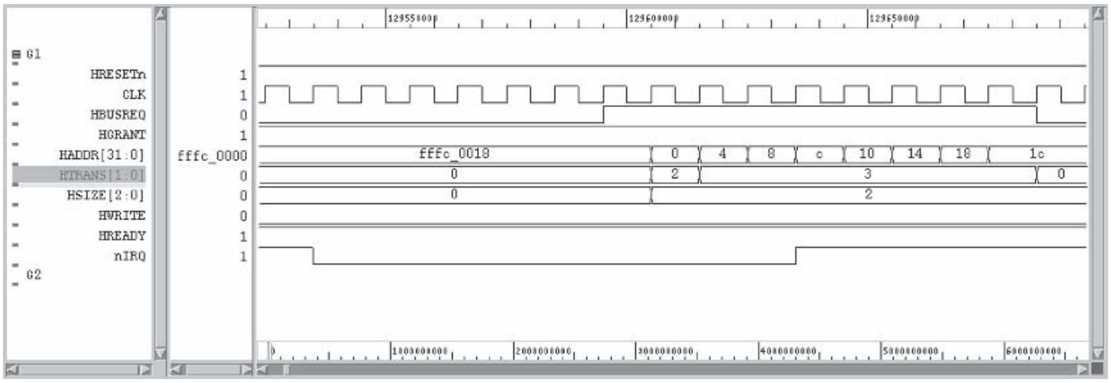


Figure 3-28 Fetching interrupt vector after an interrupt

Instruction and Data Caches

Almost all microprocessors use caching as a way to increase performance. Caches provide high memory performance by storing frequently used instructions and data. Cache operation is mostly transparent to software operation. This means software is not required to explicitly control what is stored in the cache. The cache controller hardware will automatically determine which data should be stored in the cache based on the cache algorithm. While it is not the aim of this section to explain all of the principles of caching, it provides a good overview of ARM caches and some basics of how they work.

Caches operate by defining a unit of data called a *cache line*. This is the amount of data the cache will operate on. Since bursting data on the bus is more efficient than moving single words, the cache line size is chosen as some number of bytes that represents a burst access on the bus. For 32-bit data busses a common line size is 16 or 32 bytes. This corresponds to a burst of four or eight words. Cache operations will load a cache line into the cache or write a line from cache to memory. If the CPU needs to read or write data that is not in the cache it is called a *cache miss*. If the CPU needs to read or write data that is already in the cache it is called a *cache hit*.

Memory reads that miss the cache and are retrieved via the bus are always placed in the cache. The assumption in caching is there is a high probability the software will access the same information or addresses very near to the data being read (within the same cache line). Writes that miss the cache can be handled different ways. One way is to write memory and leave the cache as it was. Another way is to write memory and then read the line into cache with the assumption will be used again. An even more efficient way is to read the line into cache and write the contents of cache without writing the data to memory. This is sometimes called a *write-allocate* because it makes all writes look like cache line reads.

There are many variations, but algorithms that ensure the latest data is consistent with memory are called *write-through caches*. With a write-through cache, write hits to the cache will update the cache and also memory (to maintain consistency). An algorithm that allows the cache to contain different data than memory is called a *write-back cache*. Write-back caches are more complex since the inconsistencies between cache and memory must be resolved if some other bus master would like to read data from memory for an address that has been modified in one of the data caches. In the workstation and server world it is common to have multiple processor systems where many processors, each with multiple levels of write-back caches, are connected to a common bus. This architecture is known as *symmetric multi-processing (SMP)*. Debugging cache coherency issues can be hair-pulling at best. Fortunately, the world of embedded systems and SoC design has not reached this level of complexity, yet.

Most ARM cores have one or more caches, but not all. Some are write-back and some are not. Synthesizable cores allow for different cache sizes to be selected. Caches are controlled via coprocessor 15, part of the CPU macrocell.

Following is a list of the ARM cores and a summary of the type of caches they use:

- **ARM7TDMI** has no cache.
- **ARM720T** contains an 8 kb unified instruction and data cache. Cache line size is 16 bytes (4 words) and it uses a 4-way set associative algorithm.
- **ARM9TDMI** has no cache.
- **ARM940T** uses separate instruction and data caches, each is 8 kb. Cache line size is 16 bytes (4 words) and it uses a 64-way set associative algorithm. The data caches can be configured for write-through or write-back operation.
- **ARM920T** uses separate instruction and data caches, each is 16 kb. Cache line size is 32 bytes (8 words) and it uses a 64-way set associative algorithm. The data caches can be configured for write-through or write back operation.
- **ARM922T** is identical to ARM920T except the cache sizes are only 8 kb.
- **ARM9E-S** has no cache.
- **ARM966E-S** has no cache.
- **ARM946E-S** uses separate instruction and data caches. It allows the designer to select from cache sizes between 0 and 1 Mb. Cache line size is 32 bytes (8 words) and it uses a 4-way set associative algorithm. The data caches can be configured for write-through or write back operation.
- **ARM9EJ-S** has no cache.
- **ARM926EJ-S** uses separate instruction and data caches. It allows the designer to select from cache sizes between 4 kb and 128 kb. Cache line size is 32 bytes (8 words) and it uses a 4-way set associative algorithm. The data caches can be configured for write-through or write back operation.

- **ARM1022E** uses separate instruction and data caches, each is 16 kb. Cache line size is 32 bytes (8 words) and it uses a 64-way set associative algorithm. The data caches can be configured for write-through or write back operation.
- **ARM1026EJ-S** uses separate instruction and data caches. It allows the designer to select from cache sizes between 0 and 128 kb. Cache line size is 32 bytes (8 words) and it uses a 4-way set associative algorithm. The data caches can be configured for write-through or write back operation.
- **ARM1136J-S** uses separate instruction and data caches. It allows the designer to select from cache sizes between 4 and 64 kb. Cache line size is 32 bytes (8 words) and it uses a 4-way set associative algorithm. The data caches can be configured for write-through or write back operation.

Cache control is done via coprocessor 15 registers. Figure 3-29 shows a small example of how to turn on caching for the ARM926EJ-S.

```
; Enable the caches
 MOV R1, #0xFFFFFFFF
 MCR p15, 0, R1, c3, c0, 0
 MRC p15, 0, R0, c1, c0, 0
 ORR R0, R0, #0x7D  ; bits 3:6 should be HIGH
 ; bit 0: mmu
 ; bit 2: d-cache
 ORR R0, R0, #0x5000  ; bit 12: i-cache
 ; bit 14: Round Robin replacement
 MCR p15, 0, R0, c1, c0, 0

 MOV    PC,   R14           ; return
```

Figure 3-29: Instructions to enable cache on ARM926EJ-S

Tightly Coupled Memory (TCM)

Embedded applications often have requirements for fast, deterministic memory to store real-time data and performance critical instruction sequences. Some ARM cores provide tightly-coupled memory (TCM) to satisfy this requirement. The ARM core provides an interface to TCM, but the memory itself is implemented outside of the CPU core. When TCM functionality is provided for Harvard architecture cores, there are separate memory interfaces for instruction (ITCM) and data (DTCM). By locating the memory outside of the core, designers have the greatest flexibility in system design and can work with differences in RAM libraries for specific semiconductor processes.

TCM is different from cache memory since it can be addressed directly by software at a specific location in the microprocessor memory map. TCM is usually implemented as single-cycle SRAM, and TCM size is specified using input signals to the CPU core. TCM status and control is done via programming registers in coprocessor 15. Software can enable and disable TCM as well as assign the location in the memory map. As an example, information about the TCM interface for the ARM926EJ-S is presented below.

Data TCM is always disabled at reset and must be explicitly enabled by software. Instruction TCM is disabled at reset unless the **INITRAM** signal is high. With **INITRAM** high, ITCM starts enabled and will respond to memory requests at address 0. This option allows the CPU to boot directly from TCM. Of course, booting from TCM implies the design can load instructions into the ITCM before reset and without the use of software. If ITCM is to be enabled at reset, but not used for the reset vectors, the ARM926EJ-S offers the **VINITHI** signal to tell the CPU to boot from address `0xffff0000` instead. When **VINITHI** is high, the CPU will locate the exception vectors at `0xffff0000`.

Because the TCM interface is optimized for single-cycle performance, there is no protection against software reading from this memory. The MMU can be used with TCM to protect against unauthorized access, but only aborted writes are guaranteed not to take place. Aborted reads will still read memory, so there is no way to protect against TCM reads.

The ITCM and DTCM signals for the ARM926EJ-S are summarized below. Signals starting with IR are for the instruction TCM (ITCM) and those starting with DR are for the data TCM (DTCM). Not all ARM cores with TCM use the same signal names, but the protocol is very similar. The less frequently used DMA signals are not covered here.

IRADDR[17:0] and **DRADDR[17:0]** are the address busses for ITCM and DTCM.

IRCS and **DRCS** are the chip selects that enable the memory.

IRIDLE and **DRIDLE** indicate the TCM interface is idle. This allows for the design to stop the clock to the memories or even power down the memories when they are not being used.

IRnRW and **DRnRW** indicate if the access is a read (low) or write (high).

IRRD[31:0] and **DRRD[31:0]** are the data busses used for reads.

IRWD[31:0] and **DRWD[31:0]** are the data busses used for writes.

IRSEQ and **DRSEQ** indicate a sequential address is being accessed. If single-cycle memory is used there is no use for this signal, but it is useful for pipelined burst memories that take two cycles for the first access then one cycle on successive accesses.

IRSIZE[3:0] and **DRSIZE[3:0]** specify the size of the memory. Size values are given in Figure 3-30. Size values should match the value programmed in cp15 by software.

IRWAIT and **DRWAIT** insert wait states in the memory access.

IRWBL[3:0] and **DRWBL[3:0]** are the write enables for each of the 4 data byte lanes to allow for byte and half-word writes. These signals are not used for reads since all reads are word reads.

Size	IRSIZE[3:0] DRSIZE[3:0]
4 kb	0011
8 kb	0100
16 kb	0101
32 kb	0110
64 kb	0111
128 kb	1000
256 kb	1001
512 kb	1010
1 Mb	1011

Figure 3-30: Values for TCM size

Following is an example of enabling the TCM in the CPU memory map. To setup a 32 kb DTCM and ITCM at addresses `0x40008000` and `0x4001000` respectively, register 9 of coprocessor 15 must be programmed. Bits [31:12] of this 32-bit register contain the base address of the TCM, bits [5:2] are programmed according to TCM size, and bit 0 is the enable/disable bit. All other bits should be programmed to 0. The instructions to program r9 of cp15 are shown in Figure 3-31.

```
; Set up the TCMs and enable them.
; This must be done before any stack use
 LDR R0, =0x40010019
 MCR p15, 0, R0, c9, c1, 1 ; Initialise ITCM

 LDR R0, =0x40008019
 MCR p15, 0, R0, c9, c1, 0 ; Initialise DTCM
```

Figure 3-31: Instruction to setup TCM

When physical address `0x40008000` is accessed by software, a DTCM access will take place with **DRADDR** of 0. When software accesses address `0x40008004`, TCM access with **DRADDR** of 4 will be done.

ARM Summary

ARM is successful because it offers a wide variety of CPU cores that meet the requirements of many SoC applications. ARM originally built its reputation on low-power as an early pioneer embedding 32-bit microprocessors into chips designed for handheld consumer electronics. Certainly, as the ARM architecture has grown and more complex designs have been completed to provide higher performance, the power must also increase to meet performance levels. Even though there are probably other CPU designs with lower power or higher performance compared to ARM, the company has continued to dominate the market by providing the broadest set of designs, tools, and partnerships to provide many ways for design projects to make use of ARM technology and the supporting tools provided by ARM partners. An engineer I met at a conference summed it up accurately, “nobody gets fired for choosing ARM.”

With a good overview of the ARM architecture and the key features of ARM cores, the next chapter will focus on hardware/software co-verification and how both hardware engineers and software engineers can better understand how to improve the process of integrating software with hardware before a design is committed to fabrication. There are many ways to do this and we will examine many of them including the pros and cons of each.

CHAPTER 4

Hardware/Software Co-Verification

Although hardware/software co-verification has been around for many years, over the last few years, it has taken on increased importance and has become a verification technique used by more and more engineers. The trend toward greater system integration, such as the demand for low-cost, high-volume consumer products, has led to the development of the system-on-a-chip (SoC). In Chapter 1, the SoC was defined as a single chip that includes one or more microprocessors, application specific custom logic functions, and embedded system software. Including microprocessors and DSPs inside a chip has forced engineers to consider software as part of the chip's verification process in order to ensure correct operation. The techniques and methodologies of hardware/software co-verification allow projects to be completed in a shorter time and with greater confidence in the hardware and software. In the EE Times "2003 Salary Opinion Survey," a good number of engineers reported spending more than one-third of their day on software tasks, especially integrating software with new hardware. This statistic reveals that the days of throwing the hardware over the cubicle wall to the software engineers are gone. In the future, hardware engineers will continue to spend more and more time on software related issues. This chapter presents an introduction to commonly used co-verification techniques.

History of Hardware/Software Co-Verification

Co-verification addresses one of the most critical steps in the embedded system design process, the integration of hardware and software. The alternative to co-verification has always been to simply build the hardware and software independently, try them out in the lab, and see what happens. When the PCI bus began supporting automatic configuration of peripherals without the need for hardware jumpers, the term *plug-and-play* became popular. About the same time I was working on projects that simply built hardware and software independently and differences were resolved in the lab. This technique became known as *plug-and-debug*. It is an expensive and very time-consuming effort. For hardware designs putting off-the-shelf components on a board it may be possible to do some rework on the board or change some programmable logic if problems with the interaction of hardware and software are found. Of course, there is always the "software workaround" to avoid aggravating hardware problems. As integration continued to increase, something more was needed to perform integration earlier in the design process. The solution is co-verification.

Co-verification has its roots in logic simulation. The HDL logic simulator has been used since the early 1990's as the standard way to execute the representation of the hardware before any chips or boards are fabricated. As design sizes have increased and logic simulation has not provided the necessary performance, other methods have evolved that involve some form of hardware to execute the hardware design description. Examples of hardware methods include simulation acceleration, emulation, and prototyping. In this chapter, we will examine each of these basic execution engines as a method for co-verification.

Co-verification borrows from the history of microprocessor design and verification. In fact, logic simulation history is much older than the products we think of as commercial logic simulators today. The microprocessor verification application is not exactly co-verification since we normally think of the microprocessor as a known good component that is put into an embedded system design, but nevertheless, microprocessor verification requires a large amount of software testing for the CPU to be successfully verified. Microprocessor design companies have done this level of verification for many years. Companies designing microprocessors cannot commit to a design without first running many sequences of instructions ranging from small tests of random instruction sequences to booting an operating system like Windows or UNIX. This level of verification requires the ability to simulate the hardware design and have methods available to debug the software sequences when problems occur. As we will see, this is a kind of co-verification.

I became interested in co-verification after spending many hours in a lab setting trying to integrate hardware and software. I think it was just too many days of logic analyzer probes falling off, failed trigger conditions, making educated guesses about what might be happening, and sometimes just plain trial-and-error. I decided there must be a better way to sit in a quiet, air-conditioned cubicle and figure out what was happening. Fortunately for me, there were better ways and I was fortunate enough to get jobs working on some of them.

Commercial Co-Verification Tools Appear

The first two commercial co-verification tools specifically targeted at solving the hardware/software integration problem for embedded systems were Eaglei from Eagle Design Automation and Seamless CVE from Mentor Graphics. These products appeared on the market within six months of each other in the 1995-1996 timeframe and both were created in Oregon. Eagle Design Automation Inc. was founded in 1994 and located in Beaverton. The Eagle product was later acquired by Synopsys, became part of Viewlogic, and was finally killed by Synopsys in 2001 due to lack of sales. In contrast, Mentor Seamless produced consistent growth and established itself as the leading co-verification product. Others followed that were based on similar principles, but Seamless has been the most successful of the commercial co-verification tools. Today, Seamless is the only product listed in market share studies for hardware/software co-verification by analysts such as Dataquest.

The first published article about Seamless was in 1996, at the 7th IEEE International Workshop on Rapid System Prototyping (RSP '96). The title of the paper was: "Miami: A Hardware Software Co-simulation Environment." In this paper, Russ Klein documented the use of an instruction set simulator (ISS) co-simulating with an event-driven logic simulator. As we will see in this chapter and the next, the paper also detailed an interesting technique of dynamically partitioning the memory data between the ISS and logic simulator to improve performance.

I was fortunate to meet Russ a few years later in the Minneapolis airport and hear the story of how Seamless (or maybe it's Miami) was originally prototyped. When he first got the idea for a product that combined the ISS (a familiar tool for software engineers) with the logic simulator (a familiar tool for hardware engineers) and used optimization techniques to increase performance from the view of the software, the value of such an idea wasn't immediately obvious. To investigate the idea in more detail he decided to create a prototype to see how it worked. Testing the prototype required an instruction set simulator for a microprocessor, a logic simulation of a hardware design, and software to run on the system. He decided to create the prototype based on his old CP/M personal computer he used back in college. CP/M was the operating system that later evolved into DOS back around 1980. The machine used the Z80 microprocessor and software located in ROM to start execution and would later move to a floppy disk to boot the operating system (much like today's PC BIOS). Of course, none of the source code for the software was available, but Russ was able to extract the data from the ROM and the first couple of tracks of the boot floppy using programs he wrote. From there he was able to get it into a format that could be loaded into the logic simulator. Working on this home-brew simulation, he performed various experiments to simulate the operation of the PC, and in the end concluded that this was a valid co-simulation technique for testing embedded software running on simulated hardware. Eventually the simulation was able to boot CP/M and used a model of the keyboard and screen to run a Microsoft Basic interpreter that could load Basic programs and execute them. In certain modes of operation, the simulation ran faster than the actual computer!

Russ turned his work into an internal Mentor project that would eventually become a commercial EDA product. In parallel, Eagle produced a prototype of a similar tool. While Seamless started with the premise of using the ISS to simulate the microprocessor internals, Eagle started using native-compiled C programs with special function calls inserted for memory accesses into the hardware simulation environment. At the time, this strategy was thought to be good enough for software development and easier to proliferate since it did not require a full instruction set simulator for each CPU, only a bus functional model. The founders of Eagle, Gordon Hoffman and Geoff Bunza, were interested in looking for larger EDA companies to market and sell Eaglei (and possibly buy their startup company). After they pitched the product to Mentor Graphics, Mentor was faced with a build versus buy decision. Should they continue with the internal development of Seamless or should they stop development and partner or acquire the Eagle product? According to Russ, the decision was not an easy one and went all the way to Mentor CEO Wally Rhines before Mentor finally decided to keep the internal project alive. The other difficult decision was to decide to whether to continue the use of instruction set simulation or follow Eagle into host-code execution when Eagle already had a lead in product development. In the end, Mentor decided to allow Eagle to introduce the first product into the market and confirmed their commitment to instruction set simulation with the purchase of Microtec Research Inc., an embedded software company known for its VRTX RTOS, in 1996. The decision meant Seamless was introduced six months after Eagle, but Mentor bet that the use of the ISS would be a differentiator that would enable them to win in the marketplace.

Another commercial co-verification tool that took a different road to market was V-CPU. V-CPU was developed inside Cisco Systems about the same time as Seamless. It was engineered by Benny Schnaider, who was working for Cisco as a consultant in design verification, for the purpose of early integration of software running with a simulation of a Cisco router. Details of V-CPU were first published at the 1996 Design Automation Conference in a paper titled "Software Development in a Hardware Simulation Environment."

As V-CPU was being adopted by more and more engineers at Cisco, the company was starting to worry about having a consultant as the single point of failure on a piece of software that was becoming critical to the design verification environment. Cisco decided to search the marketplace hopes of finding a commercial product that could do the job and be supported by an EDA vendor. At the time there were two possibilities, Mentor Seamless and Eaglei. After some evaluation, Cisco decided that neither was really suitable since Seamless relied on the use of instruction set simulators and Eaglei required software engineers to put special C calls into the code when they wanted to access the hardware simulation. In contrast, V-CPU used a technique that automatically captured the software accesses to the hardware design and required little or no change to the software. In the end, Cisco decided to partner with a small EDA company in St. Paul, MN, named Simulation Technologies (Simtech) and gave them the rights to the software in exchange for discounts and commercial support. Dave Von Bank and I were the two engineers that worked for Simtech and worked with Cisco to receive the internal tool and make it into a commercial co-verification tool that was launched in 1997 at the International Verilog Conference (IVC) in Santa Clara. V-CPU is still in use today at Cisco. Over the years the software has changed hands many times and is now owned by Summit Design.

Co-Verification Defined

Definition

At the most basic level HW/SW co-verification means verifying embedded system software executes correctly on embedded system hardware. It means running the software on the hardware to make sure there are no hardware bugs before the design is committed to fabrication. As we will see in this chapter, the goal can be achieved using many different ways that are differentiated primarily by the representation of the hardware, the execution engine used, and how the microprocessor is modeled. But more than this, a true co-verification tool also provides control and visibility for both software and hardware engineers and uses the types of tools they are familiar with, at the level of abstraction they are familiar with. A working definition is given in Figure 4-1. This means that for a technique to be considered a co-verification product it must provide at least software debugging using a source code debugger and hardware debugging using waveforms as shown in Figure 4-2. This chapter describes many different methods that meet these criteria.

```
HW/SW Co-Verification is the process of verfying embedded system
software runs correctly on the hardware design before the design
is committed for fabrication.
```

Figure 4-1: Definition of co-verification

Co-verification is often called *virtual prototyping* since the simulation of the hardware design behaves like the real hardware, but is often executed as a software program on a workstation. Using the definition given above, running software on any representation of the hardware that is not the final board, chip, or system qualifies as co-verification. This broad definition includes physical prototyping as co-verification as long as the prototype is not the final fabrication of the system and is available earlier in the design process.

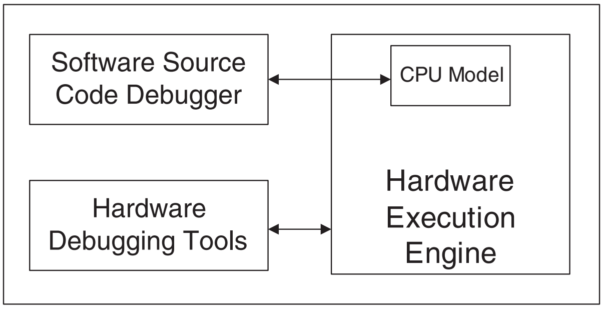


Figure 4-2: Co-verification is about debugging hardware and software

A narrower definition of co-verification limits the hardware execution to the context of the logic simulator, but as we will see, there are many techniques that do not involve logic simulation and should be considered co-verification.

Benefits of Co-Verification

Co-verification provides two primary benefits. It allows software that is dependent on hardware to be tested and debugged before a prototype is available. It also provides an additional test stimulus for the hardware design. This additional stimulus is useful to augment testbenches developed by hardware engineers since it is the true stimulus that will occur in the final product. In most cases, both hardware and software teams benefit from co-verification. These co-verification benefits address the hardware and software integration problem and translate into a shorter project schedule, a lower cost project, and a higher quality product.

The primary benefits of co-verification are:

- Early access to the hardware design for software engineers
- Additional stimulus for the hardware engineers

Project Schedule Savings

For project managers, the primary benefit of co-verification is a shorter project schedule. Traditionally, software engineers suffer because they have no way to execute the software they are developing if it interacts closely with the hardware design. They develop the software, but cannot run it so they just sit and wait for the hardware to become available. After a long delay, the hardware is finally ready, and management is excited because the project will soon be working, only to find out there are many bugs in the software since it is brand new and this is the first time is has been executed. Co-verification addresses the problem of software waiting for hardware by allowing software engineers to start testing code much sooner. By getting all the trivial bugs out, the project schedule improves because the amount of time spent in the lab debugging software is much less. Figure 4-3 shows the project schedule without co-verification and Figure 4-4 shows the new schedule with co-verification and early access to the hardware design.

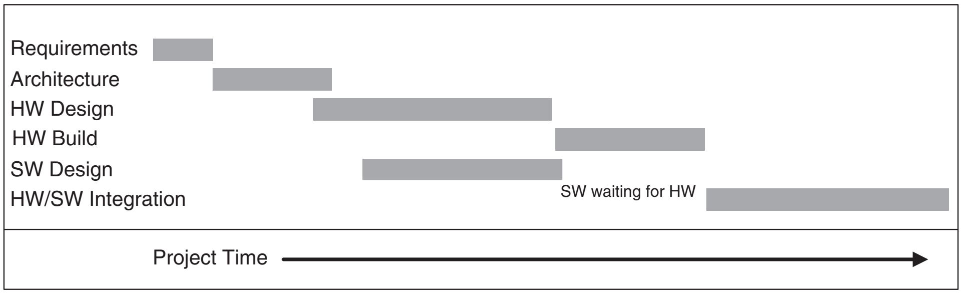


Figure 4-3: Project schedule without co-verification

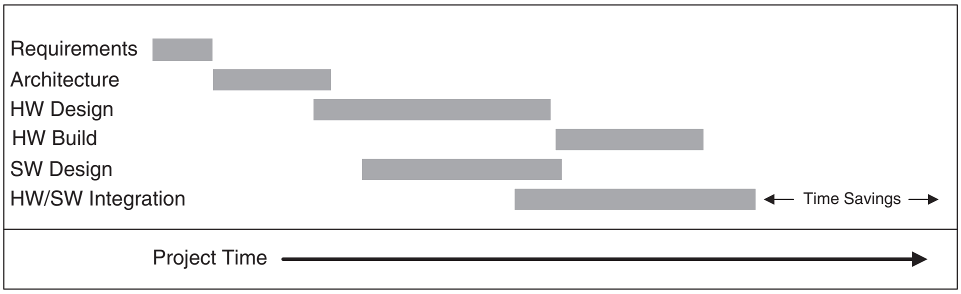


Figure 4-4: Project schedule with co-verification

Co-Verification Enables Learning by Providing Visibility

Another greatly overlooked benefit of co-verification is visibility. There is no substitute for being able to run software in a simulated world and see exactly the correlation between hardware and software. We see what is really happening inside the microprocessor in a nonintrusive way and see what the hardware design is doing. Not only is this useful for debugging, but it can be even more useful in providing a way to understand how the microprocessor and the hardware work. We will see in future examples that co-verification is an ideal way to really learn how an embedded system works. Co-verification provides information that can be used to identify such things as bottlenecks in performance using information about bus activity or cache hit rates. It is also a great way to confirm the hardware is programmed correctly and operations are working as expected. When software engineers get into a lab setting and run code, there is really no way for them to see how the hardware is acting. They usually rely on some print statements to follow execution and assume if the system does not crash it must be working.

Co-Verification Improves Communication

For some projects, the real benefit of co-verification has nothing to do with early access to hardware, improved hardware stimulus, or even a shorter schedule. Sometimes the real benefit of co-verification is improved communication between hardware and software teams. Many companies separate hardware and software teams to the extent that each does not really care about what the other one is doing, a kind of "not my problem" attitude. This results in negative attitudes and finger pointing. It may sound a bit far fetched, but sometimes the introduction of co-verification enables these teams to work together in a positive way and make a positive improvement in company culture. Figure 4-5 shows what Brian Bailey, one of the early engineers on Seamless, had to say about communication:

"Software engineering for electronic systems is a very different culture, they have very different ways of doing things. We're just beginning to find ways that the two groups can communicate. It's getting to be a cliché now. When we first started going out and telling people about Seamless, we would insist on companies that we talked to having both hardware and software engineers there for our meeting. In many of those meetings, the hardware and software guys (from within the same potential customer) literally met for the first time and exchanged business cards."
"There is still a big divide. We find there is no common boss until perhaps the vice-president level. And we are not seeing that change quickly."

Brian Bailey, chief technologist, Mentor Graphics, December 2000

Figure 4-5

Co-Verification versus Co-Simulation

A similar term to co-verification is co-simulation. In fact, the first paper published about Seamless used this term in the title. Co-simulation is defined as two or more heterogeneous simulators working together to produce a complete simulation result. This could be an ISS working with a logic simulator, a Verilog simulator working with a VHDL simulator, or a digital logic simulator working with an analog simulator. Some co-verification techniques involve co-simulation and some do not.

Co-Verification versus Co-Design

Often co-verification is lumped together with co-design, but they are really two different things. In Chapter 1, verification was defined as the process of determining something works as intended. Design is the process of deciding how to implement a required function of a system. In the context of embedded systems, design might involve deciding if a function should be implemented in hardware or software. For software, design may involve deciding on a set of software layers to form the software architecture. For hardware, design may involve deciding how to implement a DMA controller on the bus and what programmable registers are needed to configure a DMA channel from software. Design is deciding what to create and how to implement it. Verification is deciding if the thing that was implemented is working correctly. Some co-verification tools provide profiling and other feedback to the user about hardware and software execution, but this alone does not make them co-design tools since they can do this only after hardware and software have been partitioned.

Is Co-Verification Really Necessary?

After learning the definition of co-verification and its benefits, the next logical question asks if co-verification is really necessary. Theoretically, if the hardware design has no bugs and is perfect according to the requirements and specifications then it really does not matter what the software does. For this situation, from the hardware engineer's point of view, there is no reason to execute the software before fabricating the design.

Similarly, software engineers may think that early access to hardware is a pain, not a benefit, since it will require extra work to execute software with co-verification. For some software engineers, no hardware equals no work to do. Also, at these early stages the hardware may be still evolving and have bugs. There is nothing worse for software engineers than to try to run software on buggy hardware since it makes isolating problems more difficult.

The point is that while individual engineers may think co-verification is not for them, almost every project with custom hardware and software will benefit from co-verification in some way. Most embedded projects do not get the publicity of an Intel microprocessor, but most of us remember the famous (or infamous) Pentium FDIV bug where the CPU did not divide correctly. Hardware always has bugs, software always has bugs, and getting rid of them is good.

Co-Verification Methods

Most co-verification methods can be classified based on the execution engine used to run the hardware design. A secondary classification exists based on the method used to model the embedded system microprocessor. Before discussing specific co-verification methods, a quick review of some of the key ingredients in co-verification is useful.

Native Compiling Software

Many software engineers prefer to work as much as possible in the host environment (on a PC or workstation) before moving to the embedded system in a lab setting. There are two ways to do software development and software simulation in the host environment. The first is to use workstation tools to compile the embedded system software for the host processor (instead of the embedded processor) and execute it on the workstation. If the embedded system software is written in C or C++, host compiled simulation works very well for functional testing. The embedded system software now becomes a program that runs on a PC or workstation and uses all of the compilers, debuggers, profilers, and other analysis tools available for writing workstation software. Workstation tools are more plentiful and higher quality since more programmers are making use of them (remember, the embedded system space is extremely fragmented). Errors like memory leaks and bad pointers are a joy to fix on the workstation when compared to the tools available on the target system in the lab.

Instruction Set Simulation

The instruction set simulator (ISS) was mentioned in Chapter 2 as a type of microprocessor model that is used in co-verification. The second method to work in the host environment is to compile the embedded system software for the target processor using a cross compiler and simulate the software using an application called an *instruction set simulator*. The ISS is a model of the target microprocessor at the instruction level. It has the ability to load programs compiled for the target instruction set, it contains a model of the registers, and it can decode and model all of the processor's instruction set. Typically, this type of tool is accurate at the instruction level. It runs the given program in a sequential manner and does not model the instruction pipeline, superscalar execution, or any timing of the microprocessor at the hardware level in terms of a clock or digital logic. For this reason a good, fast, functional simulation is provided, but detailed timing and performance estimation is not available. Most instruction set simulators come with an interface to one or more software debuggers. The same embedded software tool companies that provide debuggers and cross-compilers may also provide the instruction set simulators. The ISS is also useful for testing compilers and debuggers without requiring a real processor on a working board. When a new processor is developed, compilers must be developed in parallel with silicon, and the ISS enables a compiler to be ready when the silicon is ready so software can be run immediately upon silicon availability.

Hardware Stubs

The major drawback of working on the host with native compiled code or the ISS is the lack of a model of the rest of the embedded system hardware. Much of the embedded system software is dependent on the hardware. Software such as diagnostics and device drivers cannot be tested without a model of how the hardware will react. This hardware dependent software is usually the most important software during the crucial hardware and software integration phase of the project. To combat this limitation, software engineers started using C code to implement simple behavioral models, or stubs, of how the target hardware is expected to behave. These stubs can provide the expected results for system peripherals and other system interfaces. Some instruction set simulators also started to incorporate hardware stubs that could be included in the simulation by providing a C interface to the memory model of the ISS. Peripherals such as timers, UARTs, and even Ethernet controllers can be included in the simulation. The number of hardware models needed to make the ISS useful will determine whether it is worth investing in creating C models of the hardware. For a large system, it can be more work to create the stubs than creating the embedded system software itself. Figure 4-6 shows a diagram of an ISS with a memory model interface that allows the user to add C code to take care of the memory accesses. Figure 4-7 shows a fragment of a simple stub model that returns the ID register of a CPU so the executing software does not get an error when it reads an expected ID code.

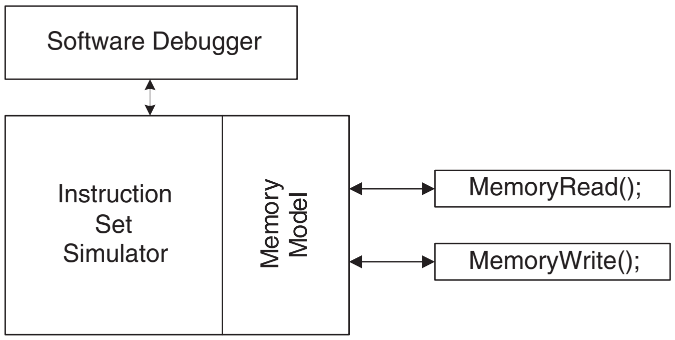


Figure 4-6: ISS with memory model interface

```
static void Access(int nRW, unsigned long addr, unsigned long *data)
{
    if (!nRW)  /* read */
    {
        if (addr == ID_REGISTER)
        {
            *data = 0x7926F; /* return ID value */
        }
    }
}
```

Figure 4-7: Code for a simple stub

Real-Time Operating System (RTOS) Simulator

For projects that use real time operating systems (RTOS), it is possible to use a host-compiled version of the RTOS. Some commercial operating system vendors provide the host-compiled version that can be run on a workstation. For custom or proprietary operating systems, the RTOS code can usually be "ported" to the host. The RTOS simulator is fast and most useful for higher levels of software. It can be used to test the calls to RTOS libraries for tasking, mailboxes, semaphores, and so forth. The RTOS simulator is more abstract then the ISS, and usually runs at a higher speed. Since the software is compiled for the host machine, it does not allow the use of any assembly language. Again, it suffers from the same limitation of the ISS since the custom hardware is not available.

An example of an RTOS simulator is VxSim, a simulation of the popular RTOS VxWorks from Wind River. VxSim allows device drivers and applications to be tested in the host environment before moving to the embedded system. Drivers usually require hardware stubs to provide simulated responses.

Microprocessor Evaluation Board

Among software engineers, the most popular tool used for learning a processor and testing code before the target system is ready is the microprocessor evaluation board. This is a board with the target microprocessor and some memory, that typically uses a network connection or a serial port to communicate with the host. It allows initial code to be developed, downloaded, and tested. Target tools are used to debug and verify the code. Many software engineers prefer to use the evaluation board since the tools are the same as those that will be used when the system is ready and it is most like working with the true product being developed. Every microprocessor vendor has an evaluation board for sale soon after the processor is available, usually at a very reasonable price. Vendors also provide sample code and even hardware schematics for the board. Some embedded system designs even go so far as to copy the evaluation board and just add a small amount of custom hardware or even buy and use the evaluation board in a product without modification. This is very tempting to get a hardware design quickly, but the boards are not usually designed for higher production volume products. Check the cost and the reliability of the design before directly using an evaluation board as part of a product.

If the embedded system contains a fair amount of custom hardware, the evaluation board is less useful. Depending on the amount and nature of the custom hardware, it may be possible to modify the evaluation board by including extra programmable logic or other semiconductor devices to make it look and act more like the target system design.

Waveforms, Log Files, and Disassembly

For SoC designs, many software engineers are forced to do early software verification with full-functional logic simulation models and waveforms in a hardware design environment. Engineers who are skilled in both software development and hardware design may be able to debug this way, but it is not the most comfortable debugging environment for most software engineers. A source level debugger with C code is preferred to bus waveforms and large log files from a Verilog or VHDL simulator.

I once introduced co-verification to a project team working on a complex video chip with four ARM CPU cores. After preaching the benefits of co-verification and the ability to debug software using a source level debugger the software engineers shook their heads and seemed to understand. Their current setup involved the use of the RTL code for the ARM cores running in a logic simulator. As part of this environment, they included a model that monitored the execution of the ARM cores and output a log file with the disassembly of the executing software as a way to track software progress. Since the tests ran very slow, they would wait patiently for simulation to complete and then get this log file and try to correlate it with the source code to see what happened. When they went to start co-verification they immediately asked if the co-verification tool could output the same kind of log file so they could track execution after the test finished. Of course, it could, but this type of debugging does not really improve their situation. After some coaxing, they agreed to try interactive software debugging with a source-level debugger and were pleased to discover this type of debugging was possible.

A Sample of Co-Verification Methods

This section introduces some of the commonly used co-verification methods and architectures used to verify embedded software running on the hardware design. All of these have some pros and cons. That is why there is so many of them and it can be difficult to sort out the choices.

Host-Code Mode with Logic Simulation

Host-code mode is a technique to compile the embedded system software, not for the embedded processor in the hardware design, but instead for the host workstation. This is also referred to as native compile. To perform co-verification the resulting executable is run on the host machine, and it connects to a logic simulator that executes the hardware design. Some type of inter-process communication (IPC) is required to exchange information between the host-compiled embedded software and the logic simulator. The IPC implementation could be a socket that allows each of the two processes to be on different machines on the network or shared memory that runs both processes on the same machine.

Host-code mode is not limited to using a logic simulator as the hardware execution engine. Any hardware execution engine can be used. Some others that have been used with host-code mode are an accelerator/emulator and a prototyping platform.

With host-code mode, a bus functional model is used in the hardware execution engine to create bus transactions for the bus interface of the microprocessor. The combination of the host-compiled program plus the bus functional model serves as a microprocessor model.

Host-code mode provides an attractive environment for both software and hardware engineers. Software engineers can continue to use the software tools they are already using, including source code debuggers and other development and debug tools on the host. Hardware engineers can also use the tools they are already using as part of the design process; a Verilog or VHDL logic simulator and associated debug tools. This requires a minimal methodology change for both groups of engineers and can benefit both software and hardware verification. The ability to do pre-silicon co-verification is a great benefit when the processor does not yet exist. Figure 4-8 shows the basic architecture.

Host-code mode can also be used when the software does not access the hardware design via a microprocessor bus, but instead via a generic bus interface like PCI. Many chips do not have an embedded microprocessor, but are designed with the PCI bus as a primary interface into the programmable registers. In this case the software can be run on the host and read and write operations from the software can be translated into PCI bus transactions in the hardware execution engine. This is a good example of when it is useful to abstract the software execution to the host and link it to hardware execution at the PCI interface.

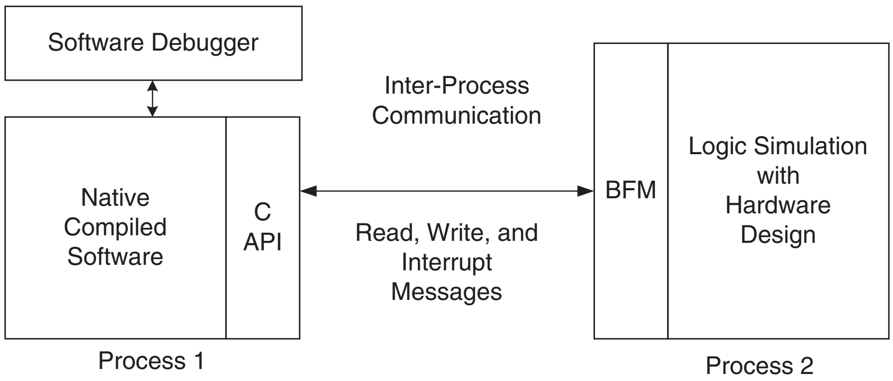

Software Debugger
Inter-Process Communication
Native Compiled Software
C API
BFM
Logic Simulation with Hardware Design
Read, Write, and Interrupt Messages
Process 1
Process 2

Figure 4-8: Host-code execution with logic simulation

Host-code mode requires the embedded software to be modified to perform function calls when it accesses the hardware design through the bus functional model. This process of putting in specific function calls can either be a pain if a lot of embedded software already exists or be little or no problem if the code is being written from scratch and all memory accesses are coded to go through a common function call. Examples of C library calls that are used for host-code execution are shown in Figure 4-9.

```
ret_val = CoverRead(address, &data, size, options);

ret_val = CoverWrite(address, data, size, options);
```

Figure 4-9: Host-code mode example function calls

Inserting these C calls into the software is called *explicit access* because the user must explicitly put in the references to the hardware design. The other way to use host-code mode is to use implicit access. Implicit access does not require the user to put in special calls, but automatically figures out when the software is accessing the hardware based on the load and store instructions being run. This technique will be covered in more detail in the next chapter, but with implicit access, the user can use ordinary C code to access hardware via pointers as shown in Figure 4-10.

```
unsigned long *ptr;
unsigned long data;
ptr = 0xff0000000 /* address of ASIC control registers */
data = *ptr; /* read the control register */
data |= 1;   /* set bit 0 to 1 */
*ptr = data; /* write new value back to control register */
```

Figure 4-10: Example of implicit access

Host-code mode can also be used to integrate an RTOS simulator such as VxSim as discussed above. A diagram of host-code execution in the context of an RTOS simulator is shown in Figure 4-11.

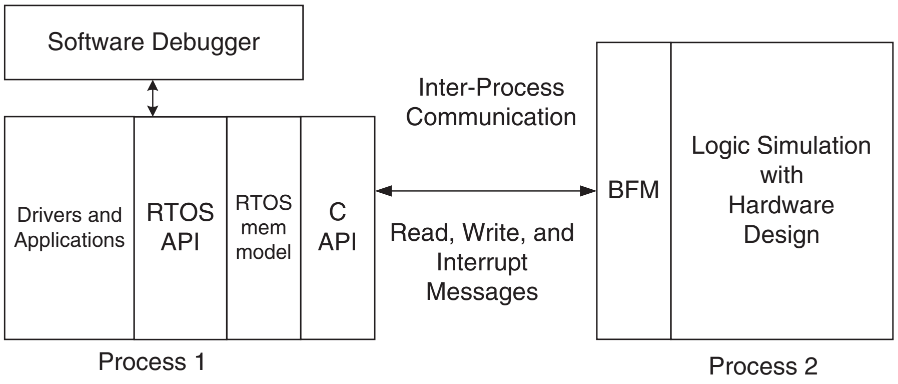

Software Debugger
Inter-Process Communication
Drivers and Applications
RTOS API
RTOS mem model
C API
Read, Write, and Interrupt Messages
BFM
Logic Simulation with Hardware Design
Process 1
Process 2

Figure 4-11: RTOS Simulation and host-code execution

Instruction Set Simulation with Logic Simulation

Another way to perform co-verification is to compile the embedded system software for the target processor and run it on an instruction set simulator. An ISS allows not only C code but also assembly language of the target processor to be run. This allows more realistic simulation of things normally coded in assembly language such as initialization sequences, cache and MMU configuration and simulation, and exception handlers. This mode of operation is referred to as target-code mode.

As with host-code mode, some type of inter-process communication (IPC) is required to exchange information between the instruction set simulator and the logic simulator.

Target-code mode is not limited to using a logic simulator as the hardware execution engine. Any hardware execution engine can be used, but since the instruction set simulator will likely run slower than a host code program it is important to make sure the speed of the instruction set simulator is not too slow to see benefits from a hardware execution engine such as an accelerator.

The bus functional models used with an ISS are the same or similar to those used in host code mode. The main difference is that with an ISS it may be possible to understand the context of the bus transactions better. In host code mode, only a single bus transaction is considered at a time. On a bus that supports address pipelining, such as AHB, there is no way to determine the next bus cycle that will be done by the host code program, so only a single transaction would be simulated and there is no pipelining. The ISS can utilize knowledge of what will be the next bus transaction to occur and can supply the bus functional model with the next address so that it can model the address pipelining correctly. This is a major benefit of using a good ISS for co-verification. Target-code mode also enables instruction fetches to be verified.

Like host-code mode, software engineers can debug code in a familiar environment. In target-code mode, the debugger is not a host debugger, but rather a debugger that can work with the ISS and debug programs cross-compiled for the embedded processor. Figure 4-12 shows the architecture.

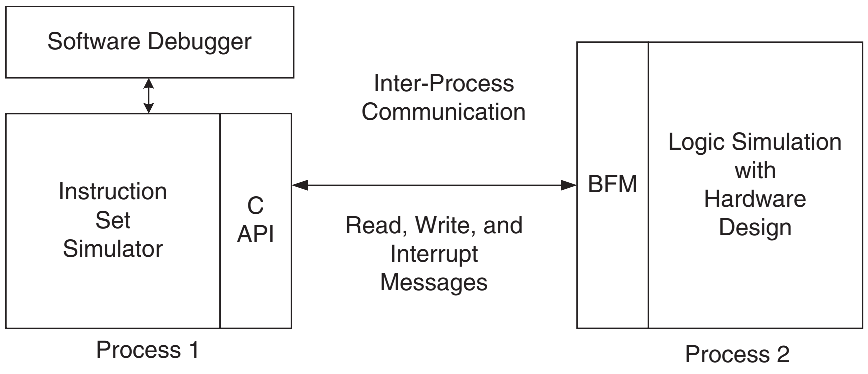

Software Debugger
Inter-Process Communication
Instruction Set Simulator
C API
Read, Write, and Interrupt Messages
BFM
Logic Simulation with Hardware Design
Process 1
Process 2

Figure 4-12: Instruction set simulator connected to logic simulation

To integrate an ISS with a bus functional model, the memory interface to the ISS must be modified to run logic simulation to satisfy the memory accesses. Instruction set simulators as used by software engineers normally have a flat memory model that is a simple C model allowing the program to be loaded and run. Some instruction set simulators have the ability to customize this memory model so the users can add their C models (stubs) to provide some rudimentary model of the hardware. Without at least the ability to put in stub models, most embedded system code will not run on a flat memory model since it deals with memory-mapped hardware registers that should have nonzero values after reset. Doing co-verification with an ISS is really just a simple extension of the use of stubs to instead turn memory transactions into calls to the logic simulator for execution on the bus functional model. The other thing that must be reported to the ISS is interrupts. When an interrupt occurs on the bus, the ISS must know about it so it can model the exception processing and start the service routine. Most commercial co-verification tools provide many more features that just gluing the memory model of the ISS to a bus functional model and reporting interrupts, but this description is easy to understand and has been used by many users to construct their own co-verification environment using an ISS.

Some instruction set simulators keep statistics and account for the simulation cycles that have been used to satisfy memory requests. This allows useful features such as performance estimation and profiling to be used to find out details of software execution. In the simple ISS integration description above, the read and write activity would have to report a number of bus clocks that were consumed to satisfy the transaction. The ISS may be able to use this clock cycle count and update its internal notion of time. Unfortunately, this is not always easy to do since the time domain of the ISS is now out-of-step with that of the logic simulator. Synchronization between the software execution environment and the hardware execution environment are discussed in the next chapter, but these types of issues have led to the shift from a transaction-based interface to one that is cycle based.

One way to think of a cycle-based ISS is to say that it exchanges pin values between the ISS and logic simulator on every bus clock cycle. This is equivalent to moving the bus functional model state machine into the ISS and just applying the signal values in logic simulation. Another way to view it is as a transaction-based interface where the logic simulator has the ability to report wait states to the ISS and the ISS will return with the same memory transaction until it completes. This approach is better suited for cases where better accuracy is needed. It is also better suited for multiprocessor designs since it can keep all processors synchronized with the logic simulator on a cycle-by-cycle basis. Figure 4-13 shows the architecture of a cycle-based instruction set simulator.

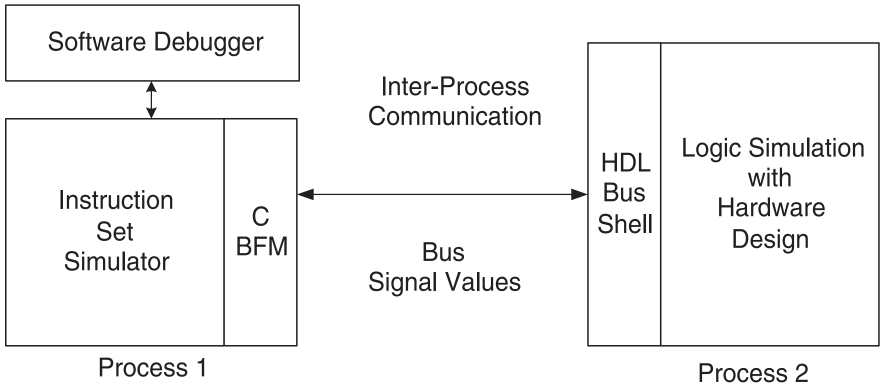

Software Debugger
Inter-Process Communication
Instruction Set Simulator
C BFM
Bus Signal Values
HDL Bus Shell
Logic Simulation with Hardware Design
Process 1
Process 2

Figure 4-13: Cycle-based instruction set simulator connected to logic simulation

C Simulation

The logic simulation and acceleration techniques discussed so far evolved from the hardware simulation domain. One complaint about co-verification developed by extending the hardware simulation platform to include software engineers includes limited availability of the platform. For example, to perform co-verification using logic simulation requires a logic simulation license for each software engineer that is running and debugging software. Most companies purchase logic simulation licenses based on the demand for hardware verification and don't have extras available for the purposes of co-verification. Similarly, higher performance hardware execution engines such as simulation acceleration and emulation are even more difficult to acquire for software development. Most companies have only one or two such machines that must be shared by verification engineers and software engineers. This limited scalability often leaves engineers wondering if there is a way to do co-verification that doesn't require traditional logic simulation.

The natural conclusion is to think about using a C or C++ simulation environment to eliminate the need for logic simulation. At the same time, there is a perception that C simulation is faster than Verilog and VHDL simulation. SystemC is one such environment that is gaining momentum as a modeling language that can provide C++ simulation of the design without requiring logic simulation, and at the same time can also co-simulate with an HDL simulator when needed. SystemC by itself is not a co-verification method, but rather an alternative hardware execution environment or even an alternative modeling language to be used instead of Verilog and VHDL. Model-based methods require a library of models to be created, and missing models are a common source of difficulty.

The question with any C simulation environment, SystemC or homegrown, has always been the development of the design model. Like the primitive hardware stub methods used by software engineers, somebody must create the simulation model of the hardware design. Since this model creation is not yet a mainstream path to design implementation, any work to create an alternative model that is not in the critical path of design implementation is usually a lower priority that may never become reality. Contrast this to logic simulation where RTL code for the design must be developed for implementation so using this RTL code in a logic simulator is always a model that is readily available.

Tools are now available to take the Verilog and VHDL code for the design and turn it into a C model by translating it into C or SystemC or even directly to an executable program that is not a traditional logic simulator. Of course, such tools must do more than just eliminate the need for the logic simulator license; they also must offer some performance gain to satisfy the perception that somehow C should be faster than Verilog or VHDL; a tough job considering the optimization already being done by today's logic simulators. By doing nothing more than eliminating the logic simulator license the price would have to dramatically lower than that of a simulator to be compelling, which is very difficult since the simulation market is mature and prices will only come down as time progresses. The approach of these Verilog to C translators is to turn the Verilog into a cycle based simulation by eliminating timing. Cycle-based simulation has never been a mainstream methodology, so it is not clear that converting Verilog code into a cycle-based executable will succeed; only time will tell. A common post on newsgroups related to Verilog simulation is from the engineer looking for the Verilog to C translator. There are many of them, and a couple of them are shown in Figure 4-14. The answer usually comes back that the best Verilog to C translator is the VCS logic simulator. Most engineers asking for the translator are not clear on how it would benefit them. In fact, many of the products mentioned are no longer available as commercial products.

```
> Was wondering if anyone could point me in the direction of a
> Verilog to C translator.....if such a thing exists.
>


> Hi all,
> I am looking for a Verilog to C converter.
```

Figure 4-14: Verilog-to-C translator requests

The only real way to gain higher performance from C or SystemC simulation is to raise the abstraction level of the model. Instead of modeling the design at RTL, more abstract models must be developed that eliminate the detail of the model and as a result enable it to run faster. The theory on high-level modeling is that an engineer can make an abstract model in about 1/10 the time it takes to develop an RTL model and the model should run 100 to 1000 times faster in a C or SystemC environment. Engineers are looking for a minimum of 100 kHz performance, and 1 MHz is more desirable. Some tools translating HDL into C are starting to show about 10x performance speedup over logic simulation by eliminating some of the detailed timing of logic simulation without requiring the user to make any changes to the RTL code. Raising the level of abstraction holds promise for running software before the RTL for the hardware design is available.

Co-verification utilizing C simulation environments is very much the same as with traditional logic simulators. Instruction set simulators and host code execution methods can be used to run the embedded system software and perform software debug. The compelling reason to look into co-verification based on C simulation is the ability to scale co-verification to many software engineers. Once a C model of the design is in place and co-verification is available, then every software engineer can use it by simply making copies of the software model. This also makes it possible to give the model and environment to software engineers that are outside the company to start developing software and doing such tasks as porting an RTOS without waiting for hardware and without the need to use logic simulation. I have never confirmed it, but I can guess that software companies such as Wind River have a need to port vxWorks to new processors and custom hardware designs before chips and boards are available. I can also guess they don't have a Verilog simulator and even if they could get a simulator they probably don't want to learn how to use it.

Companies that started out developing co-verification tools that allow users to create their own C models and combine them with microprocessor models and debugging tools to form a representation of the design face a difficult modeling dilemma about who will create the models. To enable wider use of the technology and go beyond focusing on the creation of models for custom designs, some products shifted toward the use of a C model as a replacement for the common tool that all software engineers know and love, the evaluation board. The all-software virtual evaluation board is an alternative to buying hardware, cables, power supplies, and JTAG tools. When many engineers need access to the board, it becomes much more cost effective to deploy a software version of it. In addition to basic microprocessor evaluation boards, C models can be created for reference designs and platforms that are often used as starting points for adding custom hardware. This type of virtual board enables debugging that is not possible on a real piece of hardware. Value is derived from being able to monitor hardware states and have easy access to performance information. By constraining support to off-the-shelf boards it is easier to serve the market, but does not address custom designs. Model based methods always seem to face model availability questions.

Co-verification revolving around C simulation is an interesting area that will continue to evolve as engineers start to look at top down design methodology that could leverage such a model for high-speed simulation and also use it for the design implementation.

RTL Model of CPU with Software Debugging

As we have seen, there are benefits and drawbacks of using software models of microprocessors and other hardware. This section and the next discuss techniques that avoid model creation issues by using a representation of the microprocessor that doesn't depend on an engineer coding a model of its behavior.

As the world of SoC design has evolved, the design flows used for microprocessor and DSP IP have changed. In the beginning, most IP for critical blocks such as the embedded microprocessor were in the form of hard IP. The company creating the IP wanted to make sure the user realized the maximum benefit in terms of optimized performance and area. The hard macro also allows the IP to be used without revealing all of the source code of the design. As an example, most of the ARM7TDMI designs use a hard macro licensed from ARM. Today, most SoC designs don't use hard macros but instead use a soft macro in the form of synthesizable Verilog or VHDL. Soft macros offer better flexibility and eliminate portability issues in the physical design and fabrication process.

Now that the RTL code for the CPU is available and can easily be run in a logic simulator or emulation system, everybody wants to know the best way to perform co-verification. Is a separate model like the instruction set simulator really needed? It does not seem natural to most engineers (especially hardware engineers) to replace the golden RTL of the CPU, the representation of the design that will be implemented in the silicon, with something else. The reality is that the RTL code can be used for co-verification and has successfully been used by project teams.

The drawback of using the RTL code is that it can only execute as fast as the hardware execution engine it is running on. Since it is totally inside the hardware execution engine, there is no chance to take any simulation short cuts that are possible (or automatic) with host-code execution or instruction set simulation. Historically, logic simulation has always been too slow to make the investigation of this technique interesting. After all, a simulation environment for a large SoC typically runs less than 100 cycles/sec and running at this speed it is not possible to use a software debugger to perform interactive debugging.

The primary area where this technique has seen success is with simulation acceleration and emulation systems that are capable of running at much higher speeds. With a hardware execution engine that runs a few hundred kHz up to 1 MHz it is possible to interactively debug software running on the RTL model of the CPU.

To perform co-verification with an RTL model of the microprocessor, a software debugger must be able to communicate with the CPU RTL. To debug software programs, a software debugger requires only a few primitive operations to control execution of a microprocessor. This can best be seen in a summary of the GNU debugger (gdb) remote protocol requirements. To communicate with a target CPU gdb requires the target to perform the following functions:

- Read and write registers
- Read and write memory
- Continue execution
- Single step
- Retrieve the current status of the program (stopped, exited, and so forth)

In fact, gdb provides an interface and specification called the *remote protocol interface* that implements a communication channel between the debugger and the target CPU to implement the necessary functionality to enable gdb to debug a program.

On a silicon target where a chip is placed on a board, the only way to communicate with gdb to send and receive the protocol information is by adding some special software to the user's software running on the embedded processor that will communicate with gdb to send information such as the register contents and memory contents. The piece of code added to the software is called a *gdb stub*. The stub (running on the target) communicates with gdb running on a different machine (the host) using a serial port or an Ethernet connection. While this may seem complicated, it is the easiest way to debug without requiring the CPU to provide provisions in silicon for debugging.

The good news is that for simulation acceleration and emulation applications there is much greater flexibility since it is really a simulation of the CPU RTL code and not a piece of silicon. The difference is visibility. In silicon there is no visibility. There is no way to see the values of the registers inside without the aid of software to export the values or special purpose hardware to scan out the values. Simulation, on the other hand, has very good visibility. In a simulation acceleration or emulation platform, all of the values of the registers and wires are visible at all times. This visibility makes the use of the gdb remote protocol even better than its original intent since a special stub is no longer needed by the user in the embedded system code. Now the solution is totally transparent to the user. Now gdb can use the remote protocol specification to talk to the simulation, both of which are programs running on a PC or workstation. This technique requires no changes to gdb, and the work to implement it is contained in the simulation environment to bridge the gap between gdb and the data it is requesting from the simulation. The architecture of using the gdb remote protocol with simulation acceleration and emulation is shown in Figure 4-15.

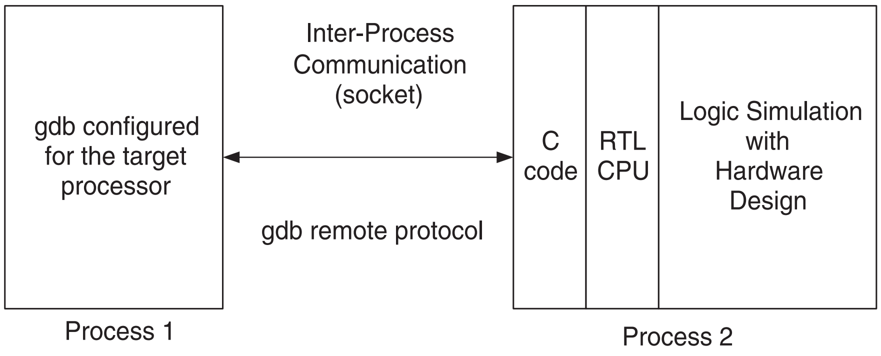

gdb configured for the target processor
Inter-Process Communication (socket)
gdb remote protocol
C code
RTL CPU
Logic Simulation with Hardware Design
Process 1
Process 2

Figure 4-15: gdb connected to the RTL code of the microprocessor

Hardware Model with Logic Simulation

Another way to eliminate the issues associated with microprocessor models is to use the concept of a "hardware model." A hardware model uses the silicon of the microprocessor as a model for Verilog and VHDL simulation. A custom socket holds the silicon and captures the outputs from the silicon, sends them to a logic simulator and applies the inputs from the simulator to the input pins of the silicon. The communication mechanism between the hardware modeler and the simulator must involve software to talk to the simulator so a network connection is most natural. The concept is much like that of a tester where the stimulus and response is provided by a logic simulator. The architecture of using the hardware model for co-verification is shown in Figure 4-16.

Software debugging with the hardware model can be accomplished in multiple ways. In the previous section, the gdb stub was presented. This is a technique that can be used on the hardware model to debug software. Unlike the RTL model in a simulation environment, the hardware model cannot provide visibility of the internal registers so the user must integrate the stub with the other software running on the microprocessor. The other technique for debugging software is a JTAG connection for those microprocessors that support this type of debugging by providing dedicated silicon to connect to the JTAG probe and debugger. In both cases, performance of the environment can limit the utility of the hardware model for software debugging.

The hardware model can also provide local memory in the hardware to service some memory requests that are not required to be simulated. For pure software development, software engineers are interested in high performance and less interested in simulation detail. By servicing some of the memory requests locally on the hardware modeler and avoiding simulation, the software can run at a much higher speed. Hardware modelers can run at speeds of up to 100 kHz when running independently of the logic simulator. Of course, in the lock step mode they will only run as fast as the logic simulator and exchange pin information every cycle.

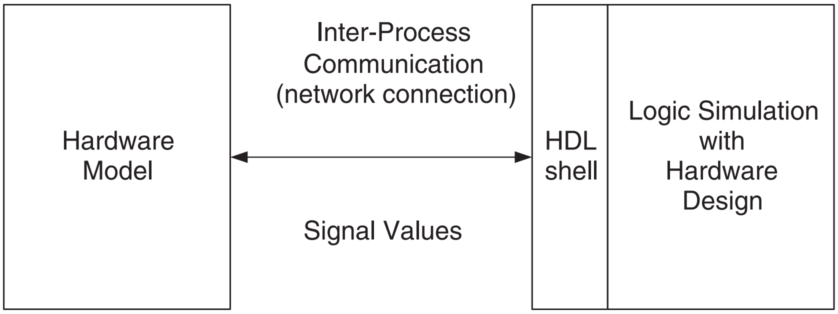

Hardware Model
Inter-Process Communication (network connection)
Signal Values
HDL shell
Logic Simulation with Hardware Design

Figure 4-16: Hardware model of the microprocessor

With the hardware model, co-verification is no longer completely virtual since a real sample of the microprocessor is used, but for those engineers that have negative experiences with poor simulation models in the past, the concept is very easy to understand and very appealing. What could be a better model than the chip itself?

Clocking limitations are one of the main drawbacks of the hardware model. To do interactive software debugging, the CPU must be capable of running slowly and maintaining its state. Early hardware modeling products were developed at a time when many microprocessor chips started using phase-locked loops and could not be slowed down because the PLLs don't work at slow speeds. To get around this problem, the hardware modeler would reset the device and replay the previous n vectors to get to vector n + 1. This allowed the device to be clocked at speeds high enough to support PLL operation, but made software debugging impossible, except by using waveforms from the logic simulator. As we have seen, today's microprocessors come in two flavors, the high-performance variety with PLLs and those more focused on low power. The high-performance variety usually have mechanisms to bypass the PLL to enable static operation and the low-power variety are meant for static design and are very flexible in terms of slow clocking and even stopping the clock. Unfortunately, experiments with such processors have revealed that when bypassing the PLL, device behavior is no longer 100% identical to behavior with the PLL. For low-power cores like ARM, irregular clocking can also be trouble since it requires the clock input to be treated more like a data input since it must be sampled in simulation and is not required to be regular.

With the RTL core becoming more common, there are now products that provide an FPGA for the synthesizable CPU and link to the logic simulator in the same way as the more traditional hardware modeler. Using the CPU in an FPGA gives some benefit by allowing JTAG debugging products to be used, but performance is still likely to be a concern. If the JTAG clock can run independently of the logic simulator, high performance can be obtained for good JTAG debugging.

Evaluation Board with Logic Simulation

The microprocessor evaluation board is a popular way for software engineers to test code before hardware is available. These boards are readily available for a reasonable cost. To extend the use of the evaluation board for co-verification, the board can serve a similar purpose as the instruction set simulator. Since most boards have networking support, a socket connection between the board and the logic simulator can be developed. A bus functional model residing in the logic simulator can interface the board to the rest of the hardware design. The architecture of using the evaluation board for co-verification is shown in Figure 4-17.

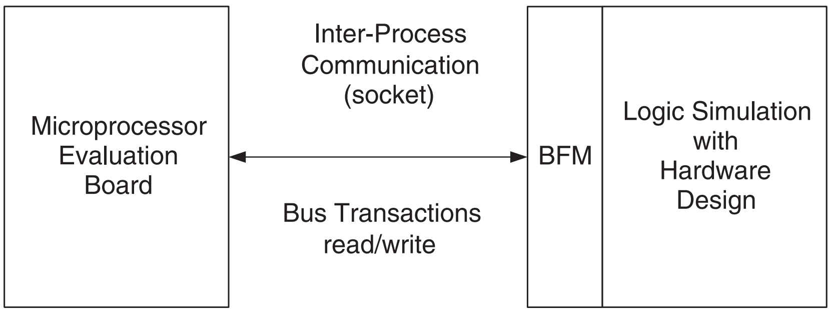

Microprocessor Evaluation Board
Inter-Process Communication (socket)
Bus Transactions read/write
BFM
Logic Simulation with Hardware Design

Figure 4-17: Microprocessor evaluation board with logic simulation

This combination of a CPU board connected to logic simulation via a socket connection and BFM is most appealing to software engineers since the performance of the board is very good. Since each is running independently, there is no synchronization or correlation between the two time domains of the board and the logic simulator.

The drawback to this type of environment is the need to add custom software to the code running on the CPU board to handle the socket connection to the logic simulator. Some commercial co-verification vendors provide such a library that may be suitable, but must always be modified since each board is different and the software operating environment is different for different real-time operating systems. Although the solution requires a lot of customization, it has been used successfully on projects.

In-Circuit Emulation

In-circuit emulation involves using external hardware connected to an emulation system that runs at much higher speeds than a logic simulator. Emulation is an attractive platform to do co-verification since the higher speed enables software to run faster. This section discusses three different ways to perform co-verification with an emulation system.

The first method is useful for microprocessor cores that are available in RTL form. As we have seen, there is a trend for the IP vendors to provide RTL code to the user for the purposes of simulation and synthesis. If this is available, the microprocessor can be mapped directly into the emulation system. Most cores used in SoC design today support some kind of JTAG interface for software debugging. To perform co-verification a software engineer can connect a JTAG probe to the I/O pins of the emulator and communicate with the CPU that is mapped inside the emulator. The architecture of using a JTAG connection to an emulator for co-verification is shown in Figure 4-18.

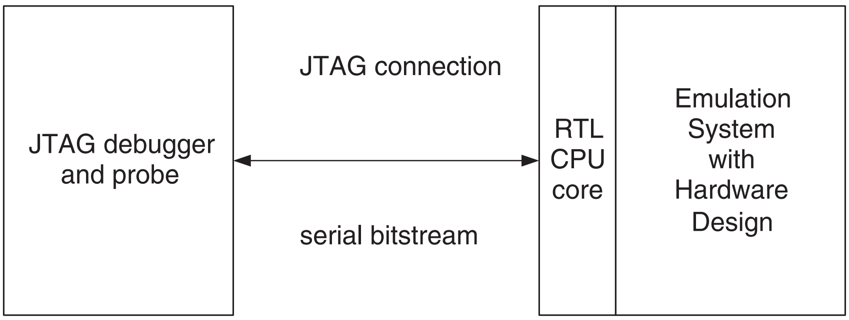


Figure 4-18: JTAG connection to an emulation system

In this mode of operation, the CPU runs at the speed of the emulation system, in lock-step with the rest of the design. The main issues in performing co-verification are the overall speed of the emulator and its ability to maintain the JTAG connection reliably at speeds that are lower than most hardware boards.

A second way to perform co-verification with an emulation system is to use a board with the microprocessor test chip and connect the pins of the chip to the I/O pins of the emulator. This technique is useful for hard macro microprocessor IP such as the ARM7TDMI that cannot be mapped into the emulation system. JTAG debugging can also be done by connecting to the JTAG port on the chip. The architecture of using a JTAG connection to an emulator for co-verification is shown in Figure 4-19.

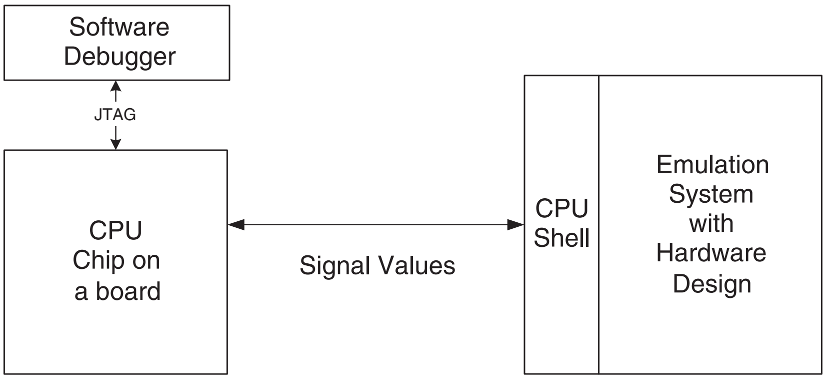


Figure 4-19: JTAG connection with test chip and emulation system

Like the previous method, the CPU core will run at the speed of the emulation system. Signal values will be updated on each clock cycle. The result is a cycle-accurate simulation of the connection between the test chip and the rest of the design. The cycle-accurate lock-step simulation is desired for hardware engineers that want to model the system exactly and want to run faster using emulation technology for long software tests and regression tests.

In both of the previous techniques, the user must make sure to confirm that the JTAG software and hardware being used for debugging can tolerate slow clock speeds. Most emulation systems run in the 250 kHz to 1 MHz range depending on the emulation technology and the design being run on the emulator. While this is much faster than a logic simulator, it is much slower than what the developers of the JTAG tools probably expected. Most JTAG tools have built in timeouts, either in the hardware or in the software debugger (or both) for situations when the design is not responding. It is crucial to verify that these timeouts can be turned off. Emulation, like simulation, allows the user to stop the test by pressing Ctrl+c, waiting for some unspecified amount of time, and then restarting operation. If timeouts exist in the JTAG solution, this will certainly cause a disconnect and result in the loss of software debugging. The best way to provide a stable JTAG connection is to use a feedback clock to the JTAG hardware to help it adapt its speed based on the speed of the emulation system.

The third co-verification method commonly used with emulation is to use a speed bridge between hardware containing a microprocessor device and the emulation system. The classic case for this application is for verification of a chip that connects to the PCI bus. A common setup is for software engineers that are developing device drivers for operating systems such as Windows or Linux and the board they are writing the driver for sits on the PCI bus. Since the PCI board is not yet available, they can use a PC to test the software and the emulation system provides a PCI board that plugs into the PC and bridges the speed differences between the real speed of the PCI bus in the PC (33 or 66 MHz) and the slower speed of the emulator. The PC will run at full speed until the device driver makes a memory or I/O access to the slot with the hardware being developed. When this occurs, the bridge to the emulator will detect the PCI transaction and send it over to the emulator. While the emulator is executing the PCI transaction, the bridge card will continuously respond with a retry response to stall the PC until the emulator is ready. Eventually, the emulator will complete the PCI transaction and the bridge card will complete the transaction on the PC. This method is shown in Figure 4-20.

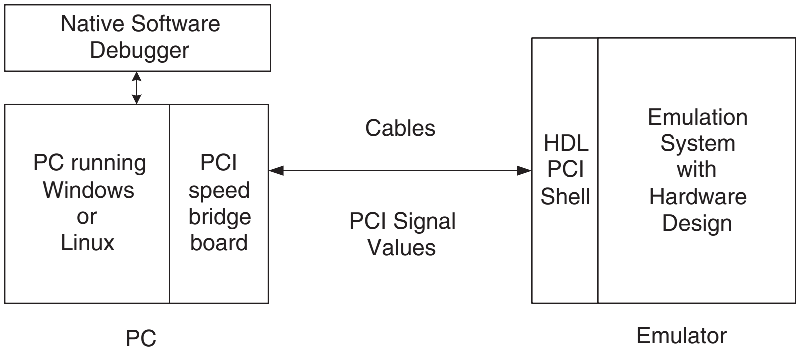


Figure 4-20: JTAG connection speed bridge and emulation system

Similar environments are common for embedded systems where a board containing a microprocessor can run an RTOS such as VxWorks and communicate with the emulator through a speed bridge for a bus such as PCI or AHB.

FPGA Prototype

I always get a laugh when the FPGA prototype is discussed as a co-verification technique. Prototyping is really just building the system out of programmable logic and using the debugger just as if the final hardware was constructed. The only difference may be ASICs are substituted for FPGA, and as a result the performance is lower than the final implementation. Since hardware debugging is very difficult, prototyping barely qualifies as co-verification, but since the representation of the hardware is not the final product it is a useful way for software engineers to get early access to the hardware to debug software.

Recent advances in FPGA technology have caused many projects to re-examine hardware prototyping. With FPGAs from Altera and Xilinx now exceeding 250k to 500k ASIC gates, custom prototyping has become a possibility for hardware and software integration. Until now design flow issues, tool issues, and the great density differences between ASIC and FPGA have limited the use of prototyping. With the latest FPGA devices, most ASICs can now be mapped into a set of one to six FPGAs. New partitioning tools have also been introduced that work at the RT level and do not require changes to the RTL code or difficult gate-level, post-synthesis partitioning.

Although prototyping is easier than it has ever been it is still not a trivial task. Prototyping issues fall into two categories: FPGA resource issues and ASIC/FPGA technology differences. Common resource issues can be the limited number of I/O pins available on the FPGA or the number of clock domains available in an FPGA. Technology differences can be related to differences in synthesis tools forcing the user to modify the design to map to the FPGA technology. Another common technology issue is gated clocks that are difficult to handle in FPGA technology. If resource and technology issues can be overcome, prototyping can provide the highest performance co-verification solution that is scalable to large numbers of software engineers. Before committing to prototyping is it important to clearly understand the issues as well as the cost. On the surface, prototyping appears cheap compared to alternatives, but like all engineering projects cost should be measured not only in hardware but also in engineering time to create a working solution.

Co-Verification Metrics

Many metrics can be used to determine which co-verification methods are best for a particular project. Following is a list of some of them:

- Performance (speed)
- Accuracy
- Synchronization
- Type of software to be verified
- Ability to do hardware debugging (visibility)
- Ability to do performance analysis
- Specific vs. general-purpose solutions
- Software only (simulated hardware) vs. hardware methods
- Time to create and integrate models: bus interface, cache, peripherals, RTOS
- Time to integrate software debug tools
- Pre-silicon compared to post-silicon

Performance

It is common to see numbers thrown out about cycles/sec and instructions/sec related to co-verification. While some projects may indeed achieve very high performance using co-verification, it is difficult to predict performance of a co-verification solution. Of course every vendor will say that performance is "design dependent," but with a good understanding of co-verification methods it is possible to get a good feel for what kind of performance can be achieved. The general unpredictability is a result of two factors; first, many co-verification methods use a dual-process architecture to execute hardware and software. Second, the size of the design, the level of detail of the simulation, and the performance of the hardware verification platform results in very different performance levels. The next chapter will provide more information about co-verification performance.

Verification Accuracy

While performance issues are the number one objection to co-verification from software engineers, accuracy is the number one concern of hardware engineers. Some common questions to think about when evaluating co-verification accuracy are listed here. The key to successful hardware/software co-verification is the microprocessor model.

- *How is the model verified to guarantee it behaves identically to the device silicon?*

  Software models can be verified by using manufacturing test vectors from the microprocessor vendor or running a side-by-side comparison with the microprocessor RTL design database. Metrics such as code coverage can also provide information about software model testing. Alternatively, not all co-verification techniques rely on separately developed models. Techniques based on RTL code for the CPU can eliminate this question altogether. Make sure the model comes with a documented verification plan. Anybody can make a model, but the effort required to make a good model should not be underestimated.

- *Does the model contain complete functionality, including all peripherals?*

  Using bus functional models was a feasible modeling method before so many peripherals were integrated with the microprocessor. For chips with high integration, it becomes very difficult to model all of the peripherals. Even if a device appears to have no integrated peripherals, look for things like cache controllers and write buffers.

- *Is the model cycle accurate?*

  Do all parts of the model take into account the internal clock of the microprocessor? This includes things such as the microprocessor pipeline timing and the correlation of bus transaction times with instruction execution. This may or may not be necessary depending on the goals of co-verification. A noncycle accurate model can run at a higher speed and more may be more suitable for software development.

- Are all features of the bus protocol modeled?

  Many microprocessors use more complex bus protocols to improve performance. Techniques such as bus pipelining, bursting, out-of-order transaction completion, write posting, and write reordering are usually a source of design errors. Simple read and write transactions by themselves rarely bring out hardware design errors. It is the sequence of many transactions of different types that bring out most design errors.

  There is nothing more frustrating than trying to use a model for a CPU that has multiple modes of operation only to find out that the mode that is used by the design suffers from the most dreaded word in modeling, “unsupported.”

I once worked on a project involving a design with the ARM920T CPU. This core uses separate clocks for the bus clock (BCLK) and the internal core clock (FCLK). The clocking has three modes of operation:

- *FastBus Mode*: The internal CPU is clocked directly from the bus clock (BCLK) and FCLK is not used.
- *Synchronous Mode*: The internal CPU is clocked from FCLK which must be faster and a synchronous integer multiple of the bus clock (BCLK).

- *Asynchronous Mode*: The internal CPU is clocked from FCLK which can be totally asynchronous to BCLK as long as it is faster than BCLK.

As you can probably guess, the asynchronous mode caused this particular problem. The ARM920T starts off using FastBus mode after reset until which time software can change a bit in coprocessor 15 to switch to one of the other clocking modes to get higher performance. When the appropriate bit in cp15 was changed to enable asynchronous mode, a mysterious message comes out:

```
"Set to Asynch mode, WARNING this is not supported"
```

It is quite disheartening to learn this information only after a long campaign to convince the project team that co-verification is useful.

Following are some other things to pay attention to when evaluating models used for co-verification:

- *Can performance data be gathered to ensure system design meets requirements?*

  If the model is not cycle accurate, the answer is NO. Both hardware and software engineers are interested in using co-verification to obtain measurements about bus throughput, cache hit rates, and software performance. A model that is not cycle accurate cannot provide this information.

- *Will the model accurately model hardware and software timing issues?*

  Like the bus protocol, the more difficult to find software errors are brought out by the timing interactions between software and hardware. Examples include interrupt latency, timers, and polling.

When it comes to modeling and accuracy issues there are really two different groups. One set of engineers is mostly interested in the value of co-verification for software development purposes. If co-verification provides a way to gain early access to the hardware and the code runs at a high enough speed the software engineer is quite happy and the details of accuracy are not that important. The other group is the hardware engineers and verification engineers that insist that if the simulation is not exact then there is no reason to even bother to run it. Simulating something that is not reality provides no benefit to these engineers. The following examples demonstrate the difficulty in satisfying both groups.

AHB Arbitration and Cycle Accuracy Issues

I once worked with a project that used a single-layer AHB implementation for the ARM926EJ-S. One way to perform co-verification is to replace a full-functional logic simulation model of the ARM CPU by a bus functional model and an instruction set simulator. Since the ARM926 bus functional model will actually implement two bus interfaces, a question was raised by the project team about arbitration and the ordering of the transfers on the bus between the IAHB and the DAHB. The order in which the arbiter will grant the bus is greatly dependent on the timing of the bus request signals from each AHB master. From the discussion of AHB the **HREADY** signal plays a key role in arbitration since the bus is only granted when both **HGRANT** and **HREADY** are high. The particular bus functional model being used decided that since **HREADY** is required to be high for arbitration and data transfer it can be used more like an enable for the bus interface model state machine since nothing can happen without it being high. To optimize performance the bus functional model decided to do nothing during the time when **HREADY** was low. This assumption about the function of **HREADY** is nearly correct, but not exactly. The CPU indication to the bus interface unit of the ARM926 that it needs to request the bus has nothing to do with **HREADY** or the bus interface clock, it uses the CPU clock. This produced a situation where the **IHBUSREQ** and **DHBUSREQ** were artificially linked to the **HREADY** and the timing of these was incorrect. The result to the user was the arbiter granting the IAHB to use the bus instead of the DAHB. Since the two busses are independent, except for the few exceptions we discussed, there is no harm in running the transactions in a different order on the single-layer AHB. Functionally, this makes no difference and all verification tests and software will execute just fine, but to hardware engineers seeking accuracy the situation is no good. This case does bring up some interesting questions related to performance:

- Does arbitration priority affect system performance?
- What are the performance differences between single-layer AHB versus multilayer AHB versus separate AHB interfaces?

Figure 4-21 shows the correct bus request timing. The sequence shows the IAHB reading addresses 0x44 and 0x48 followed by the DAHB reading from address 0x90. It is difficult to see, but the next transaction is the IAHB reading from address 0x4c. Notice the timing of **DHBUSREQ** at the start of the waveform. It transitions high before the first **IHREADY** on the waveform. This demonstrates that the timing of **DHBUSREQ** is not related to **HREADY**.

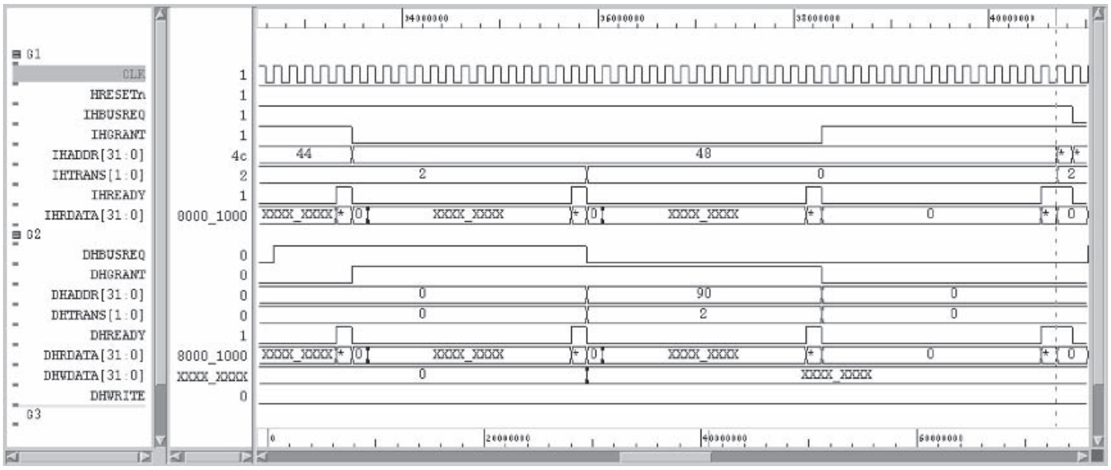


Figure 4-21: Correct timing of bus request

Figure 4-22 shows the incorrect ordering of bus transfers caused by the difference in the timing of **DHBUSREQ**. The sequence start the same way with IAHB reads from 0x44 and 0x48, but the read from 0x4c comes before the DAHB read from address 0x90. The reason is the timing of **DHBUSREQ**. Notice **DHBUSREQ** transitions high AFTER the first **IHREADY** on the waveform. This difference results in out-of-order transactions.

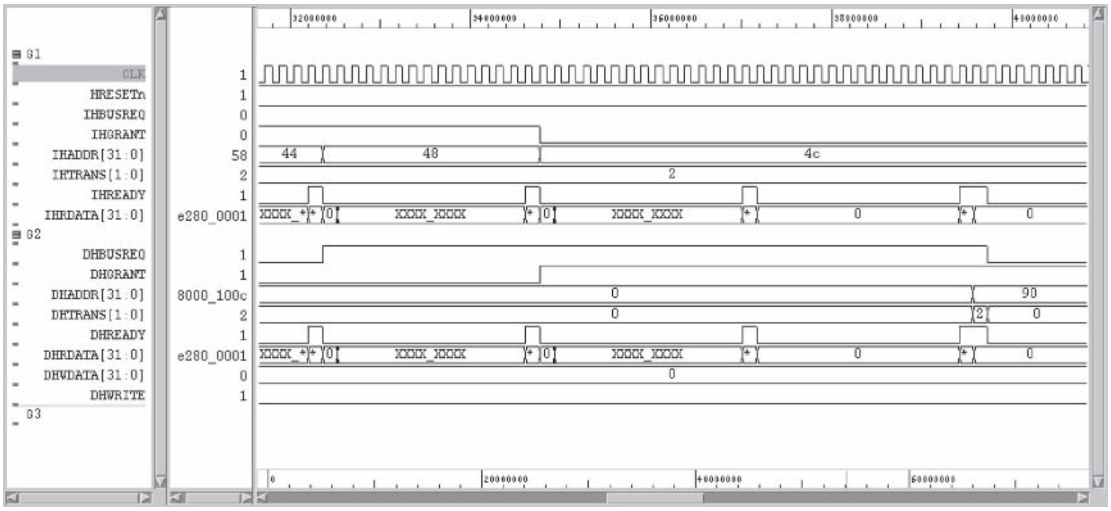


Figure 4-22: Incorrect timing of bus request

Contrast this pursuit of accuracy with a software engineer I met once that didn't care anything about the detail of the hardware simulation. Skipping the majority of the simulation activity just to run fast was the best way to go. He had no desire to run a detailed, cycle-accurate simulation. Actually, he was interested in making sure the software ran on the cycle-accurate simulation, but once it had been debugged using a noncycle accurate co-verification environment the final check of the software was better suited for a long batch simulation using the ARM RTL model and farm of workstations that was maintained by the hardware engineers, not him. Since the chance of finding a software bug was low there was no reason to worry about the problem of debugging the software in a pure logic simulation environment using waveforms or logfiles.

Modeling Summary

Modeling is always painful. There is no way around it. No matter what kind of checks and balances are available to compare the model to the actual implementation, there are always differences. One of the common debates is about what represents the golden view of the IP. In the case of ARM microprocessors, there are three possible representations that are considered "golden":

- RTL code (for synthesizable ARM designs)
- Design sign-off model (DSM) derived from the implementation
- Silicon in the form of a test chip or FPGA

Engineers view these three as golden, even more so than the specification. Co-verification techniques that use models that do not come from one of these golden sources are at a disadvantage since any problems are always blamed on the "non-golden" model. I have seen cases where the user's design does not match the bus specification, but does work when simulated with a golden model. Since a spec is not executable, engineers feel strongly that the design working with the golden model is most important, not the spec. When alternative models are used for co-verification a model that conforms to the spec is still viewed as a buggy model in any places it differs from the golden model. It is not always easy to convince engineers that a design that runs with many different models and adheres to the specification is better than a design that runs only with the golden model.

Synchronization

Most co-verification tools operate by hiding cycles from the slower logic simulation environment. Because of this, issues related to synchronization of the microprocessor with the rest of the simulated design often arise. This situation is also true using in-circuit emulation with a processor linked to the emulator via a speed bridge. In co-verification there are two distinct time domains, the microprocessor model running outside of the logic simulator and the logic simulator itself. Understanding the correlation of these two time domains is important to achieving success with co-verification. Co-verification uses mainly the spatial memory references to decide when software meets the hardware simulation. Synchronization is defined by what happens to the logic simulator when there is no bus transaction occurring in the logic simulator. It could be stopped until a new bus transaction is received. It could just "drift" forward in time executing other parts of the logic simulation hardware (even with an idle microprocessor bus). In either case the amount of time simulated in the logic simulator and the microprocessor model is different. Another alternative is to advance the logic simulation time the proper number of clock cycles to account for the hidden bus transaction but don't run the transaction on the bus. Now the correction of the time domains is maintained at the expense of performance. Synchronization is also important for those temporal activities where the hardware design communicates with software such as interrupts and DMA transfers. Without proper synchronization, things like system timers and DMA transfers may not work correctly because of differences in the two time domains.

Types of Software

The type of software to be verified also has a major impact on which co-verification methods to deploy. As we saw in Chapter 2, there are different types of software that engineers see as candidates for co-verification: system diagnostics, device drivers, RTOS and application code. Different co-verification methods are better suited to different types of software. Usually the lower level software requires a more accurate co-verification environment and higher-level software is less interested in accuracy and more focused on performance because of the code size. Running an RTOS such as VxWorks has been shown to be viable by multiple co-verification methods including in-circuit emulation, an ISS and the RTOS simulator, VxSIM. Even with the marketing claims that software does not have to be modified, expect some modification to optimize such things like long memory tests and UART accesses. The major confusion today exists because of the many types of software and the many methods of hardware execution. Often, different levels of performance will enable different levels of software to be verified using co-verification. A quick sanity check to calculate the number of cycles required to run a given type of software and the speed of the environment will ensure engineers can remain productive. If it takes 1 hour to run a software program to get to the new software this means the software engineer will have only a handful of chances per day to run the code and debug any problems found.

Other Metrics

Besides performance and accuracy, there are some other metrics worth thinking about. Project teams should also determine if a general-purpose solution is important versus a project specific solution. General-purpose solutions can be reused on future projects and only one set of tools needs to be learned. Unfortunately, general-purpose solutions are not general if the model used on the next project is not available. Methods using the evaluation board or prototyping are more specific and may not be applicable on the next project. For many engineers, especially software engineers, a solution that consists of simulation only is preferred over one that contains hardware. Another important distinction is whether the solution is available pre or post silicon. Many leading edge projects use microprocessors that are not yet available and a pre-silicon method is required. All of these variables should be considered when deciding on a co-verification strategy.

Understanding all of these metrics will avoid committing to a co-verification solution that will not meet the project needs. Remember, the goal of co-verification is to save time in the project schedule. With the understanding of co-verification methods and metrics, the next chapter will look into more details of how co-verification works and how to get the most benefit from it.

This page intentionally left blank

CHAPTER 5

Advanced Hardware/Software Co-Verification

Hardware/software co-verification is much more than executing a hardware design before fabrication and using an interactive software debugger to do basic operations like breakpoint, single step, and view memory. Using co-verification effectively requires finding the right mix of performance, simulation detail, and debugging views for the job. Unfortunately, in co-verification there is no one-size-fits-all solution that can be applied to every problem. Far too often, I see projects where one engineer sets up a co-verification method or configuration (probably best suited for his own work) and other engineers blindly use the same configuration for tasks that would be better suited to some other method or configuration. One of the causes is software engineers cannot easily modify the simulation environment without some help from hardware or verification engineers. Conversely, hardware engineers do not always understand the needs of software engineers and try to apply a one-size-fits all solution. This chapter discusses some of the advanced topics of co-verification to help users get the most benefit from co-verification.

Direct Access to Simulation Memories

One of the benefits of simulation is the level of visibility and controllability that is available. This is most useful for co-verification in the area of memory. Embedded software makes use of a memory map and microprocessor addresses to access different types of memory and memory mapped registers in the design. Software debugging is also memory intensive. The primary operation performed by a software debugger is reading memory. Scrolling the debugger's memory window or source code window requires quantities of memory data.

In Chapter 2, we discussed accessing simulation memory models in ways other than performing simulation and changing interface signals to do a read or a write. One example of this is the $readmemh system task in Verilog that can be used load memory contents from a text file. Another example is the use of a C programming interface to read and write simulation memory models directly without performing simulation. This feature is useful in co-verification because a software debugger can now read and write memory directly without performing any simulation. Example C API functions from the xsim logic simulator are shown in Figure 5-1.

```
/* get a verilog memory handle by the path name */
extern void* axisGetVmemHandle(char* path, int* left, int* right, int* top, int*
bottom);

/* read 'data' from verilog memory 'handle' at 'addr'. data is in verilog memory
format */
extern int axisReadVmem(void* handle, int addr, unsigned char* data);

/* write 'data' to verilog memory 'handle' at 'addr'. data should be in verilog
memory format */
extern int axisWriteVmem(void* handle, int addr, unsigned char* data);

/* tell the simulator to propagate verilog memory 'handle' */
extern int axisPropVmem(void* handle);

/* like $readmemh, read verilog memory 'handle' from 'file' in hex */
extern int axisReadVmemh(void* handle, char* file);

/* like $readmemb, read verilog memory 'handle' from 'file' in binary */
extern int axisReadVmemb(void* handle, char* file);
```

Figure 5-1: Example C-API for Verilog memory models

Before understanding how direct memory access is used, it is important to understand the two types of memory accesses that are present when debugging software. The first type of memory access comes from the software instructions running on the CPU. This could be instruction fetches, or data operations such as memory load and store instructions. Even if a software debugger is not used, these will always occur on the bus when the program is run. This type of memory access is shown in Figure 5-2 in the area marked (1). The second type of memory access comes from the debugger.

Software debuggers read both CPU registers and memory data to display the current context of the software. Many debugger commands require access to memory to display useful information. The debugger can even change the program or data values to change software and system behavior. Memory accesses generated by the debugger are shown in the area marked (2) in Figure 5-2.

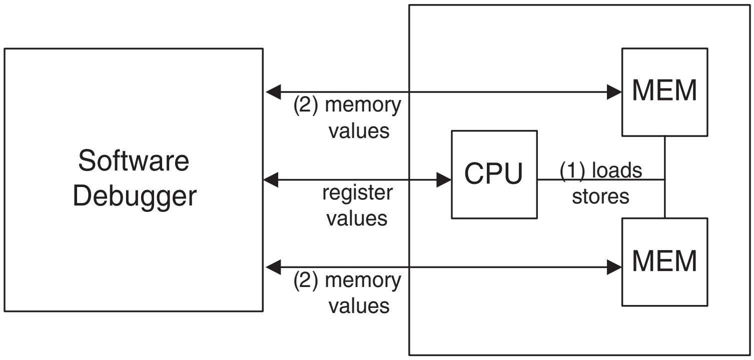


Figure 5-2: Two types of memory accesses

Since memory operations caused by the debugger are not part of the embedded software they are intrusive because they cause extra activity that is not normally part of the hardware/software interaction. Most software engineers can relate to embedded system problems that are timing related. A common problem is for software to fail when run, but when the software engineer uses a debugger to put a breakpoint on a function near the failure the problem disappears, even if no changes are made. The timing of the software and hardware is affected by the breakpoint and the problem disappears. Similarly, in co-verification is it advantageous to avoid intrusiveness caused by the software debugger. Debugger generated memory activity causes two problems:

- The accuracy of the simulation is changed
- Debugger response time can be slow if all debugger-generated memory activity is simulated

If the memory requests generated by the debugger can directly access memory without any simulation, both of the problems are immediately solved. The only remaining hurdle is to describe to the debugger the memory map from the view of the CPU. In a system design, there are many instances of smaller memories that fit into the global CPU address range. Between the CPU and individual memory instances there are memory controllers, decoders, chip-selects, write-enables and other logic that decides how a particular CPU address ends up accessing specific memory instances. An example of how memories in a system may be implemented is shown in Figure 5-3.

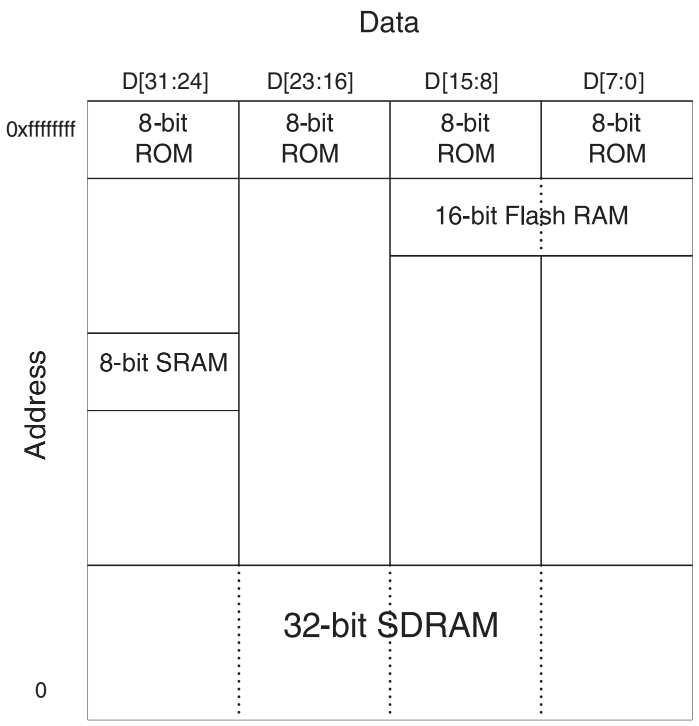


Figure 5-3: Example of memory configuration

A debugger cannot automatically figure out which memories need to be accessed to retrieve the data for a particular CPU address. One way to help it is to allow the user to specify the mapping of the CPU addresses to the individual memory instances. Different tools have different ways to specify the information. Some of the ways it can be done are:

- Text file containing information
- Graphical tool to draw boxes in the address map
- Drag-and-drop from a Verilog hierarchy description
- User can write C functions using provided hooks to implement direct memory access

A text file is the easiest way to specify the configuration information. Figure 5-4 shows an example of specifying memory mapping so the debugger can directly access memories without any simulation to maintain cycle accuracy and provide a fast debugger response.

```
; Global variables defining bus
data_bus_width   32
bit_order = descending
byte_order = littleendian

; Individual Memories in the design

name   TBplatform.uMemory.uBootROM0.Rom
start_address  0x0
start_bit = 0
last_bit  = 7

name   TBplatform.uMemory.uBootROM1.Rom
start_address  0x1
start_bit = 8
last_bit  = 15

name   TBplatform.uMemory.uBootROM2.Rom
start_address  0x2
start_bit = 16
last_bit  = 23

name   TBplatform.uMemory.uBootROM3.Rom
start_address  0x3
start_bit = 24
last_bit  = 31

name TBplatform.uMemory.uRAM6.Ram
start_address 0x28000000
start_bit 0
last_bit 31

name TBplatform.uTrickWrapper.uIntROM.Mem
start_address 0xffff0000
start_bit 0
last_bit 31

name TBplatform.uPlatform.uProcSubSys.uProcCoreMod.uDRAM.mem
start_address 0x40008000
start_bit 0
last_bit 31

name TBplatform.uPlatform.uProcSubSys.uProcCoreMod.uIRAM.mem
start_address 0x40010000
start_bit 0
last_bit 31
```

Figure 5-4: Example memory configuration file

This section talked about using direct memory access for the memory reads and writes caused by the debugger. The next section talks about using direct memory access for memory activity such as instruction fetches or data operations.

Memory Optimizations and Performance

Software engineers always request faster performance. It's interesting that software engineers more naturally talk about performance in terms of slowdown from the real system, and hardware engineers talk about performance in terms of speedup from logic simulation speed. Logic simulation speed ranges from 10 to 500 cycles/second for SoC projects. For hardware engineers, a 10x speedup is useful if it can be achieved with little or no effort, a 100x speedup is worth some effort to achieve, and a 1000x speedup enables a new set of tests to be run that would otherwise have to be skipped since it takes runtimes from days to minutes. For software engineers a 10x slowdown is not a problem, taking a 100 MHz design down to 10 MHz. A 100x slowdown to 1 MHz is probably still useful, but a 1000x slowdown is not useful for many types of software including RTOS and applications.

In co-verification, there are two basic ways to provide faster performance. The first is to simulate less so that from the view of software engineer, the performance appears to be faster, and the second is to speed up the hardware execution engine.

Ways to improve co-verification performance are:

- Simulate less
- Increase raw simulation performance

Achieving higher performance by simulating less can be done using the same direct memory access techniques described above. When the software running on the embedded CPU makes memory accesses, either instruction fetches or data reads and writes, co-verification tools can be configured for logic simulation or simply use direct memory access to complete the read or write. The ability to skip simulation for certain areas of the memory map is sometimes called *optimization*, *optimized memory*, or *memory access optimization*. The ability to bypass simulation to increase performance is also an alternative definition of co-verification. This ability is primarily what differentiates some co-verification methods from pure co-simulation.

Memory optimization is really the only hope for reasonable performance when co-verification is performed using an ISS and logic simulation. Without optimization the simulation will run at the speed of logic simulation (remember that's about 10 to 500 cycles/second). Host-code mode has some amount of memory optimization that occurs naturally since it does not fetch instructions from the logic simulator, only data accesses are simulated. This eliminates much of the simulation that would be performed using target-code mode with an ISS.

Although not all co-verification methods use the dual-process architecture, let's look at the common ISS+BFM setup to get a flavor for co-verification performance. The embedded system memory map is used to decide which memory accesses are optimized and which are simulated. Performance depends on how often software accesses the hardware design. This variable is referred to as *hardware density*. Think of it much like the way an instruction cache works. A high number of code fetches hit the cache and are serviced at very high speeds, and some fetches miss the cache and will take longer to get from memory.

Overall performance is determined by the speed of each environment and the number of memory accesses that are required for each. Following is a simplified example that will help to understand co-verification performance. Assume an ISS can run software at 100,000 instructions per second, and assume that instructions that access the logic simulator run at a speed of 50 instructions per second (ips). Also, assume (for simplicity) that 1 instruction takes 1 clock cycle. For a given design, assume the hardware density is 10%, this means 10% of the program's instructions must access logic simulation. The other 90% of instructions are not simulated due to memory optimization. To run a program with 100,000 instructions, the total execution time is:

> 90,000 instructions * (1 inst / 100k ips) + 10,000 instructions * (1 inst / 50 ips) = 200.9 seconds
>
> To run this 100,000 instruction program exclusively in a logic simulator running at 50 ips would take
>
> 100,000 instructions * (1 inst / 50 ips) = 2,000 seconds
>
> To run this 100,000 instruction program exclusively in the ISS would take
>
> 100,000 instructions * (1 inst / 100k ips) = 1 second

Co-verification achieves 10x performance increase, or a two order of magnitude performance decrease of the ISS speed depending how you view it. Hardware density is highest for lower-level types of software, such as diagnostic software, and will decrease for higher-level types of software such as applications. This type of calculation is nearly identical to those required to determine how much performance improvement can be obtained from simulation acceleration where some of the testbench runs on a workstation and most of the design is mapped into the acceleration system.

Many co-verification solutions promote the benefit of increased simulation performance. This increased performance usually comes with a price, typically simulation detail. When using memory optimizations, there is a performance versus detail trade-off. Ideally, this trade-off is configurable by the user depending on the goals of a particular simulation. Sometimes engineers are interested in performance figures for the target system. Things like cache hit rates, data sizes, and interrupt latency are important. If these issues can be uncovered in a simulation early in the project, the risk of the system design not meeting requirements is minimized. Other times these details are not necessary and performance for software debugging is most important.

Most co-verification tools allow memory optimizations to be changed dynamically during simulation using either a GUI, command line, or even programmatically (calling C functions) in the case of host-code execution.

The benefit of memory optimization is easy to understand. There is no reason for software engineers to execute instruction fetches again and again when the hardware design has been proven to fetch instructions from memory successfully. The elimination of simulating fetches can easily result in a 10x speedup and it keeps software engineers from watching uninteresting activity.

Traditionally, processors used for embedded systems use only a single bus for both instruction and data accesses to and from memory. As we saw in Chapter 3, all ARM7 processors have a single memory bus. Cores with multiple busses have been introduced to meet increasing system performance requirements. The first popular ARM core with multiple buses was the ARM926EJ-S. It contains separate buses for instruction and data accesses. It also contains two more busses for tightly coupled memory (TCM). One of the future questions for co-verification is the effectiveness of memory optimizations as the number of busses on a CPU continues to increase. Memory optimization takes advantage of the fact that memory accesses occur serially on a bus and some can be skipped. If two busses now operate in parallel, the chances of both busses performing an access to optimized memory is much less, especially when many of the data accesses are interesting to simulate.

The second way to achieve faster performance is by speeding up the hardware execution engine. The previous example demonstrated a 10x speedup by using memory optimizations, but this alone is not high enough performance to run long tests with high software content. Using simulation acceleration and in-circuit emulation is a way to improve the speed of the hardware execution engine and increase performance. Co-verification methods using emulation and models such as RTL or testchips may achieve 200k to 400k cycles/sec. This is over 1000x over logic simulation. Of course, this provides a 100% cycle accurate environment, but may have some software debugging limitations.

Simulation acceleration based on an ISS and an acceleration system as the hardware execution engine can additionally provide high-performance co-verification. Let's look at the same example assuming simulation acceleration runs at 10k ips versus the logic simulator of 50 ips. Let's use the same technique for memory optimization.

Again, the program length is 100,000 instructions. The total execution time is:

> 90,000 instructions * (1 inst / 100k ips) + 10,000 instructions * (1 inst / 10k ips) = 1.9 seconds
>
> To run the program exclusively with simulation acceleration running at 10k ips would take 100,000 instructions * (1 inst / 10k ips) = 10 seconds

Recall the ISS alone took 1 second to run the program at 100k ips. If simulation acceleration using an ISS can provide beneficial debugging features beyond what in-circuit emulation or prototyping can provide it can be a valuable technique for co-verification.

We have seen that direct access to simulation memory, whether it be in a logic simulator or simulation acceleration/emulation platform is useful for both interactive debugging and increasing performance. When taking advantage of such techniques the synchronization between multiple processes must also be considered.

Modes of Synchronization

Whenever co-verification uses multiple processes (or threads) such as with an ISS and BFM in a logic simulator, or with host-code execution, there must be synchronization between the processes. The easiest case to consider is an ISS that runs in conjunction with a logic simulator, as discussed in Chapter 4, Figure 4-12. If all memory accesses are simulated in the logic simulator it is easy to imagine the ISS runs and models the internal workings of the CPU and then passes information over to the logic simulator for a bus transaction and then blocks so the simulator can run. The logic simulator will then activate and run the bus transaction to completion. Once the result is available it will be stored somewhere for the ISS, the logic simulator will block, and the ISS will be activated again. Assuming some method to exchange data and a way to activate the other process and block itself this toggling back and forth between ISS and logic simulator will produce a complete simulation of both hardware and software. Interrupts from the hardware can be reported back to the ISS along with the bus transaction results. The reason this synchronization is straightforward is because we know the ISS will always be sending something for the BFM to do, even if it just idle cycles (AHB **HTRANS[1:0]** = 0). This ensures all interrupts will be communicated back to the ISS within a few clocks of when they occur.

Of course, we have seen that running every memory access in a logic simulator is slower than skipping simulation. When co-verification techniques that make use of memory optimization or host-code operation are used, the synchronization gets more complex. In situations where there is no guarantee the two processes will constantly exchange information, there is a possibility for problems. It can occur that the software is waiting for an interrupt from hardware but because it is not making any requests to the logic simulator there is no way for it to get any new interrupt messages from the simulator. This type of deadlock is best shown by example. The example in Figure 5-5 shows some software that sets a timer and then waits until the timer has expired. When an interrupt comes from the timer, the software resumes.

```
;; assembly function to wait for interrupt from timer
EnterPause ROUT
        LDR     R0, =Halt
        LDR     R1, =Pause
        MOV     R2, #4
        STR     R2, [R1]
        STR     R2, [R0]
100     B       %100              ; wait here forever until interrupt
        MOV     PC, R14           ; function return
        EXPORT  EnterPause
        END

/* C funtion to set timer and wait */
void delay (int delay)
{
      *Load    = delay;    /* set up period */
      *Control = (Enable + Periodic + Prescale0);  /* enable */
       EnterPauseMode();   /* wait for interrupt */
      *Control = Disable;  /* disable timer */

}
```

Figure 5-5: Example of synchronization problem

This example may hang if the logic simulator does not have a chance to run long enough for the timer to expire and send the interrupt to software. This can occur if techniques to skip simulation are used to try to increase performance.

Co-verification tools typically allow different modes of synchronization. Without going into too much detail about them, the following modes are commonly used:

- **Lock Step** means the logic simulator will block until a request is ready. After the request is completed, it will block again until the next request.
- **Free Running** means the logic simulator is always running. It will service requests from software as they come in. Test results may not be repeatable using this type of synchronization.

A simple way to think about the modes is by looking at the simulation waveform for each. For lock step, the transactions will be close together with no gap between them. Free running mode produces a waveform with gaps of idle time on the bus since the simulator was running and no transactions were coming from software.

For designs with multiple busses where it can be guaranteed that each software program will continually send bus requests, a multibus lock step mode can be used. If not, a multibus free running mode is better to avoid synchronization problems.

There may also be C API functions that can be used for host-code mode to advance simulation time. In the context of host-code mode, this can be used to provide synchronization which is something between lock step and free running. The simulator can block, but even if no memory access is made a C function can be called periodically to advance the simulation. The only parameter to the function is the number of bus clocks to advance. An easy way to automate the synchronization is to create a separate thread that takes care of the simulation advancement.

Of course, co-verification tools hide as much detail about synchronization as possible, but understanding how synchronization works allows users to better evaluate the benefits of memory optimization and host-code operation.

Interprocess Communication

Interprocess communication (IPC) is used to exchange data between two or more processes on the same or different computers. There are various forms of IPC such as sockets, memory mapped files, shared memory, pipes, and message queues. In co-verification there are two main places IPC is used. Debuggers often connect to CPU models via IPC. Examples are shown in Figure 5-6.

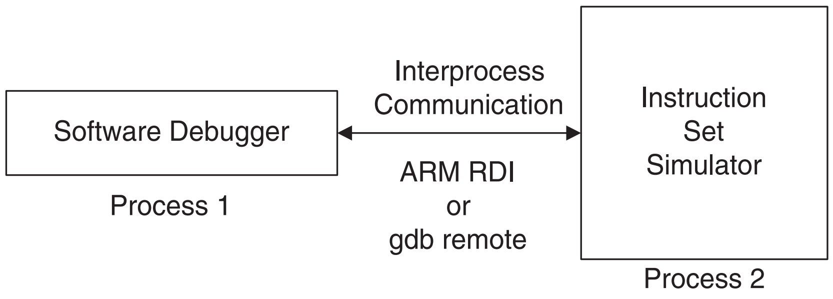


Figure 5-6: IPC from a software debugger to a model

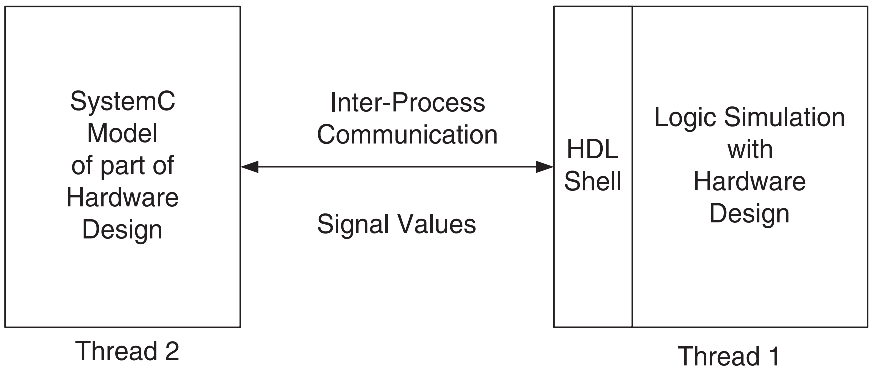

SystemC Model of part of Hardware Design
Inter-Process Communication
Signal Values
HDL Shell
Logic Simulation with Hardware Design
Thread 2
Thread 1

Figure 5-7: Examples of IPC in co-simulation

The second place IPC is used is between different execution engines (simulation kernels) in co-simulation. The most common example is between an ISS and a logic simulator. Another example is between a SystemC model and a logic simulator as shown in Figure 5-7. Other places for IPC in co-verification include the connection between a software debugger and an RTL model, between a hardware modeler and a logic simulator, and between an evaluation board and a logic simulator.

There is no need to give details about how various IPC mechanisms work as there are plenty of resources for engineers looking to learn about IPC and implement solutions. Following is a simple review of the differences between three IPC techniques commonly used in co-verification:

Sockets are used when communication is needed between different computers. Sockets work using IP addresses and can connect multiple computers of different types. The basic process to create a socket and start to receive data from another program is to use three system calls: socket(), bind(), and listen(). The program that creates the socket and listens for another program to send something is the server. The program that wants to communicate with the server is the client and uses socket() and connect(). The beauty of sockets is that they can connect any machines together, even over long distances via the Internet. The most common benefit sockets provide in co-verification is the ability to allow software engineers to run on one type of platform and hardware engineers on another. When

V-CPU was originally developed, the hardware engineers ran logic simulation on HP workstations and the software engineers all coded and tested software on Sun workstations. The socket allowed communication between the native compiled software program on Sun and the Verilog simulation on HP. It is also common for software engineers to work on Windows platforms and want to run debuggers on Windows that will communicate with a hardware execution engine on UNIX or Linux. The drawback of sockets is the performance. In applications where the time required to send and receive data between processes is critical, sockets are not the highest performance IPC method.

Shared memory is another IPC technique used in co-verification to communicate between multiple processes running on the same machine. Shared memory is a special block of memory that is created by the operating system and that allows different processes to map it into their address space. Once multiple processes map it, they use ordinary pointers to access the memory just like memory created with malloc(). Setting up shared memory is done using shmget() and shmat() by each process that wants to use it. Shared memory is faster than sockets when two processes on the same machine need to communicate. It is also easier to use, since it behaves just like regular memory. Processes can unmap or detach from shared memory using shmdt(). Care must be taken to remove shared memory segments with shmctl() when the programs using it terminate. If processes terminate abnormally, the shared memory must be manually removed using the ipcrm command.

Threads are the third IPC method used in co-verification. Threads are not separate processes, but are part of a single process and can execute instructions using their own program counter and stack. Since threads are all part of the same process, they can easily share data within the process. With threads there is no need for shared memory since the data segment is visible from each thread. Threads are the most efficient IPC for co-simulation applications such as an ISS or C model co-simulating with a logic simulator. To use threads requires some control over the source code of the models or applications. For example, if there are two independent applications, each with a main() function, it is not possible to make them run in one process with two threads without modifying the source code around the main() function. Even if the startup can be modified to run in a single process with multiple threads, it may be better to leave the processes separate for other reasons. Multithreaded programs share the same controlling terminal and I/O streams. If each thread needs different keyboard input, multithreading may not be the solution. In co-verification and other co-simulation applications, standard thread libraries such as Posix pthreads or Solaris threads are not used because they are implemented in the operating system kernel. System calls only slow down the performance of the thread library. More often QuickThreads, a toolkit for building threads packages developed at the University of Washington, is used to implement user-level threading. An example is the use of Quick-Threads in OSCI SystemC.

Mixing HDL and C Models

There are times when mixing C models with logic simulation is useful, especially when RTL for some blocks is not available. Most co-verification tools allow a mixture of C and RTL to represent the hardware. There are different approaches to mixing C and HDL simulation. Some approaches are more C focused and consist of a C environment that includes the CPU model and C models working together outside of the logic simulator. The purpose is to use the logic simulator as little as possible, since it has slower performance than the C models. In this case, the C simulation is the master and the logic simulator acts more like a slave that is activated only when necessary by the master. A pin interface or transaction interface is normally used to activate the logic simulator. This type of interface is used when design teams start from a C environment and iteratively move to HDL for each block. Figure 5-8 shows this approach with a pin interface and Figure 5-9 shows it with a transaction interface. The benefit of the transaction interface is performance in hardware execution engines such as simulation acceleration and emulation. These engines run faster if communication is minimized. To increase performance, the bus is “mirrored” in the HDL simulator and transactions are captured from the C simulation and run in the logic simulator using a bus functional model. To target acceleration applications, the bus functional model must be synthesizable.

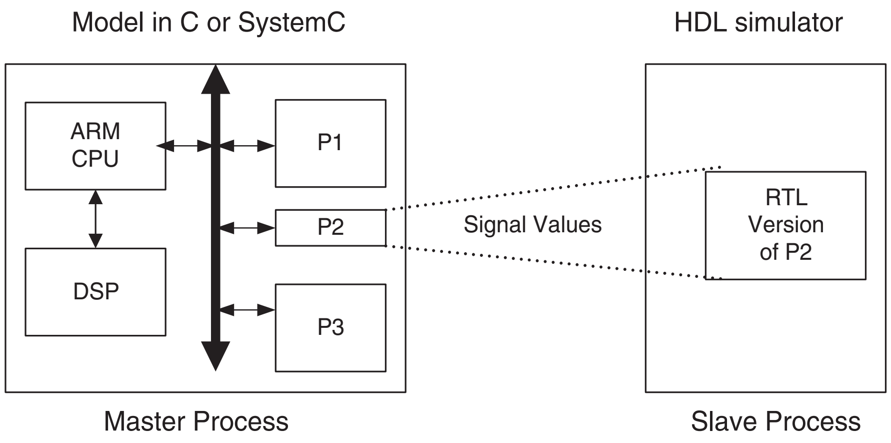


Figure 5-8: C environment with HDL co-simulation at pin level

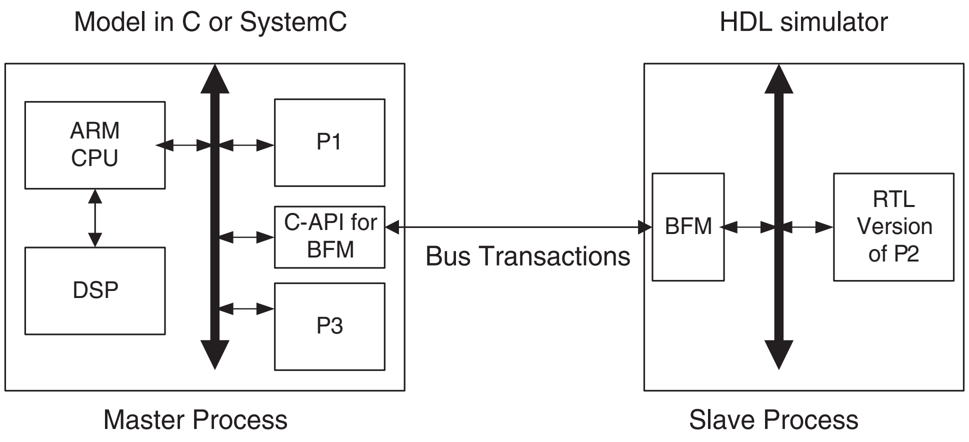


Figure 5-9: Environment with HDL co-simulation at transaction level

The second approach is to co-simulate C more tightly coupled to the logic simulator. Simulators have always provided ways to incorporate C models directly in the simulation process via C interfaces such as Verilog PLI. Simulators now provide ways to co-simulate SystemC directly in the simulation. SystemC has become just another language that is compiled and simulated automatically within the simulation process. With this capability the simulation can mix Verilog, VHDL, and SystemC at any level of hierarchy. This approach may be a bit slower than using C outside of the logic simulation process, but as we discussed previously the synchronization issues are more easily understood inside the context of the logic simulator versus the scheduling and callbacks required when running C outside of the simulator. This approach to co-simulation is well suited for using C and SystemC for modeling specific blocks of the design as well as for testbench purposes. Figure 5-10 shows the classical co-verification setup from Figure 4-12 with the use of a SystemC model being co-simulated with the logic simulator. This design requires that all activity between the CPU and the SystemC model go through the logic simulator. Performance is not as good as the previous approaches, but with SystemC co-simulation built into logic simulators and automatic synchronization, it is very easy to setup and use.

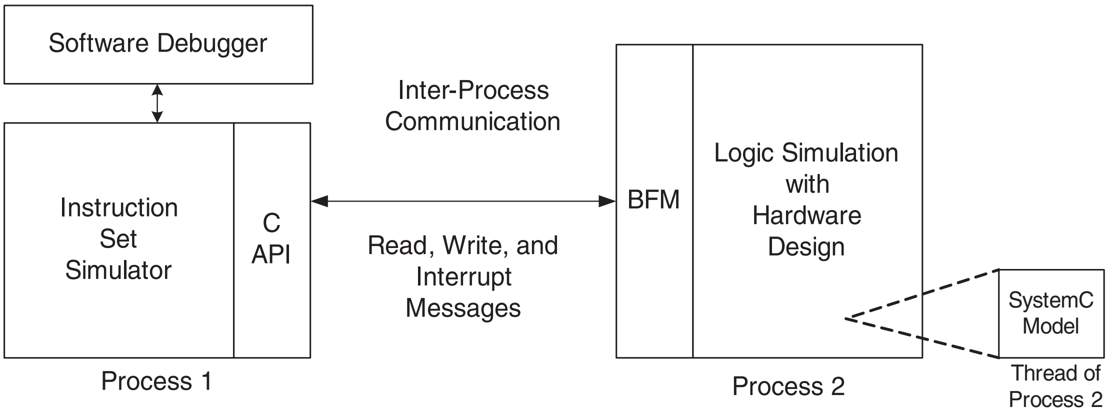

Software Debugger
Instruction Set Simulator
C API
Process 1
Inter-Process Communication
Read, Write, and Interrupt Messages
BFM
Logic Simulation with Hardware Design
Process 2
SystemC Model
Thread of Process 2

Figure 5-10: Co-verification using co-simulation of HDL and SystemC

Implicit Access

One of the benefits of using co-verification with host-code mode is the performance of the software program. A native compiled program running on today's workstations is a few orders of magnitude faster than on an ISS. This gap may continue to grow as embedded processors get more complex and the ISS gets more complex. The main complaint of software engineers about host-code operation is the need to modify the software for workstation compilation versus cross-compilation for the target processor. After all, if the software is not the exact software engineers cannot be sure they verified the right code. In the worst case, every line of software has to be changed and then the software is 100% something else. Early co-verification tools advocated replacing all memory accesses to the embedded hardware with C function calls to activate logic simulation. To make this transition easier, they advocated that all memory accesses from software should call a set of a few common C functions and then the changes needed for co-verification could be confined to just a few functions. Unfortunately, software is not that easy; in C memory accesses are everywhere because of the easy use of pointers. Figure 5-11 show a memory access using pointers and how it should be replaced with a C call for co-verification.

```
#include "cover-api.h"        /* include file with C API definitions */

 void mem_fill(unsigned long addr, unsigned char pattern, unsigned long length)
 {
    unsigned int  i;

    printf("Setting memory buffer to %x\n\n",pattern);

#ifndef COVERIFICATION
    (void) memset((char *) addr, pattern, (size_t) length);*/
#else
    for (i = 0; i < length; i++)
    {
        coverWrite(addr, pattern, 1);   /* explicity write 1 byte */
        addr++;
    }
#endif
 }
```

Figure 5-11: Host-code example of replacing memory accesses with C calls

For many projects, code modifications like this are not practical, and a better solution is needed. The best solution to this problem is to capture the memory accesses automatically and simulate them in the logic simulator without requiring the user to change the software. This automatic capture of memory accesses is called *implicit access*.

Implicit access is possible because embedded software often accesses memory directly by use of pointers in C. When writing software for a workstation that is running an operating system like Windows, UNIX, or Linux a program cannot directly access specific memory. Doing so is dangerous, and poorly written programs will cause the operating system to crash. To avoid crashes, operating systems provide memory protection to stop programs from cashing the OS. Programs must use functions like malloc() to allocate memory and use the returned pointer to access the memory. Figure 5-12 shows the difference between a workstation program accessing memory and an embedded program accessing memory directly.

```
/* On a workstation malloc must be used to create memory */
   unsigned long *ptr;
   size_t         buf_size;
   unsigned long data = 0xabcd1234;

   ptr = (unsigned long *) malloc(buf_size);
   *ptr = data;  /* Write to allocated mem, no idea what the address is */


/* In embedded diagnostics C can directly access hardware */

   unsigned long *ptr = 0xffff0000;
   unsigned long data = 0xabcd1234;

   *ptr = data;  /* Write address 0xffff0000 directly */
```

Figure 5-12: Memory access on workstation vs. embedded

The key to implicit access is what happens if the embedded software in Figure 5-12 is compiled and run on a workstation. Even without actually trying it you know it will not work, because the pointer is set to access an address that is specific to the embedded system. A workstation program cannot directly access this memory because it will be prevented by the operating system's memory protection. Implicit access for co-verification intervenes and captures the embedded style memory access and sends the address, data, and control information to logic simulation to carry out the request. Once the logic simulator carries out the request, implicit access will receive the result and put it into the proper register of the workstation CPU. The software program does not even know what happened or that simulation was used to execute this line of C code.

Implicit access allows large amounts of existing code to be easily ported to a workstation platform for co-verification with limited changes. At a minimum, a software library must be linked with the user's code to enable implicit access to work, and some initialization must be done at startup to configure implicit access, but overall this is very nonintrusive for the user.

One of the most interesting things about implicit access is its implementation. Those interested in computer architecture and instruction sets will find it very interesting. Workstations with RISC processors such as Sun Solaris machines have only a few load and store instructions that access memory. Memory accesses are always between registers and memory. In contrast, Intel processors running either Windows or Linux have a large number of instructions that access memory with more complex addressing modes.

There are two things to be aware of when using implicit access. First, depending on the memory areas to be accessed by the embedded software, there may be some conflicts with the memory space that is accessible by the native compiled program. Consider a 32-bit address space of workstation program. The code for this program must be located in some addresses of the virtual address space so this will prevent some range of addresses from being accessed as data. This issue can be solved by compiler and linker switches to move the program addresses to different locations that are compatible with the embedded system memory map.

The second thing to be aware of are the differences in the instruction set of the workstation versus the embedded processor. This may cause some differences in the memory transactions that are run in simulation. For example, if the embedded CPU is ARM, which allows 1, 2, and 4 byte aligned memory accesses, and the workstation CPU is Sun Solaris which allows 1, 2, 4, and 8 byte aligned memory accesses it is possible for the native compiler to generate a 64-bit read or write that will not occur on the target. Similar differences may be found in the implementation of C library functions such as those dealing with strings. A workstation call to strcpy() may be implemented as a series of 1 byte memory accesses and the corresponding C library function for the embedded processor may use some number of 4 byte memory accesses followed by one to three single byte accesses. Bigger differences exist when the host workstation is an architecture that allows unaligned memory accesses such as PowerPC. When mixing an unaligned host machine with an aligned target, more precautions must be taken to ensure the bus transactions passed to the BFM are legal in the target system. All in all, these differences are minor and don't normally cause any problem for the embedded software engineer using host-code mode for co-verification.

Save and Restart

One of the most commonly asked about co-verification features is the ability to save the state of a simulation and restart it at a later time from the saved point. This function is common in logic simulators, so it is easy to imagine it extended to co-verification. Whether or not save and restart can be done depends on the architecture of the co-verification solution.

Techniques using an ISS typically cannot be saved and restarted, because the ISS does not support such a feature. Technically, it should be possible to implement, but I have seen it implemented only once. To save and restart the ISS requires the internal state of the model including registers, cache, pipeline, and so forth, to be saved to disk for later reloading. Techniques using host-code mode can be saved and restarted since the software is a C program that the user has control of and the co-verification tool can provide API functions to save and restart connections between the host-code program and the logic simulator.

Techniques using actual physical hardware, such as in-circuit emulation, cannot be saved since there is no way to tell a chip to save state to a file. Some wishful thinking related to using JTAG to scan out values may sound interesting but is not possible. Save and restart is possible on a chip (or even ISS) with help from software. Laptops have a feature called *hibernate*, which writes the entire contents of memory to disk and then shuts down. When turned back on it knows to reload the memory and start from where it left off. This requires software to be run on the CPU to take care of saving memory and using whatever power management features are in the CPU to shutdown and restart. For co-verification it doesn't make sense to add this type of software only for simulation purposes.

Co-verification techniques based on RTL models is the best way to do the save and restart feature. Since the CPU model runs in the logic simulator, the feature is automatically available. The key is to be able to reconnect the debugger after the state is restored. The good news is that a debugger does not need to store any state and can be reconnected at any time. Debugger reconnection is easy to demonstrate on a workstation. The debugger can be connected at any time to a running program. Figure 5-13 shows connecting gdb to a logic simulation using the "attach" command. The debugger can also be disconnected using the "detach" command.

The same is true for embedded debuggers. If the connection is gdb to an RTL model, it can be connected at any time using the "target remote" command. If the connection is JTAG via in-circuit emulation then it depends more on the JTAG tool and debugger used. Some JTAG debuggers will try to identify the CPU and immediately issue a reset as a way to take control of the CPU. This will restart the CPU from the reset vector and trash the current state. Other JTAG tools will not issue any reset, just read the registers and figure out the software context. Connecting without requiring any reset allows save and restart to be performed.

```
sp8:7 % ps -ef | grep vlg
   jason 16606 16175  3 08:12:39 pts/3    0:01 vlg
sp8:8 % gdb vlg
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb) attach 16606
Attaching to program `/tools/example/ARM926EJS/vlg', process 16606
0xff1969d8 in _semsys () from /usr/lib/libc.so.1
(gdb) where
#0  0xff1969d8 in _semsys () from /usr/lib/libc.so.1
#1  0xff3647f4 in xSemLock (xShmemHandle=0xff2e0000) at xPli.c:353
#2  0xff364ac4 in xSuspend (xShmemHandle=0xff2e0000) at xPli.c:465
#11 0x00078ae8 in main ()
(gdb) det
Detaching from program: /tools/example/ARM926EJS/vlg LWP 1
 (gdb) q
```

Figure 5-13: Attaching gdb to a running program

An example of when save and restart is useful is for RTOS boot that is the same every time. A wireless project manager once estimated that the operating system for a mobile phone would take about 18 minutes to boot with an emulation system. All of the software engineers needed to test software that starts running after the 18 minute startup. Saving startup time by the use of save and restart would allow software engineers to run many more times per day as they make software changes. There is nothing worse for software development than having to wait a half hour for each new test when software engineers are accustomed to taking only minutes or seconds to run such tests.

Post-Processing Software Debugging Techniques

We have talked about the different expectations of software and hardware engineers in terms of performance. Co-verification tools often promote the fact that hardware engineers continue to work in the familiar simulation environment, and software engineers continue to work with familiar debugging tools and doing interactive debugging just like they normally do. An alternative to promoting "no change" and "familiar environment" is to promote the use of post-processing debugging techniques for software engineers. Hardware engineers have always operated with the mentality that simulation is slow so the best way to cope is to run batch jobs and save log files and waveforms. If there are tests with failures, these can be analyzed while the next test is running. This is the most efficient way to keep the simulators (or emulators) busy and do the manual debugging in parallel. Contrast this to software engineers that have a history of interactive debugging. The way to debug software is to observe the failure, make educated guesses about where to put breakpoints, and iteratively rerun the test, moving the breakpoints around and inspecting the system until the problem is found. A combination of breakpoints and printf() statements will usually find the problem.

Since co-verification often depends on hardware execution engines that are not as fast as real hardware, this greatly slows down the run-debug-run-debug loop. Post-processing analysis of software is an area that holds promise for improved debugging techniques. Today, engineers use crude methods for post-processing debug with logic simulation. Most simulation environments use a bus monitor to print address and data values as they occur on the bus with the simulation timestamp. Many CPU models also allow the software engineer to view registers in a waveform tool. With a combination of the bus monitor log, registers and software listings, it is possible to figure out manually where the software is executing and trace execution. Unfortunately, this is a very tedious process. Figure 5-14 is a message I received from a software engineer trying to trace his software using registers and logfiles from simulation.

```
I seem to be getting aborts or other fatal exits near entry
to main() in dhry_1.c, and my current methodology of
tracking PC through my map/symbol table isn't yielding
perfect results.
```

Figure 5-14: A software engineer trying to debug

Some models and bus monitors have a disassembly feature so they can print out not only the bus address and data activity, but also the assembly language instruction the data represents. While this makes correlation back to software source code a little easier because a software engineer can match the assembly instructions easier than just data values, it still does not provide an automated link back to source code. For software debugging, a good automated link back to the original C or assembly code is valuable to find and fix problems.

Another hardware-centric utility that can be useful is a program that can read waveform files in VCD or Novas FSDB format and print out a log of bus transactions and/or register values from it. For more complex busses with address pipelining such as AHB, we know that it is not that easy for a software engineer or verification engineer to look at the waveform and instantly see the address and data values for the bus transactions that are occurring on the bus. In fact, misreading waveforms can lead to wasted debugging time when incorrect assumptions are made. A utility that can reliably take a waveform and generate a simple log file of bus activity is useful to track software problems. This utility is handy when tests are not run with a regular bus monitor. An example application is for simulation acceleration and emulation. When using an emulation system, performance is the most critical factor. Adding a bus monitor that uses $display statements to a log file every few clocks will slow down overall performance. Most emulation systems have ways to extract waveforms for the design after the test is complete without re-running the test. These waveforms can be read by a utility to convert them into a log file that can drive post-processing software debugging tools.

The next logical evolution for debugging software running on models like the DSM or RTL code for an ARM core is to extend the hardware debugging tools to display software context also. Hardware debugging tools are very good at displaying Verilog and VHDL source code and the values of signals during simulation. They are also good at capturing the main elements of an embedded system required for software debugging, CPU registers and memory contents. Embedded software compilation tools can also be used to find out the line of software that was executing for a particular program counter value automatically by reading the ELF file for the software. This automation gives software and hardware engineers a correlated view of which line of software was executing during a specific simulation timestamp. Some projects end up customizing hardware debugging tools and extending them to provide a software view of the design.

For co-verification methods based on instruction set simulation, an interesting post-processing debugging technique is available that captures bus transactions during a test and replays them using only the ISS. Recall the software engineer's view of the world. The only thing that matters is how the ARM CPU executes the instructions. Consider a scenario where a long diagnostic test has been developed. On a normal logic simulator, the test may run for hours. Even with simulation acceleration, this test may run for a longer time than a software engineer can stand to wait for it. If the test fails, the diagnostic developer will not be too keen on restarting the test, guessing where to put a breakpoint in the code and restarting the test and trying to interactively debug the test. This process of restarting the test and moving the breakpoints will be quite tedious.

As a way to address this problem, the engineer can run a single simulation and save a compressed file that contains the bus transactions at the processor interface. The memory transactions, including address, data, and simulation timestamp along with interrupt information, are "recorded" into this file. After the simulation is complete, software engineers can start the instruction set model and software debugger and "re-run" the software execution sequence. However, this time instead of interacting with hardware simulation, the results are read from the recorded file. This "playback" of the bus interface replicates the exact sequence of software execution as shown in Figure 5-15.

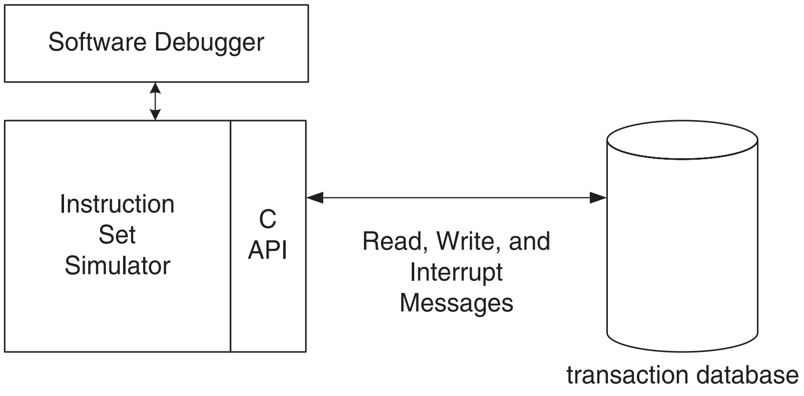
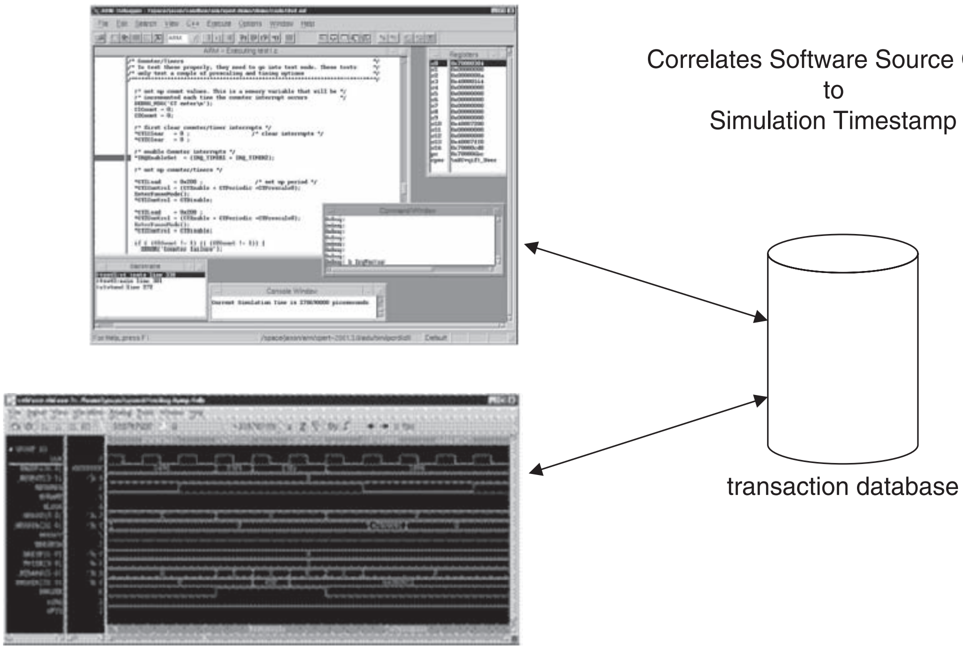

Software Debugger
Instruction Set Simulator
C API
Read, Write, and Interrupt Messages
transaction database

Figure 5-15: Software playback using ISS and recorded bus transactions

Because the simulation now runs at the speed of the standalone ISS, software engineers can re-run the software as many times as needed to find the problem. The simulation timestamp is also provided at any time to help correlate the software and hardware execution. This record and playback methodology is a good way to debug long simulation tests that make interactive debugging unproductive.

For engineers that understand both hardware and software, or when software and hardware engineers want to sit down together and debug, the use of the ISS playback can be enhanced to provide a view of the hardware design as well. Since the simulation timestamps were saved with the stimulus to the ISS the timestamp can be used to automate the hardware view. For hardware engineers, they would like to use waveforms to see what is happening in the design. Hardware debugging tools have a C API that can be used to perform most any command that can be done with the mouse. By using the C API to control the cursor and center it on the screen, a correlated view of both hardware and software synchronized by the timestamp is available. This is the most powerful view of what is happening and it can be done without interactive debugging. Figure 5-16 shows how the debugger and waveform are synchronized by the simulation timestamp. As the software steps, the cursor on the waveform is updated to the new simulation time automatically.

Changing software debugging from interactive to post-processing may not be the easiest change, but it is a co-verification trend that holds promise as designs get larger and interactive debugging becomes more difficult.



```
/* Counter/Timers                                                    */
/* To test these properly, they need to go into test mode. These tests */
/* only test a couple of prescaling and timing options               */
/*-------------------------------------------------------------------*/

  /* set up count values. This is a memory variable that will be */
  /* incremented each time the counter interrupt occurs         */
  DEBUG_MSG("CT enter\n");
  CTCount = 0;
  C2Count = 0;

  /* first clear counter/timer interrupts */
  *CT1Clear   = 0 ;                /* clear interrupts */
  *CT2Clear   = 0 ;

  /* enable Counter interrupts */
  *IRQEnableSet  = (IRQ_TIMER1 + IRQ_TIMER2);

  /* set up counter/timers */

  *CT1Load    = 0x200 ;            /* set up period */
  *CT1Control = (CTEnable + CTPeriodic +CTPrescale0);
  EnterTestMode();
  *CT1Control = CTDisable;

  *CT2Load    = 0x200 ;
  *CT2Control = (CTEnable + CTPeriodic +CTPrescale0);
  EnterTestMode();
  *CT2Control = CTDisable;

  if ( (CTCount != 1) || (C2Count != 1)) {
    ERROR("Counter failure");
```

```
Current Simulation Time is 278690000 picoseconds
```

Figure 5-16: Correlated view of hardware and software

Embedded Software Tool Issues

One of the embedded system issues still unsolved is the incompatibility between tools. Writing workstation software is no different. Anybody that has written software and compiled it with gcc knows that gdb is the best way to debug it. Similarly, a Sun debugger is best to debug a program compiled with the Sun compiler, and the Microsoft debugger is best for a program compiled with a Microsoft compiler. The dependency between compiler and debugger can plague co-verification since only a limited number of debuggers can be supported for a specific solution. Another variable is the embedded operating system. Each operating system supports only a limited number of compilers so the software engineers must use tools the operating system supports.

To best explain this, let's look at an example. All co-verification solutions that support ARM cores naturally support ARM debuggers and ARM compilers. As long as projects use ARM tools there is no problem with efficient co-verification debugging. One of the operating systems gaining popularity for embedded applications is Linux. Due to its free software roots, the Linux kernel includes some assembly language code written in GNU assembler (gas) format, and the C code includes GNU extensions. This means users must use GNU tools to use Linux for an embedded application for an ARM core. In fact, ARM does not recommend anybody even try to use ARM compilation tools for Linux. There are efforts to standardize application binary interfaces and compiler conventions so that different compilation tools can mix and match on Linux, but such solutions will take time to complete and mature. This results in the possibility of there being no software debugger for co-verification tools to debug embedded Linux.

ASIC and ASSP projects often must support many different operating systems depending on which one the system design company wants to use and what are the de facto standards for a particular application. I once met a team of software engineers designing a multimedia chip with an embedded ARM. They handed me the following list of environments various software engineers were working with:

- Support for VxWorks and Tornado (based on gcc 3.2)
- Support for Linux (compiled with gcc 2.95)
- Support for pSOS 2.5 and Diab compiler
- Support for diagnostics compiled with ARM compiler

Maybe someday all of these things will be compatible and software engineers can easily change tools without compatibility problems, but for now it pays to pay attention.

Debugging Co-Verification Issues

Co-verification environments can be difficult to debug when things go wrong. Based on the advanced topics in this chapter and the difficulty of creating models that are not derived from the microprocessor design database as we discussed in Chapter 4, it is clear there are many variables involved. One of the knocks against co-verification is that it is difficult to use. The difficulty stems more from the fact that when something goes wrong it is not easy to find out what it is. The problem may or may not be related to co-verification. We have discussed how modeling differences may cause a test to fail, even though the model matches the specification of the CPU bus protocol. We have discussed the connections between software, the embedded system memory map, and the logic simulation memory models. An application engineer once e-mailed to tell me a co-verification model was not matching the results of the ARM DSM. After repeated efforts to blame the CPU model, the problem turned out to be something wrong in a memory model that was only exposed by the co-verification model. We talked about how synchronization issues can cause tests to fail because of timing issues. The list of things that can go wrong is long. The only way to avoid such pitfalls is to become educated in how co-verification works and try to introduce new variables in a logical way. As happens with most engineering projects, changing too many variables at once is cause for confusion when things go wrong.

This page intentionally left blank

CHAPTER 6

Hardware Verification Environment and Co-Verification

The hardware verification environment and testbenches are closely related to co-verification. To verify an SoC takes a combination of tools and techniques that must work together to be effective. This chapter discusses the relationship between co-verification and the hardware verification environment. The first part of the chapter talks about passive verification activities that aid debugging and monitoring of the design and the latter part talks about active techniques that provide stimulus to create activity in the design. Since the focus is on co-verification, the discussion is primarily related to the microprocessor bus interface. The concept of software verification is also introduced.

Bus Monitor

One of the most basic things that nearly every design uses is a bus monitor simply to display bus activity as it occurs. It is best to separate the monitor from any specific CPU or bus functional models that will be used so that models can easily be interchanged during different phases of verification. A simple example of an AHB monitor is shown in Figure 6-1.

```
/*
 * Synthesizable AHB Bus Montor
 *
 * This monitor works with Xcite and Xtreme and monitors AHB activity.
 *
 * The following features are implemented:
 *
 * 1. Display Address and Data information for bus activity
 *    When all accesses are not required the peek feature can be used
 *    to selectively display activity.
 *
 * 2. Provide Address and Data breakpoints to stop simulation based on
 *    given address and read/write data values.
 *
 *    To enable the monitor to display all address and data information
 *    use +EN_AHB_MON on the vlg command line
 *
 *    To use peek capability: if you do not want to monitor all the time
 *      use  +EN_AHB_PEEK, which turns on monitoring for some
 *      cycles (ahb_peek_duration) every few cycles (ahb_peek_interval)
 *      (both numbers can be changed on command-line).
 *
 *    To use simple address and data breakpoints:
 *      To use address breakpoint:
 *        from CLI:
 *          $scope(<set to ahbmon scope>);
 *          #1 $stop;.
 *          ab_v=<addr>; // set address breakpoint value
 *          ab_m=<mask>; // mask, 1  means that bits is not compared.
 *          ab_en=1; // enables the address breakpoint.
 *      To use read data breakpoint:
 *        from CLI:
 *          $scope(<set to ahbmon scope>);
 *          #1 $stop;.
 *          rdb_v=<addr>; // set read data breakpoint value
 *          rdb_m=<mask>; // mask, 1  means that bits is not compared.
 *          rdb_en=1; // enables the read data breakpoint.
 *      To use write data breakpoint:
 *        from CLI:
 *          $scope(<set to ahbmon scope>);
 *          #1 $stop;.
 *          wdb_v=<addr>; // set write data breakpoint value
 *          wdb_m=<mask>; // mask, 1  means that bits is not compared.
 *          wdb_en=1; // enables the write data breakpoint.
 *
 *      Note that, the you will need to do #1 so as not to race
 *      with the intial block in the code for the monitor.
 *
 *      For simulations with multiple monitors use the NAME parameter to
 *      easily identify which messages below to which AHB interface.
 *
 *
 */
```

```
module ahbmon (
    HCLK,
    HRESETn,
    HGRANT,
    HADDR,
    HTRANS,
    HWRITE,
    HSIZE,
    HBURST,
    HPROT,
    HREADY,
    HRESP,
    HRDATA,
    HWDATA
);

    parameter DATA_BUS_WIDTH = 32;
    parameter NAME = "AHB";

    input                           HCLK;
    input                           HRESETn;
    input                           HGRANT;
    input [31:0]                    HADDR;
    input [1:0]                     HTRANS;
    input                           HWRITE;
    input [2:0]                     HSIZE;
    input [2:0]                     HBURST;
    input [3:0]                     HPROT;
    input                           HREADY;
    input [1:0]                     HRESP;
    input [DATA_BUS_WIDTH-1:0]      HRDATA;
    input [DATA_BUS_WIDTH-1:0]      HWDATA;

    parameter IDLE                    = 2'b00;
    parameter BUSY                    = 2'b01;
    parameter NONSEQUENTIAL           = 2'b10;
    parameter SEQUENTIAL              = 2'b11;

    parameter READ                    = 1'b0;
    parameter WRITE                   = 1'b1;

    parameter SINGLE                  = 3'b000;
    parameter INCR                    = 3'b001;
    parameter WRAP4                   = 3'b010;
    parameter INCR4                   = 3'b011;
    parameter WRAP8                   = 3'b100;
    parameter INCR8                   = 3'b101;
    parameter WRAP16                  = 3'b110;
    parameter INCR16                  = 3'b111;
parameter OKAY                  = 2'b00;
parameter ERROR                 = 2'b01;
parameter RETRY                 = 2'b10;
parameter SPLIT                 = 2'b11;

parameter BYTE                  = 3'b000;
parameter HALFWORD              = 3'b001;
parameter WORD                  = 3'b010;
parameter DOUBLEWORD            = 3'b011;
parameter WORD_LINE_4           = 3'b100;
parameter WORD_LINE_8           = 3'b101;
parameter BITS_512              = 3'b110;
parameter BITS_1024             = 3'b111;

parameter IDLE_STATE     = 4'h0;
parameter ADDR_STATE     = 4'h1;
parameter DATA_STATE     = 4'h2;
parameter ERROR_STATE    = 4'h3;
parameter RETRY_STATE    = 4'h4;
parameter SPLIT_STATE    = 4'h5;
parameter UNKNOWN_STATE  = 4'h6;

reg [3:0]    state;
reg          wait_reg;

reg          latched_hready;

reg [1:0]    latched_htrans;

reg [31:0]   latched_haddr;
reg          latched_hwrite;
reg [2:0]    latched_hsize;
reg [2:0]    latched_hburst;
reg [3:0]    latched_hprot;

reg          HGRANT_d;

reg [4:0]    beat_count;
reg          burst_in_progress;

reg [31:0]   error_retry_or_split_address;
reg          retry_pending;
reg          split_pending;

reg          en_ahb_mon;  // enable AHB monitor

reg          ab_en,       // address breakpoint enable
             rdb_en,      // read data breakpoint enable
             wdb_en,      // write data breakpoint enable
             ab_hit,      // address breakpoint hit
             rdb_hit,     // read data breakpoint hit;
             wdb_hit;     // write data breakpoint hit;
```

```
  reg [31:0]    ab_v, rdb_v, wdb_v; // address value, read data value, write
data value
  reg [31:0]    ab_m, rdb_m, wdb_m; // address mask, read data mask, write
data mask

  wire          trig_bkpt_stop;

  reg           trig_addr_disp, trig_reset_disp, trig_data_disp;


  reg           en_ahb_peek;
  reg           peek_on;
  integer       ahb_peek_interval;
  integer       ahb_peek_duration;

  reg [13*8:1]  htrans_name;
  always @(latched_htrans) begin
    case (latched_htrans)
      IDLE          : htrans_name = "idle";
      BUSY          : htrans_name = "busy";
      NONSEQUENTIAL : htrans_name = "nonsequential";
      SEQUENTIAL    : htrans_name = "sequential";
      default       : htrans_name = "unknown";
    endcase
  end

  reg [7*8:1] hwrite_name;
  always @(latched_hwrite) begin
    case (latched_hwrite)
      READ     : hwrite_name = "read";
      WRITE    : hwrite_name = "write";
      default  : hwrite_name = "unknown";
    endcase
  end

  reg [7*8:1] hburst_name;
  always @(latched_hburst) begin
    case (latched_hburst)
      SINGLE    : hburst_name = "single";
      INCR      : hburst_name = "incr";
      WRAP4     : hburst_name = "wrap4";
      INCR4     : hburst_name = "incr4";
      WRAP8     : hburst_name = "wrap8";
      INCR8     : hburst_name = "incr8";
      WRAP16    : hburst_name = "wrap16";
      INCR16    : hburst_name = "incr16";
      default   : hburst_name = "unknown";
    endcase
  end
```

```
reg [7*8:1] hresp_name;
always @(HRESP) begin
  case (HRESP)
    OKAY     : hresp_name = "okay";
    ERROR    : hresp_name = "error";
    RETRY    : hresp_name = "retry";
    SPLIT    : hresp_name = "split";
    default  : hresp_name = "unknown";
  endcase
end

reg [8*8:1] hsize_name;
always @(latched_hsize) begin
  case (latched_hsize)
    BYTE           : hsize_name = "byte";
    HALFWORD       : hsize_name = "half";
    WORD           : hsize_name = "word";
    DOUBLEWORD     : hsize_name = "64bits";
    WORD_LINE_4    : hsize_name = "128bits";
    WORD_LINE_8    : hsize_name = "256bits";
    BITS_512       : hsize_name = "512bits";
    BITS_1024      : hsize_name = "1024bits";
    default        : hsize_name = "unknown";
  endcase
end

/*
** Initializaton
*/
initial begin
    $timeformat(-9,3," ns");

    $display("AHB monitor instantiated at %m with name %s",NAME);
    state = IDLE_STATE;

    /*
    ** Monitor enable for all address and data
    */
    if($test$plusargs("EN_AHB_MON")) begin
      $display("%0t: [%s] monitor enabled",$time,NAME);
      en_ahb_mon = 1;
    end
    else begin
      $display("%0t: [%s] monitor disabled",$time,NAME);
      en_ahb_mon = 0;
    end

    /*
    ** Peek enable
    */
    if($test$plusargs("EN_AHB_PEEK"))
```

```
        en_ahb_peek = 1;
      else
        en_ahb_peek = 0;

    ahb_peek_interval = 10000; // Peek every 1000 clocks
    ahb_peek_duration = 500;   // Peek for 500 clocks

    ab_en = 0;
    rdb_en = 0;
    wdb_en = 0;
    ab_hit = 0;
    rdb_hit = 0;
    wdb_hit = 0;
    ab_m = 32'h0;
    rdb_m = 32'h0;
    wdb_m = 32'h0;
    trig_addr_disp = 0;
    trig_data_disp = 0;
    trig_reset_disp = 0;
    wait_reg = 0;
end


always @(posedge HCLK)
  HGRANT_d <= HGRANT;

always @(posedge HCLK or negedge HRESETn)
  begin
    if(!HRESETn) begin
       state <= IDLE_STATE;
    end
    else
       begin
         if (HRESETn && HTRANS[1] && HREADY && HGRANT_d)
         begin
           if (en_ahb_mon | peek_on)  // display address now
             trig_addr_disp = ~trig_addr_disp;

           // Address breakpoint detection
           if(!ab_hit && ab_en && ((HADDR & ~ab_m) == (ab_v & ~ab_m)))
             ab_hit <= 1;
           else
             ab_hit <= 0;

           latched_haddr <= HADDR;
           latched_hwrite <= HWRITE;
           latched_hsize <= HSIZE;
           latched_hburst <= HBURST;
           latched_hprot <= HPROT;
           latched_htrans <= HTRANS;
```

```
            state <= DATA_STATE;

          end
          else
            state <= IDLE_STATE;

        end

    end

  always @(posedge HCLK)
    begin
        if ((state == DATA_STATE) || wait_reg)
        begin
          if (HREADY)
          begin
            if (en_ahb_mon | peek_on) // display data info now
              trig_data_disp = ~trig_data_disp;

            if(!rdb_hit && rdb_en && ((HRDATA & ~rdb_m) == (rdb_v & ~rdb_m)))
              rdb_hit <= 1;
            else
              rdb_hit <= 0;

            if(!wdb_hit && wdb_en && ((HWDATA & ~wdb_m) == (wdb_v & ~wdb_m)))
              wdb_hit <= 1;
            else
              wdb_hit <= 0;

            wait_reg = 0;
          end
          else wait_reg = 1;
        end
    end

   // peek logic if you do not want to monitor every transaction
   integer peek_counter;
   always@(posedge HCLK or negedge HRESETn)
     begin
     if(!HRESETn)
       begin
         peek_on <= 0;
         peek_counter <= 0;
       end
     else
       if(en_ahb_peek)
         begin
           if (peek_counter == 0)
           begin
             // reset counter to peek interval
             peek_counter <= (ahb_peek_interval + ahb_peek_duration);
           end
           else if(peek_counter <= (ahb_peek_interval + ahb_peek_duration) &&
peek_counter > ahb_peek_duration)
           begin
             peek_counter <= peek_counter-1;
             peek_on <= 0;
           end
           else if (peek_counter <= ahb_peek_duration)
           begin
             peek_counter <= peek_counter-1;
             peek_on <= 1;
           end
         end
       end

  always @(negedge HRESETn or posedge HRESETn)

    begin
      if (en_ahb_mon | peek_on)
        trig_reset_disp = ~trig_reset_disp;
    end

  assign trig_bkpt_stop = ab_hit | rdb_hit | wdb_hit;
  axis_tbcall(trig_bkpt_stop,"bkpt_stop");

  axis_pulse (trig_data_disp_p,trig_data_disp);
  axis_tbcall (trig_data_disp_p,"data_disp");

  axis_pulse (trig_addr_disp_p,trig_addr_disp);
  axis_tbcall (trig_addr_disp_p,"addr_disp");

  axis_pulse (trig_reset_disp_p, trig_reset_disp);
  axis_tbcall (trig_reset_disp_p, "reset_disp");

  task reset_disp;
    begin
        $display("%0t: [%s] RESET changed:%b", $time,NAME,HRESETn);
    end
  endtask

  task addr_disp;
    begin
      $display("%0t: [%s-ADDR] 0x%h  %s %s
%s",$time,NAME,latched_haddr,htrans_name,hsize_name,hwrite_name);
    end
  endtask

  task data_disp;
    begin
      if (latched_hwrite)
        $display("%0t: [%s-WDATA] write data 0x%h",$time,NAME,HWDATA);
      else
        $display("%0t: [%s-RDATA] read data 0x%h",$time,NAME,HRDATA);
    end
  endtask

  task bkpt_stop;
    begin
      if(ab_hit)
        begin
          $display("%0t: [%s-Break] Stop at Addr Breakpoint HADDR=%x
Mask=%x",$time,NAME,HADDR,ab_m);
        end
      if(rdb_hit)
        begin
          $display("%0t: [%s-Break] Stop at Read Data Breakpoint HRDATA=%x
Mask=%x",$time,NAME,HRDATA,rdb_m);
        end
      if(wdb_hit)
        begin
          $display("%0t: [%s-Break] Stop at Write Data Breakpoint WRDATA=%x
Mask=%x",$time,NAME,HWDATA,wdb_m);
        end
      $stop;
    end
  endtask

endmodule
```

Figure 6-1: Simple AHB monitor code

The monitor prints basic address, data and cycle information about each transaction on the AHB interface. It also has some nice features to allow for address and data breakpoints that allow the simulation to be stopped when a desired address or data value is hit. This monitor is also beneficial because it can be used with a logic simulator as well as with simulation acceleration and emulation. All the checking code is synthesizable and a special callback mechanism is used to display information on the screen and to the simulation log file even when running in emulation mode.

The bus monitor is also useful when debugging software with an interactive debugger. If the monitor is enabled and the software engineer is single stepping through code or displaying memory data it is easy to see what is happening on the bus as a result of software debugging commands.

Protocol Checking

The monitor is useful for tracking execution of a simulation and for debugging, but offers no checking to make sure the bus agents are cooperating and following the rules of the bus protocol. Some bus monitors also provide protocol checking to indicate when a violation occurs. There are many different checks that can be made on the bus protocol, but two are listed below to get the idea of a monitor doing protocol checking.

Aligned Addresses

According to the data bus section of the AMBA AHB specification, all data transfers must be aligned to the address boundary equal to the size of the transfer. This means for word transfers **HADDR[1:0]** must be 2'b00, for half-word transfers **HADDR[0]** must be 1'b0. An example of how to check for aligned addresses is shown in Figure 6-2.

```
// Address alignment check for 32-bit bus
case (HSIZE[1:0])
        2'b01:
          if (HADDR[0] != 1'b0)
            $display("Misaligned Address for HSIZE %h at HADDR: %h at TIME: %t",
HSIZE, HADDR, $time);
        3'b10:
          if (HADDR[1:0] != 2'b00)
            $display("Misaligned Address for HSIZE %h at HADDR: %h at TIME: %t",
HSIZE, HADDR, $time);
        default: ;   // do nothing
endcase
```

Figure 6-2: Address alignment check

Issuing Idle Transfers

Another example is a check to make sure the master issues IDLE transfers correctly. It is possible that a master can be granted the bus when it is not requesting it. This may occur when no masters are requesting the bus and the arbiter grants access to a default master. Therefore, it is important that if a master does not require access to the bus it drives the transfer type **HTRANS** to indicate an IDLE transfer. A simple check is shown in Figure 6-3.

```
if (State == `GRANTED)
      begin
        if (!(HTRANS == 2'b00))
          $display("Master is not driving HTRANS to IDLE when granted at time:
                    %t", $time);
      end
```

Figure 6-3: Master issues IDLE when granted the bus

Looking at the fragments in Figures 6-2 and 6-3, it is clear that checking for bus protocol violations requires a state machine to track the advancement of arbitration, address, and data phases. This tracking state machine is similar to the state machine needed for the design of a master or target bus agent. While this can be done using RTL code and $display statements, engineers pursued a better way compared to the work required to implement such a monitor. It is also tempting to cut out parts of a master or target design and turn it into a monitor, but errors may not be found if the same engineer designed the bus interfaces and the monitor. The evolution of such monitors led to the development of assertions, a more concise way to monitor not only the bus protocol, but all of the design intent including internal design operation.

Assertions

Some of the concepts behind the motivation for assertions were discussed in Chapter 2. This section gives more definitions and describes some of the approaches used to implement assertions. The relationship between assertions in simulation is also extended to simulation acceleration and emulation.

Assertion Definitions

To understand more details of an assertion-based methodology, a common set of definitions is presented.

Property: A general behavioral attribute used to characterize a design. Properties can be high-level attributes such as characteristics of incoming and outgoing networking packets or low-level attributes related to the state encoding of a finite state machine (FSM).

Event: An occurrence of a desirable behavior. Observing events is a required part of verification to ensure completeness. Measuring the occurrence of events leads to quantitative data about specific corner cases and other properties of the design have been verified. Statistics about events lead to functional coverage metrics.

Assertion: An assertion is a statement about a specific property that is expected to be true for the design. Assertions are usually used to trap undesirable behavior. Assertions are checkers and monitors used to enforce rules and assumptions about the design.

Static: An event or assertion that is true for all time or is only checked at a specific instance of time. No knowledge of previous history of the design state is required.

Temporal: An event or assertion that spans a sequence of time. History is required to track the sequence over time.

Procedural Assertion: An assertion that is described within the context of an executing process or set of sequential statements such as a VHDL process or a Verilog always block. The assertion will be evaluated based on the path taken through the set of sequential statements.

Declarative Assertion: An assertion that exists within the structural context of the design. It is evaluated along with all of the other structural elements in the design. For example, a module that takes the form of a structural instantiation.

Regular Expression: A regular expression is a way to express how a computer program should look for a specified pattern and how to react when matching patterns are found. A common use of regular expressions is found in the UNIX grep tool. Specification of properties is easily done by the use of regular expressions. This explains why languages like PERL are used in design verification applications.

Property Language, Declarative Language, Assertion Language, Formal Property Language: All used interchangeably to describe a language that can be used to describe high-level design specification, properties, events, and assertions. These languages are designed to provide a concise format for complex temporal sequences and regular expressions.

Assertion Approaches

Five approaches to implementing assertions are described below. Assertions are still a new area that is changing rapidly as languages and tools are developed and standardized, but this section gives a good overview of some of the different ways that assertions have been used so far.

- Declarative assertions using a library of Verilog monitor modules
- Procedural assertions using a Verilog assert construct
- Formal property languages
- Pseudo-comment directives
- Post-processing simulation history

As of this writing, the various theories about which methodology is best for assertions are fragmented. The goal here is not to promote one versus another, but to introduce each and list some of the commonly documented pros and cons. The examples used are not meant to be comprehensive, but they do cover some of the more widely discussed methods.

Declarative Assertions

The most common use of assertions today are the declarative assertions in the open verification library (OVL) that is freely available at *www.verificationlib.org*. OVL is an assertion monitor library of Verilog and VHDL modules that can be easily instantiated into a design. OVL provides a consistent way to specify static and temporal assertions in RTL code. OVL provides a unified message reporting mechanism that can be easily customized for specific projects by changing very little code. It also provides an easy way to enable and disable assertions during simulation. OVL also provides a consistent severity level scheme that can be used to stop simulation on fatal errors. Work on OVL is continuing with approximately two new releases per year. OVL will continue to evolve as related standards stabilize. OVL is very easy to use and is a good first step to get started with assertions. It is available now and has been used on many projects. Its open source format makes it appealing since it can be customized for each application. Since OVL uses a library of modules, the assertions must be instantiated into the design. The one capability it does not currently have is procedural assertions or sometimes called "in-context" assertions.

An example assertion is shown in Figure 6-4. The assertion is required to check if one of four chip selects is enabled in a five-clock window of time. This assertion can be coded using the assert_change module from OVL as shown in Figure 6-5.

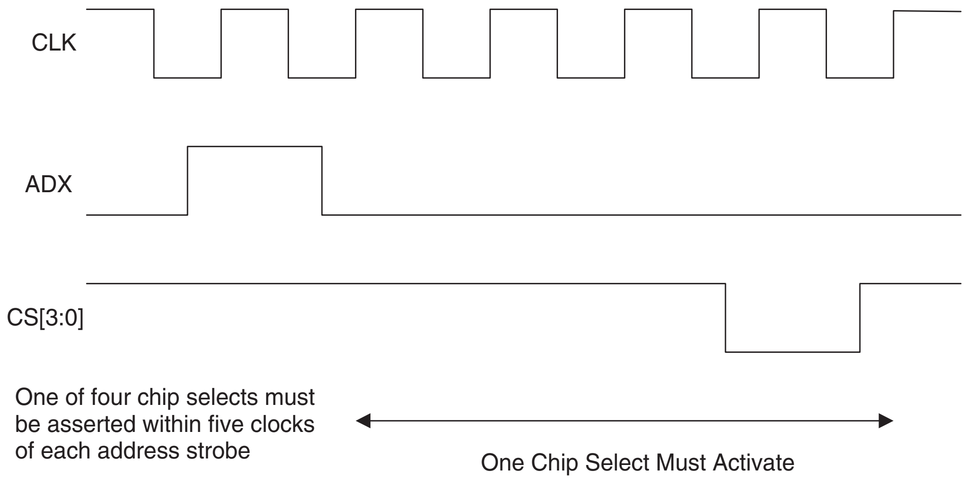

CLK
ADX
CS[3:0]
One of four chip selects must be asserted within five clocks of each address strobe
One Chip Select Must Activate

Figure 6-4: Assertion example

```
/*
** OVL Assertion to make sure a chip select responds for all addresses
*/
wire ce; assign ce =& cs[3:0];
defparam a0.num_cks = 5;
defparam a0.severity_level = 1;
defparam a0.msg = "No Chip Enable";
assert_change a0(clk, ~reset, adx, ~ce);
```

Figure 6-5: OVL implementation

Procedural Assertions

Instead of instantiating a module from a library, it is at times more convenient to specify assertions using procedural statements. This is the case with the assert construct that is part of the SystemVerilog specification developed by Accellera. Procedural assertions are useful in the context of a Verilog always block with procedural code such as a case statement or an if-then-else block. Procedural assertions can be inserted into the code depending on which branch has been taken. This allows the assertions to be active during the context in which they are important. Both declarative and procedural assertions are good ways to capture design intent during the RTL coding process. If they are not captured during this phase, the knowledge is probably lost, since it is unlikely anybody will go back and insert these assertions.

Formal Property Language

The next method that has been used for assertions is the use of a formal property language. These languages are constructed for the purpose of specifying design properties with minimum effort. They are very powerful in creating complex temporal expressions and they also make use of regular expressions to allow complex behavior to be specified with very little code. These languages have existed for many years, but have not been used in mainstream design. Current examples are PSL (property specification language) that is being developed by the Accellera formal verification committee, and the open vera assertions (OVA) promoted by Synopsys for formal verification tools as well as in the VCS logic simulator. The formal property language is useful during all phases of the project and at all levels of design. System architects can use these languages to specify high-level properties of the design. Properties can be used by verification engineers to perform black-box verification without understanding all of the details of the design, and by RTL designers to specify low-level assertions about the code. One of the points that is confusing so far about assertions is the relationship between the formal property language and procedural assertions such as the SystemVerilog assert language construct. Certainly, there is much overlap since both can be used to specify assertions. The formal property language is more general purpose, not tied to any specific language issues of Verilog or VHDL. Over time, the syntax used to specify properties will converge so that a single syntax can work for both a formal property language and HDL language extensions. If this is not possible due to constraints of the Verilog namespace, at least the syntax should be very close. Currently, tools using a formal property language for assertions usually put them into a separate file or as comments in an HDL file so formal tools and simulators can process them separately.

Pseudo-Comment Directives

Another approach that has been taken to specify assertions is the use of pseudo-comments. By embedding assertions in comments, they can be put directly into the RTL code and will not interfere with the Verilog syntax or require any changes to the simulator. Formal verification tools can read the comments using a special parser. In addition to formal methods, these tools can also output a Verilog RTL equivalent for each assertion that allows the assertions to be simulated and flagged in a standard logic simulator. This instrumentation process is useful since it can automatically gather functional coverage metrics about events and assertions during a simulation run, create a database of activity, and display the activity in a concise format for users. Data from multiple simulations can even be merged to form a complete picture of functional coverage. The methodology is similar to the instrumentation process commonly used in code coverage.

Post-Processing Simulation History

So far, all of the methods described have done assertion checking during simulation. A different approach is to check for assertions after a simulation test is complete. In the same way engineers use waveform dumps to debug a problem after the simulation is complete, they can check for events and assertions. Tools have been developed that can read a waveform file in VCD or some proprietary format, read a set of assertions specified using a formal property language, and provide information about the assertions during the simulation run. This methodology does not require any changes to the simulator, any language extensions, or any changes to the design files. Depending on the number of signals needed and the length of the simulation, it could require large waveform files.

Assertions for Simulation Acceleration and Emulation

One of the key issues in design verification is to develop methodology that provides consistency across multiple types of verification platforms. There are two areas where this consistency is important, assertions and testbenches. In order to take advantage of multiple platforms, consider the situation where logic simulation is used in conjunction with simulation acceleration and emulation. Following are the requirements for assertions to work well with both platforms:

- Enable a consistent methodology between simulation, acceleration, and emulation.
- Do assertion detection inside the acceleration/emulation system to obtain highest simulation performance.
- Provide the ability to run behavioral code such as $display and PLI for assertion processing.

As designs get larger and larger, they will require some form of acceleration and/or emulation to achieve verification goals. By enabling faster simulation that works well with assertions, users can leverage both of these emerging technologies to meet project goals. Historically, the value of emulation was raw performance. Emulation users have suffered greatly from poor visibility during emulation. Unlike simulation, emulation does not allow a testbench and monitors to report activity in the emulator. Assertions are a way to provide this visibility. An efficient implementation of assertions maintains the performance value of emulation and provides the visibility to significantly improve emulation debugging. Certainly running faster is helpful, but if the engineers have to remove the assertions during emulation this is a major loss of all of the benefits of assertions already discussed.

Assertions can be decomposed into two parts, detection and failure processing. Assertion detection is done using state machines. All of the temporal expressions that are used to write an assertion can be implemented in the form of an RTL state machine. Since detection is done by state machines, an accelerator/emulator can run all assertion detection in at hardware speed. Similar to a microprocessor interrupt, assertion failures can trigger an interrupt and activate a software service routine that runs on the workstation and has access to behavioral code such as $display or PLI. This type of assertion processing allows engineers to maintain the use of assertions from simulation to simulation acceleration and into emulation without being forced to remove them or execute assertion detection in software simulation. Assertions become a valuable debugging tool in the historically difficult to debug emulation environment.

The interrupt driven implementation of assertions processing is more efficient than polling for assertion violations from a software testbench and does not impact emulation speed when there are no assertion violations. Polling requires many bits inside the emulator to be read every clock cycle. If the design contains 10,000 assertions then an equal number of bits must be read every clock and the bits have to be examined to find out if any assertions have been triggered and if so then the appropriate assertion processing code is executed. The interrupt driven architecture provides the required visibility and the performance of assertion detection in hardware. For fatal assertions, control is required to stop the simulation instead of wasting simulation cycles. This can be done from the software service routine using ordinary Verilog commands such as $finish.

Engineers achieve the highest debugging productivity by combining assertions with waveform debugging. Assertions pinpoint design errors as early as possible and enable engineers to find the source of the trouble quickly using waveform and other hardware debugging tools.

Testbenches Using Bus Functional Models

We have discussed how the bus functional model (BFM) is used as a way to generate stimulus on the microprocessor bus. We have seen it used for co-verification and by hardware engineers that are not interested in the internal details of the microprocessor, but only the activity that is occurring on the bus. When the BFM is used by hardware engineers to verify bus operation, a testbench must be created to activate the bus functional model. As we will see, the model is only a small part of the overall verification environment. In this section, different ways of generating stimulus for the bus functional model as well as ways to measure the completeness or coverage of the stimulus are explained.

Directed Tests

The most straightforward way to use a bus functional model is to write a test that performs a specific sequence of bus transactions. The interface to a bus functional model is normally a set of tasks or functions that generate read or write transactions on the bus. An example task for a fictitious bus read is shown in Figure 6-6.

```
//
// Bus Read Task
//
task do_read;
   input [31:0] a;
   output [31:0] d;
   begin
      address = a;

      // set some bus control signals
      startn = 1'b0;

      @(posedge clk);
      while (rdyn != 1'b0)  // wait for ready
         @(posedge clk);

      d = data;  // capture data from bus

      @(posedge clk);
      address = 32'bz;  // stop driving address and control
      startn = 1'bz;

   end
endtask
```

Figure 6-6: BFM task interface

Once a BFM is defined, directed tests can be written by creating sequences of task or function calls as shown in Figure 6-7 where a loop is used to read many addresses on the bus.

```
for (i = 0; i < 5000; i = i + 1)
begin
   a = i * 21'h100;
   `my_bfm.do_read(a,d);
   $display("Read Address %0h data %0h",a,d);
end
```

Figure 6-7: Simple directed test

Directed tests are created to test some specific function or area of a design. A simple example of a directed test is a DMA test. By reading the specification of a DMA controller, a test writer can easily imagine a test that will put some data into memory, program the DMA controller by writing the starting address of the transfer, the size of the transfer into the control registers, issuing a DMA start command, waiting for the memory transfer to complete, retrieving the contents of the target memory, and comparing the data to make sure it matches the original data. This type of directed test is run carefully the first time to make sure the design is operating as expected and then added to a suite of regression tests just to make sure nothing has broken. Since the test only tests a specific range of addresses and a small set of DMA controller programmability, it is not likely to fail after the first success.

Models and directed testbenches can be created in a variety of languages including Verilog, VHDL, and C/C++. It is also common to mix languages and code the BFM in and HDL such as Verilog and the test sequences in C.

Constrained Random Tests

The main drawback of directed tests is that each test must be designed and manually coded by an engineer. Creating a large number of tests that cover many different situations takes a lot of engineering time. Random tests are another technique engineers use to enhance the directed tests. In design verification, the use of random testing must be constrained such that only legal combinations of sequences of signals are performed. For example, in an ARM SoC not every address can be accessed the same way. Some are memory that can be accessed by read or write, some are read only, and some addresses have no memory at all and accessing them will cause an abort.

Special verification languages have been developed to make constrained random testing easier. The difference between using a verification language versus Verilog or VHDL to create a testbench is the level of information the test writer provides. In a directed testbench we saw how the test writer thinks up a test situation and then writes code to produce the situation. With a verification language the test writer instead codes the specification of the design and the constraints the test must operate within and all the stimulus is automatically generated by a constraint solving algorithm. This means many more tests can be created very quickly and problems that the test writer did not even think of will be encountered. In a directed test the test writer specifies the exact addresses that are used. In a constrained random test the test writer specifies the legal ranges of addresses and the constraint solver takes care of generating the legal stimulus. Figure 6-8 shows an example of specifying the constraints for a test using the *e* verification language. The constraints include the address range used and the percentage of reads and writes on the bus.

```
// Generate mostly READ bursts
extend M1 MAIN master_seq {
   body() @driver.clock is only {
      while TRUE {
         do burst keeping {
            .first_address in [0x0000..0x2400];
            soft .direction == select {
               8: READ;
               1: WRITE
            };
         };
      };
   };
};
```

Figure 6-8: Specifying constraints for constrained random verification

Testbench Architecture

Experience has shown that to take advantage of the higher performance simulation platforms such as simulation acceleration, emulation and prototyping requires good testbench architecture from the start of the project. Three primary elements have been discussed for far: the design being verified, the bus functional models that interface to the design, and the test that provides a sequence of transactions to the BFM. It is important to maintain a clear separation between the test generation and the bus functional models. This separation of test generation from bus functional modes is commonly called *transaction-based verification* since only transactions are sent between the test and the bus functional model. A diagram of the three parts is shown in Figure 6-9.

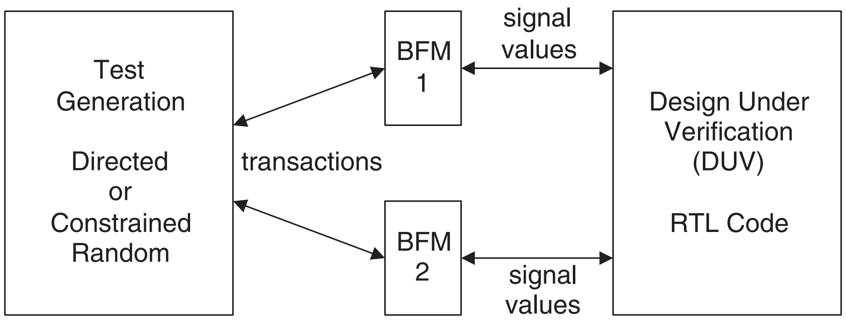


Figure 6-9: Transaction-based verification

A split between the test generation and the bus functional models allows for greater flexibility in the types of models that can be used. Projects do not like to change the languages, methods, and tools used for test generation, but are more willing to change models if there is some benefit. By separating the models from the testbench, different models can be used.

A common application for the use of multiple models is for simulation acceleration. All methods of using a testbench with simulation acceleration or even a prototype run much better if the models are synthesizable. Without synthesizable models, two out of three parts of the simulation cannot run in simulation acceleration, but with synthesizable models, two out of three parts can run in simulation acceleration. Furthermore, the remaining part (test generation) does not naturally communicate with the design every clock cycle since it is sending and receiving only transactions over the BFM interface. Synthesizable models enable high performance acceleration, emulation, and prototyping solutions while maintaining investment in test generation techniques including the use of a verification language.

This testbench architecture also effects co-verification since software engineers are primarily interested in the faster performance. If testbench issues severely limit performance, much less value is achievable for software engineers performing co-verification with a particular simulation platform.

Functional Coverage

Functional coverage is a metric engineers use to decide when verification is done. Functional coverage is useful since it determines if actual design functionality was verified. In contrast, code coverage measures which parts of the code were executed, but does not easily help determine if important functionality was exercised or not. Functional coverage requires the following steps to be performed:

- Decide which functions need to be verified
- Write tests to verify the functions
- Collect test results
- Determine if functions are working correctly

A common application of functional coverage is to verify that all possible bus transactions of a bus interface are working. To achieve high functional coverage, constrained random test generation techniques must be used. Many projects use a full-functional ARM model for the CPU and expect that if the design runs all of the diagnostic software it is working. Even if a design runs all of the diagnostics and the application software, there is no guarantee of high functional coverage on the CPU bus. Certainly, one of the measures of the health of a hardware design is its ability to run all of the software, but keep in mind that software can change after the hardware is committed. These late software changes may result in a different mix of bus transactions that could cause hardware failures.

Some investigation into the need for a combination of co-verification and directed random test generation revealed that most diagnostic software provides poor functional coverage.

A functional coverage model must be defined to include a subset of all the possible combinations of AHB transactions. There are many ways to define this coverage, but let's just include the control signals **HWRITE**, **HBURST**, **HSIZE**, and **HPROT**. This is a total of 10 bits or 1024 combinations. Also added to the coverage model is the bus transaction timing. There can be delays inserted into different parts of the transaction such as delays (idles) before the transaction starts and wait states during a transaction. This adds more combinations to the coverage model. Research in this area has shown that running a diagnostic software program on the ARM that is over

1,000,000 transactions long results in less than 2% functional coverage while using a bus functional model and constrained random testing can achieve over 95% coverage running only 10,000 bus transactions. A combination of software and constrained random testing is the best way to achieve high functional coverage. To build the best stress test with the highest functional coverage, the transactions from a software program can be recorded and merged with constrained random traffic in a single test.

Compliance Suite

Related to functional coverage is compliance testing. Engineers developing design IP that is meant to connect to a standard bus like AHB would like to have some assurance that the IP will work under many different conditions. Users of such IP would like some guarantee the IP will work when placed into their design. One way to achieve these goals is by compliance testing. A compliance test performs three functions:

- Generate stimulus, normally using constrained random techniques
- Monitor the bus for protocol violations
- Measure functional coverage

If satisfactory coverage is achieved without protocol violations then the IP can be certified compliant, and this certification will increase the confidence of both the IP designers and the integrator using the IP in a system.

Software Verification

The SoC development process involves many facets. The traditional notion of the hardware design being the device under test (DUT) and the testbench being used to stimulate the DUT is outdated. The DUT now includes software, and the testbench must verify how hardware and software operate together.

Hardware engineers talk about verification and software engineers talk about testing. Verification involves trying to exercise all possible conditions of the design to make sure nothing breaks. It involves trying as many kinds of stimulus as possible and using constrained random techniques to find problems. Testing involves running a software program and debugging until it appears to run without failures. Not much effort is usually done to stress software.

One of the emerging areas companies are starting to look at is to apply hardware verification principles to software. For example, consider a device driver for a USB device. We know from experience using a Windows PC, that drivers for USB peripherals work most of the time, but every once in a while they have problems recognizing the device or configuring it correctly when it is removed and inserted many times. The software engineer writing the driver probably tests it on real hardware, and likely has no way to verify the driver can recover when unexpected hardware errors occur in the USB protocol. The driver may work well when called under normal conditions with expected parameters, but fails when called with an error or strange parameters. This happens because the software was not verified; only tested. Software engineers don't naturally write software and then spend time trying to break it.

As software becomes a contributing factor to the success or failure of a product or chip, companies are starting to place more emphasis on software verification. I recently heard of a company that has one group that develops software drivers and another that takes the software once it appears to be working and spends time to analyze it for error conditions, perform stress testing, and otherwise search for problems. The concept of a software verification team is not common today, but may be more common in the future, and automated tools for performing both hardware and software verification will be important.

Software Print Statements

Many embedded systems do not have a monitor or screen that can be used by software to write printf( ) statements to trace execution of software. Even those that do may not be able to use it for printf( ) statements since these statements will interfere with the data that is meant to be on the display. To make software debugging easier, it is common for embedded systems to use a UART as a way to output information to a display and to control software via keyboard input. This section covers some of the ways to model a UART when performing co-verification. The best way to model the UART will depend on the execution platform being used and the level of detail required.

In simulation, it's possible to put in an HDL model of a UART and simulate all of the details of accessing the UART. Since the UART speed is very much slower than the other clocks in the design, this method will take a large amount of simulation even for transmitting a few characters. Once the basic functionality of the UART is verified by detailed simulation, it is not necessary to keep simulating at this level.

A shortcut for displaying messages from software in many ARM SoC designs is the called a *Tube*. A Tube in ARM lingo is a mechanism for software running on the ARM core to do printf() statements and have them show up in a simulation window and log file. For many embedded systems and certainly for the world of logic simulation, there is no display for software to write status output. A Tube makes up for this lack of display and allows software to output something that is visible. The origin probably comes from the use of the word tube to describe a subway, a train that tunnels underground to move from point A to point B. A Tube is a way for a software program to "tunnel" a message from software running on the ARM to logic simulation and have it display on the screen and in the simulation log file. A Tube sits on the CPU bus and watches for writes to a specific address, usually one that is not used by the design. When a write occurs the Tube captures the write data and does a print statement to the simulation log file, for example $display() in Verilog. This allows software to write characters to the defined address and have the string messages show up in the simulation window and the log file. The Tube is a shortcut to avoid simulation of the details of a UART and how it transmits messages. For projects that don't need input from a UART, a Tube is the easiest way to do printf() statements from software. An example of a Verilog Tube is shown in Figure 6-10, and an example C program is shown in Figure 6-11 that uses a Tube. The Verilog Tube relies on the design decoder to provide a select line (**HSEL**) for the Tube address of 0xFFF00000.

```
/*
** Tube for software to display strings to simulation window
*/
module tube(HCLK,
            HRESETn,
            HWDATA,
            HWRITE,
            HTRANS,
            HSEL,
            HREADY_IN,
            HREADY);

input HCLK;
input HRESETn;
input [7:0] HWDATA;
input HWRITE;
input HTRANS;
input HSEL;
input HREADY_IN;

output HREADY;

wire tube_sel;
wire WRITE;
wire FINISH;

reg HREADY;
reg [7:0] TUBE_VALUE;
reg TUBE_WRITE;
reg HWRITE_reg;

parameter  TUBE_INT_VALUE =      8'h04;  // Writing Ctrl+D stops the simulation

assign tube_sel = HREADY_IN && HSEL && HTRANS;

always @(posedge HCLK)
begin
  if (!HRESETn)
    HWRITE_reg <= 1'b0;
  else if (tube_sel && HWRITE)
    HWRITE_reg <= 1'b1;
  else
    HWRITE_reg <= 1'b0;
end

always @(posedge HCLK)
begin
  if (!HRESETn)
    HREADY <= 1'b1;
  else if (HWRITE_reg && tube_sel && !HWRITE)
    HREADY <= 1'b0;
    else
      HREADY <= 1'b1;
  end

always @(posedge HCLK)
begin

  if (!HRESETn)
    TUBE_VALUE <= 8'b0;
  else if (WRITE)
    TUBE_VALUE <= HWDATA;
end

always @(posedge HCLK)
begin

  if (!HRESETn)
    TUBE_WRITE <= 1'b0;
  else
    TUBE_WRITE <= WRITE;
end

// Finish Simulation
assign FINISH = (TUBE_VALUE == TUBE_INT_VALUE) && TUBE_WRITE;

assign WRITE = HWRITE_reg & (HWRITE_reg || !HREADY || tube_sel && !HWRITE);

always @(FINISH)
begin
  if (FINISH)
  begin
    $display("%0t  Simulation Stopped by Software Program",$time);
    $finish;
  end
 end

always @(posedge HCLK)
begin
 if (TUBE_WRITE)
   if (TUBE_VALUE != 8'h4) begin
       $write ("%0s", TUBE_VALUE);
   end
end

endmodule
```

Figure 6-10: Verilog Tube

```
#include <stdio.h>


#define TUBE_VALUE       ((volatile unsigned char *)(0xFFF00000))

void finish(void)
{
    *TUBE_VALUE = 0x4; //Ctrl+D
}


void print_string(char *str_ptr)
{
    while (*str_ptr != 0)
    {
        *TUBE_VALUE = *str_ptr;
        str_ptr++;
    }
}


int main(void)
{
    char buffer[100];

    sprintf(buffer,"Hello World!\n");
    print_string(buffer);

    finish();

}
```

Figure 6-11: C program using the Tube

If input from keyboard is required, then a more complete UART model is needed. For logic simulation, an HDL or C model that connects to an xterm that emulates the terminal is usually easy to implement. For simulation acceleration, a synthesizable UART model with a communication channel back to an xterm is available and can offer high performance.

When doing prototyping or in-circuit emulation, a real terminal and keyboard can be used to connect to the UART in the design. If the speed of the prototype or emulator is fast enough, no special consideration is needed. If the speed of the prototype or emulator is not fast enough, a speed bridge may be needed between the design and the terminal.

Summary

This chapter discussed some of the hardware verification issues that are related to co-verification. The hardware verification environment can aid co-verification by providing good information about how the hardware design is behaving when software is executed. The use of assertions can immediately indicate a problem before software runs far into the future with no indication of a problem. The cooperation between testbench and co-verification is crucial to improving the overall SoC verification environment and the productivity of both hardware and software engineers.

This page intentionally left blank

CHAPTER 7

Methodology for an Example ARM SoC

The previous chapters have covered many topics in embedded systems and SoC design. This chapter puts all of the concepts together to discuss a methodology that can be used to verify hardware and software for an example ARM SoC. The methodology is built around the five distinct types of software and the hardware execution engines discussed in Chapter 2, and it attacks these combinations using the Verification Matrix. The process of verifying ARM software with hardware before a design is committed for fabrication is explained as three distinct problems, not one. An example project is used to present different verification scenarios along with strategies of how to solve each problem.

Recall the five distinct types of embedded system software and the fact that software content increases with each type of software. The abstraction of the hardware and the assumptions about the stability and functionality of hardware also increase with each stage of software development.

- System initialization software and hardware abstraction layer (HAL)
- Hardware diagnostic test suite
- Real-time operating system (RTOS)
- RTOS device drivers
- Application software

Also, recall the four distinct methods used for the execution of the hardware design commonly used in SoC verification. Each hardware execution method has specific debugging techniques for both hardware and software associated with it, each with its own set of benefits and limitations. These range from the slowest execution method, with the most flexible debugging, to the fastest, where debugging is more difficult.

- Logic simulation
- Simulation acceleration
- Hardware emulation
- Hardware prototyping

SoC Methodology Difficulty

The main source of difficulty for projects is how to match the type of software being developed with the correct platform or execution engine. Figure 7-1 depicts the problem. Given four or five types of software and three or four verification platforms, a long list of questions is immediately generated:

- What type of software should be run on each type of platform?
- Are all hardware platforms required?
- Is there a single platform that can be used to run all types of software?
- Do all types of software need to be run before the design is fabricated?
- What kind of CPU models and debugging methods are available on each platform?

	Logic Simulation	Simulation Acceleration	Emulation	Hardware Prototype
1. System Initialization and Hardware Abstraction Layer (HAL)	?	?	?	?
2. Hardware Diagnostics	?	?	?	?
3. Real-time Operating System (RTOS)	?	?	?	?
4. Device Drivers	?	?	?	?
5. Application Software	?	?	?	?

Figure 7-1: Methodology difficulty

Verification Efficiency

Verification efficiency is a popular topic being discussed among engineers and engineering management today. Of course, every tool that is available promises improved efficiency. Engineers are wondering how to leverage all of the point tools that have been developed to solve specific issues to create a single, cohesive methodology for hardware and software verification.

In order to work smarter, engineers can make improvements to the overall verification process, by automating best practices rather than focusing on incremental speed improvements in individual point tools. In addition, engineers can improve in one of three areas that take up their time during the verification process:

Verification environment creation is the time spent to construct the environment, including testbenches, testcases, models, and so forth. This process is mostly manual with some automation in the area of testbench generation.

Execution is the time spent to run the test scenarios. Increasing raw performance is the primary way to run the test scenarios in a shorter period of time.

Interpreting results and debugging is the time spent to decide if test scenarios are working and how to find and fix problems for scenarios that are not working. This is also a mostly manual process with some automation in the area of functional coverage.

Since debugging is the most manual, it is important to do it efficiently. Optimizing the debugging process saves valuable time.

The Debugging Loop

Beyond just tools and debugging methods there is one more crucial area to cover, the human interaction between hardware and software engineers. The optimization of this interaction is as important as the optimization of the hardware and software itself. This is best illustrated by describing a common interaction model used in the early stages of many projects today.

A typical project will first partition the system into hardware and software and follow with a register map specification, usually written with a word processor, describing the software interface to the custom hardware. As time passes, these registers are changed, but the specs are not updated. The written descriptions are either incorrect or not clearly understood by the software engineers. Hardware and software engineers have different concepts and methods for describing the system functionality. For example, software engineers don't usually have a concept of logic signals being "active low."

To debug these systems, the process starts when software engineers compile code and prepare memory image files for execution. Because they are unfamiliar with logic simulation and emulation tools, they pass the image file over to the hardware or verification engineers who run the test.

When the test fails, the verification engineer tries to isolate the problems, and presents the result to a design engineer who is familiar with the specific part of the design that may have a problem. The design engineer analyzes and fixes the problem and runs the test again.

If the problem is in the software, then the software engineer must be called in to inspect the problem. For the software engineer to debug the problem, he needs the help of the verification engineer who understands waveforms from the simulation and can correlate it to the software.

Finally, the software engineer generates a fix, and the process starts all over. This iterative cycle of throwing results over the wall among three groups struggling to work together slows down the progress of the project. The debugging loop is shown in Figure 7-2.

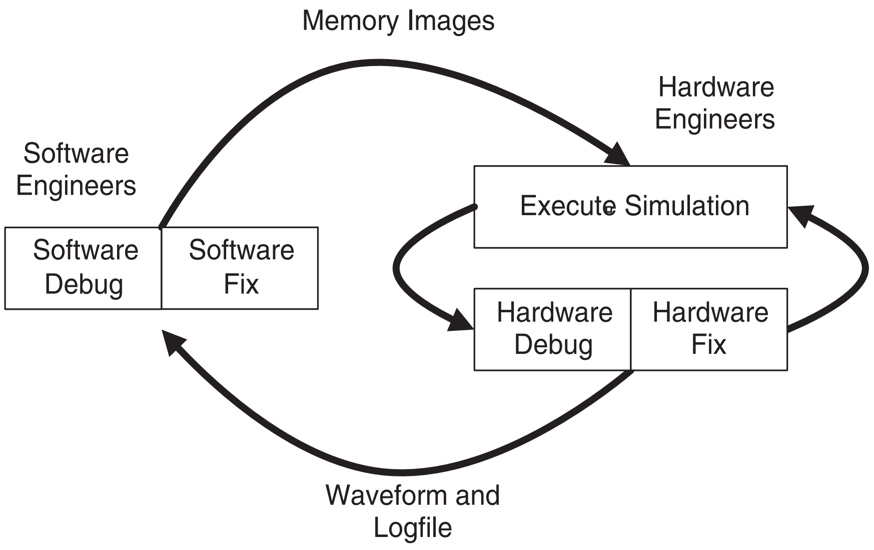


Figure 7-2: The debugging loop

Co-verification tightens the debugging loop by allowing all groups to work independently and concurrently as much as possible. Any tools or methods that empower software engineers to find and fix problems with less intervention by verification and hardware design engineers are of great benefit to the entire project. Better tools alone will not solve the differences in skills required for software engineers to work in a hardware-centric environment.

Co-Verification Methodology

The solution to implement a co-verification methodology for SoC verification and to reconcile the different views of hardware and software engineers is to combine a single platform that provides logic simulation, simulation acceleration, and in-circuit emulation with application-specific solutions for co-verification and transaction-based verification. Consider as an example an SoC that includes an ARM microprocessor. As described previously, hardware engineers are interested in bus transactions of the CPU. This requires a transaction-based interface that works well with the verification platform for use during logic simulation, acceleration, and emulation modes. Since it needs to be used with logic simulation and later with acceleration and emulation, it cannot be constructed such that it will be a bottleneck to overall acceleration and emulation performance.

Software engineers require good CPU models and debugging tools. For each of the five different types of software, they prefer either a software model of the ARM CPU or a hardware model of the ARM CPU. The three primary verification platform execution methods combined with the three representations of the ARM microprocessor form the verification matrix with nine modes of operation shown in Figure 7-3.

		ARM Models		
		CPU Bus Model (AHB)	Software Model of ARM CPU	Hardware Model of ARM CPU
Execution Methods	Software Logic Simulation	Block level Tests 1	Initialization Software 2	In-Circuit Interface Testing 3
	Simulation Acceleration	Directed Tests 4	Diagnostic Software 5	RTOS Porting 6
	Emulation	Constrained Random Tests 7	Device Drivers 8	Application Software 9

Figure 7-3: The verification matrix

The next sections describe how each type of software can be executed on either a software or hardware model of the ARM CPU using one or more of the platform's execution modes.

System Initialization and HAL Development

Many complex SoC projects use nothing more than a full-functional model of the microprocessor core in a logic simulator to write and debug this code. Software debugging with waveforms requires a true guru who understands hardware and software and can disassemble instructions in his head using instruction fetches on the data bus. For the ARM SoC example, the ideal debugging solution for early development of system initialization and HAL code is one based on a cycle-accurate instruction set simulation model tightly coupled to a logic simulator containing the SoC hardware design. This provides interactive, graphical software debugging for the software engineer to single step through the code and verify register and memory contents with excellent flexibility and control. Simulation performance is less important because the code must be verified line-by-line, and the number of lines of code is relatively small. This situation is labeled as box 2 in the matrix in Figure 7-3.

Diagnostics

During the development of diagnostic tests, the logic simulator becomes the bottleneck of the verification environment. As tests run longer and the number of tests increases, it becomes more difficult both to verify the entire hardware design and to continue to run old tests as hardware and software errors are fixed. This phase is also the most crucial since it is where most hardware bugs are found. Debugging tools for both software and hardware at this stage are very important.

The best solution uses simulation acceleration to increase the simulation performance over what is a possible using an ordinary software simulator. A simulation environment running at 10 to 100 Hz is not fast enough for engineers to run and test. Moreover, the memory optimization techniques commonly used by co-verification tools are not as useful because the main purpose of the diagnostics is hardware verification. A simulation acceleration system that runs at speeds of 10 to 50 kHz is the ideal platform for simulation performance and debugging. The use of simulation acceleration with the software model of the ARM is labeled as box 5 in the matrix in Figure 7-3.

RTOS and Device Drivers

The initial RTOS port is a good place to take advantage of memory optimizations commonly used in co-verification. These memory optimizations retrieve instructions at a much faster rate than using logic simulation. The result is less simulation detail on how the ARM SoC would work, but increased performance. Since the instruction fetch path is well verified, using the memory optimizations makes sense, rather than going back in the diagnostic test suite as a workaround for a low logic simulator. The initial RTOS boot requires box 5 on the matrix in Figure 7-3.

Once the RTOS is booted and stable with the selected device drivers, as shown in box 8, future work can be done using a faster execution method, such as in-circuit emulation. The number of hardware bugs is very small, so the increased performance is well worth any tradeoff in hardware debugging. This shifts the focus of the software engineers from box 5 to boxes 6 and 9.

Application Software

Application software requires the highest performance and possible stimulus from other sources, such as graphics, I/O interfaces like USB, or networking. This is an ideal fit for in-circuit emulation (ICE). Initial bring up for ICE is done using in-circuit simulation (ICS). ICS connects the software simulator with the target board by using the emulator as a pass-through connection to the target system. The necessary target boards, interfaces and test equipment are assembled in the lab for ICE. This represents a shift from box 3 to box 9 on the matrix in Figure 7-3.

Testbench Development

Hardware engineers are focused on making sure the bus interface logic connected to the microprocessor works correctly. The bus functional model (BFM) allows this to be done efficiently without requiring the overhead of a full functional model (FFM) and software to run on the CPU. There are many different kinds of BFMs available from IP companies, EDA vendors, and microprocessor suppliers. Unfortunately, all of them have been created using C/C++, verification languages, or behavioral Verilog or VHDL. These languages are suitable for logic simulation, but are not efficient for simulation acceleration and emulation. The SoC co-verification methodology requires a BFM that runs well for all phases of verification, from the start of a project, as shown in box 1 in Figure 7-3, moving to acceleration and emulation for directed and random testing, as shown in boxes 4 and 7. To achieve this, a transaction-based interface to synthesizable BFM for the CPU bus is ideal. By operating at the transaction level, the communication is minimized between the testbench and the verification platform. Using a synthesizable BFM and a transaction-based interface to the verification platform optimizes performance, while simultaneously allowing for the use of C/C++ or other verification languages to create testbenches. A BFM that works the same way from simulation to emulation and provides the required performance, while simultaneously following the industry trend toward verification automation, is an important part of a unified verification methodology.

Three Verification Phases

In the area of co-verification, there are really three phases of the project and three problems to solve. These phases are shown in Figure 7-4. Each phase involves both hardware and software, but the requirements to complete each phase are different.

- Hardware verification
- Hardware/software co-verification
- Software development

Hardware verification aims to make sure the hardware has as few bugs as possible. It includes the initial system integration and testing of the initialization software to set up a working simulation of both hardware and software. The software to be debugged is low-level initialization code and diagnostic software that attempts to verify the hardware and create stress tests to find problems. During this phase, the goal is to find hardware bugs. Automation of test scenarios and testcases is important. It requires the best hardware debugging tools, and performance is important but secondary to automation and debugging.

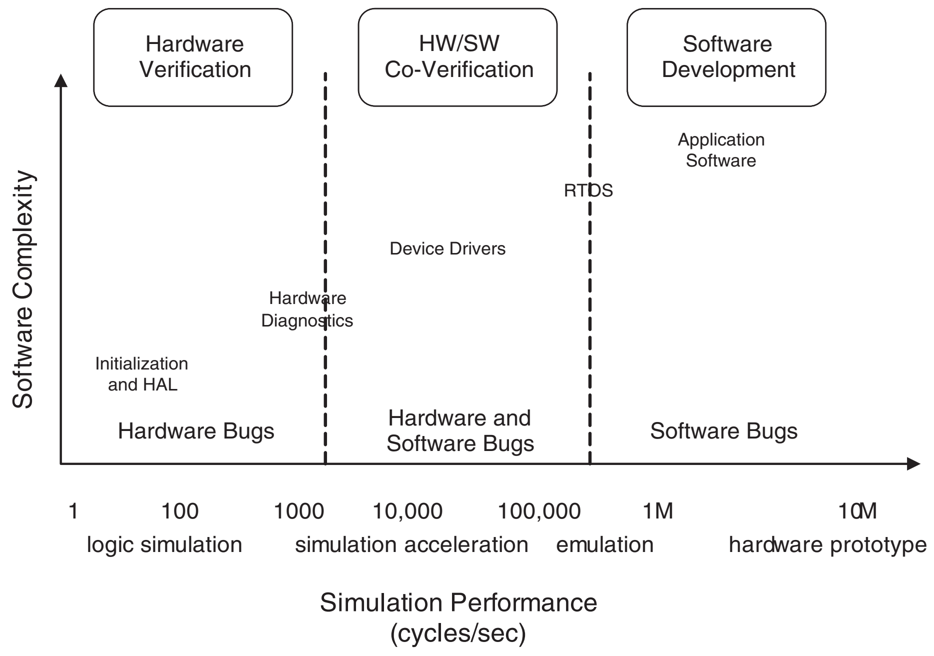


Figure 7-4: Three phases of verification

True hardware/software co-verification is the time in the project when the hardware dependent software must be verified on the hardware. This is the most challenging phase since both hardware and software are likely to have bugs so good tools for both hardware and software debugging are needed. Performance must also be increased due to the number of cycles required to verify software completely along with hardware.

Software development is also called *application software*. This is the pure software that does not interface directly to the hardware. For this phase, performance is king. Software engineers are not as patient since they are accustomed to high performance. These are also the type of software engineers that will leave if the hardware has bugs and just come back later. They are not motivated to find hardware bugs.

The methodology trend is to search for a common platform that will serve all three phases. A flow of simulation, acceleration, and emulation using the verification matrix in Figure 7-3 has shown successful. The one area where it does not always provide a suitable solution is in application software development. For this task, often hardware prototypes are used where possible.

The next section outlines in more detail how to apply the proposed verification flow.

Example of ARM Verification Flow

An example design containing an ARM CPU core, DSP, and image processor was described back in Chapter 3, Figure 3-14. This section looks at the design and describes how to verify hardware and software in a logical way.

Block and Subsystem Verification

Before any integration occurs, each individual block and subsystem should be verified using logic simulation and an appropriate verification environment. Finding easy bugs is much better to do in a small simulation environment where it is easy to compile quickly, waveform dumping is manageable, and runtimes are short.

This is also the time when assertions should be inserted into the design to document the design intent as the module or subsystem gets integrated into larger chip and system simulations. As discussed, the assertion methodology should be suitable for the hardware execution engines that will used later such as simulation acceleration and emulation. This is also the best time to measure code coverage and functional coverage to the extent possible. Constrained random verification techniques offer a good way to achieve high coverage with manageable run times. If there are areas of the design that cannot be exercised until more subsystems are integrated, these coverage points must be addressed later. It is best to achieve as much protocol coverage as possible and do any kind of compliance testing during this stage of verification to gain as much confidence as possible in the block or subsystem before its integration into larger verification environments. This gives the highest chance of success when the blocks are integrated.

Consider the verification of the image processor block of the example design. To verify this block, it's possible to construct a simulation of the image processor on the AHB with another AHB master, arbiter, and decoder. There is no need for any CPU software. A testbench can be used to generate stimulus for AHB and for the image processor. Performance is acceptable with a logic simulator since only a portion of the design is used. A possible verification scenario is shown in Figure 7-5.

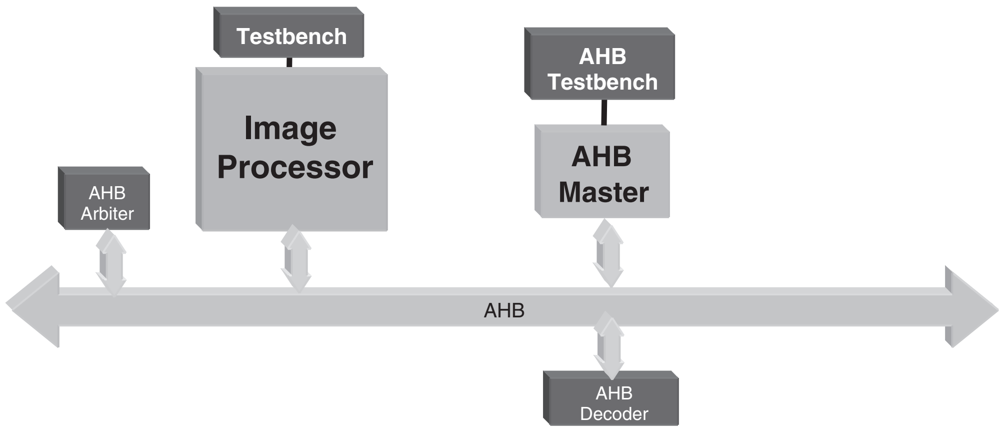
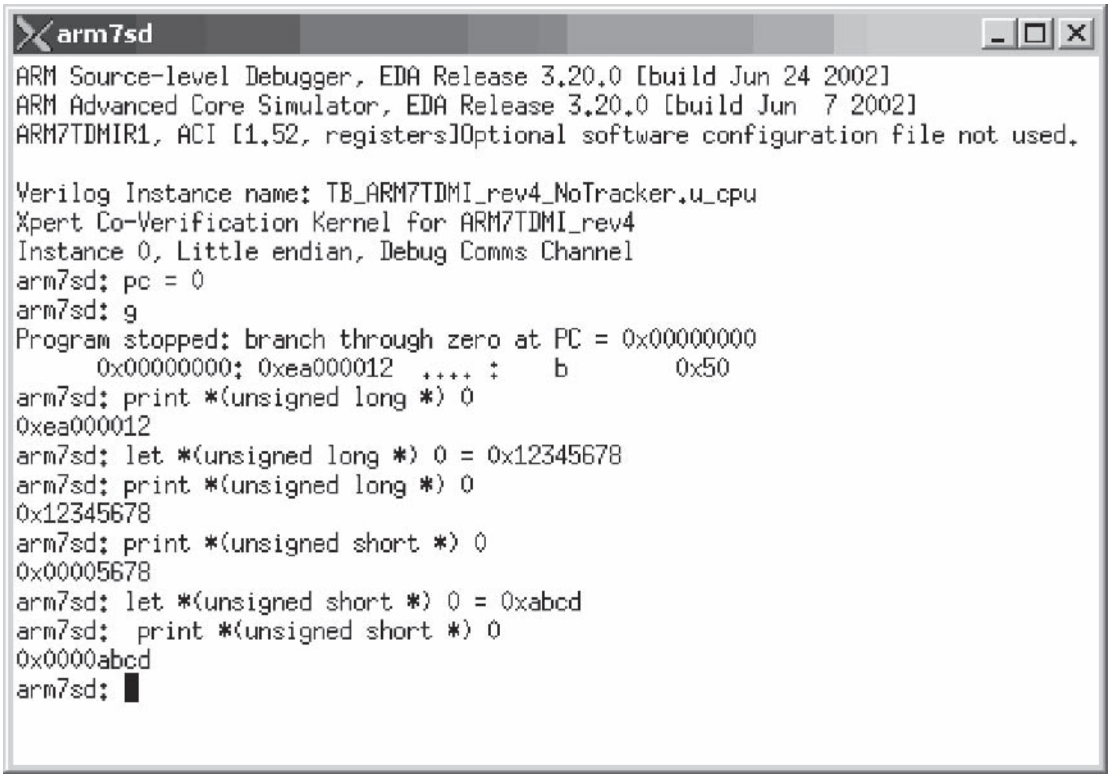


Figure 7-5: Hardware verification of image processor

Initial System Integration

When block and subsystem verification is complete the next step is to assemble a full chip RTL netlist and verification environment. The first phase is to make sure all blocks coexist and the various interfaces are working. The environment should be constructed to allow for easy substitution of the ARM design sign-off model (DSM), RTL CPU model, co-verification model, or emulation model depending on which will be used.

The best way to verify the blocks are working together is to verify the ability to access all memories and programmable registers. The easiest way to verify the initial values of registers and memory is using a software debugger. There is no requirement to write any assembly language code to start the CPU. Simply run simulation until the end of reset and use the debugger to read various addresses in the CPU memory map to check the design decoder, arbiter, bus controller, and other logic are working. The ARM debugger will automatically stop on exceptions unless the $vector_catch variable is changed to 0. To enable the debugger test to be reused it is best to either write a script that the debugger executes or use the log feature of the debugger to capture manually entered debugger commands. The result is a quick test that can be reused later if there is any suspicion of logic problems or bus conflicts as the design changes. An example of using the software debugger to check various memory locations is shown in Figure 7-6.



```
ARM Source-level Debugger, EDA Release 3.20.0 [build Jun 24 2002]
ARM Advanced Core Simulator, EDA Release 3.20.0 [build Jun  7 2002]
ARM7TDMIR1, ACI [1.52, registers]Optional software configuration file not used.

Verilog Instance name: TB_ARM7TDMI_rev4_NoTracker.u_cpu
Xpert Co-Verification Kernel for ARM7TDMI_rev4
Instance 0, Little endian, Debug Comms Channel
arm7sd: pc = 0
arm7sd: g
Program stopped: branch through zero at PC = 0x00000000
      0x00000000: 0xea000012  .... :    b       0x50
arm7sd: print *(unsigned long *) 0
0xea000012
arm7sd: let *(unsigned long *) 0 = 0x12345678
arm7sd: print *(unsigned long *) 0
0x12345678
arm7sd: print *(unsigned short *) 0
0x00005678
arm7sd: let *(unsigned short *) 0 = 0xabcd
arm7sd:  print *(unsigned short *) 0
0x0000abcd
arm7sd: 
```

Figure 7-6: Using the software debugger to check integration

The next step is to test and debug ARM initialization software and HAL using a co-verification model and logic simulator. Performance with logic simulation is typically in the range of 10 to 1000 cycles/sec, but code is small and graphical debugging provides control and visibility of the design.

Focused Hardware Verification

Now that the design is operational and the memory map is working, the focus is on comprehensive hardware verification. A complete suite of diagnostic tests is developed to make sure each function of the design is working. These diagnostics are not only useful for pre-silicon verification, but can be reused during the chip or system bring-up in the lab. The diagnostic suite should be developed to obtain maximum hardware design coverage. This is where most of the hardware design is verified and debugged. As the test suite grows, logic simulation becomes too slow to run long tests. This is the time to transition to simulation acceleration for higher performance. Speeds of 10,000 to 100,000 cyles/sec with full cycle accuracy provide the necessary performance to run nightly regression tests. During this phase, constrained random testing using high-performance synthesizable models is used to augment the functional coverage provided by software diagnostics. The benefits of a high-level language for verification combined with acceleration/emulation performance are ideal for long random tests. The architecture for such a high-performance environment is shown in Figure 7-7.

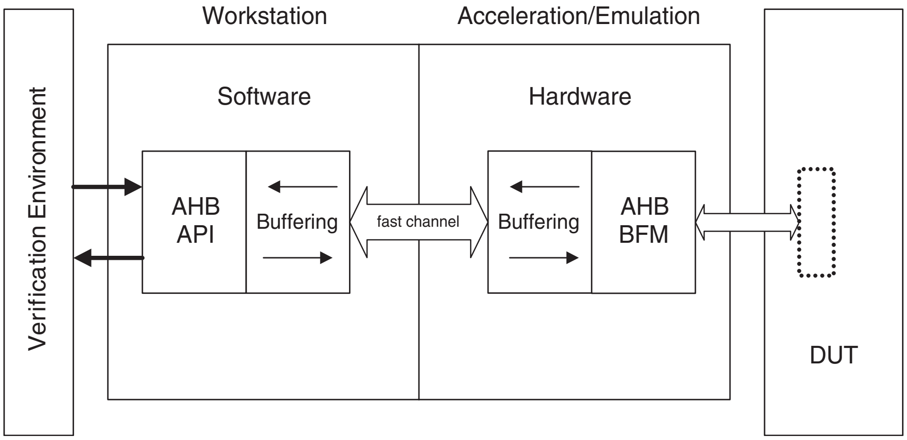


Figure 7-7: High-performance AHB verification environment

Hardware/Software Co-Verification

True co-verification is done during those times when there are still bugs in hardware as well as software. This is common during diagnostic development and initial testing of device drivers. This is the most important time for hardware and software engineers to work together to resolve problems. It requires tools with good control and visibility for software as well as high visibility for hardware debugging. Starting with a software model of the ARM core for co-verification provides the best control and ease-of-use. Depending on the level of performance required logic simulation can be used for early work and later simulation acceleration for longer tests. This is also a good place to do initial RTOS porting.

Figure 7-8 shows a verification scenario where the designer of the image processor block works with a software engineer writing directed diagnostics to verify hardware operation. Co-verification is used to debug the software test. The memory controller is now added to include memory for the microprocessor instructions and the interrupt controller on the APB is now added to allow the diagnostics to verify interrupts.

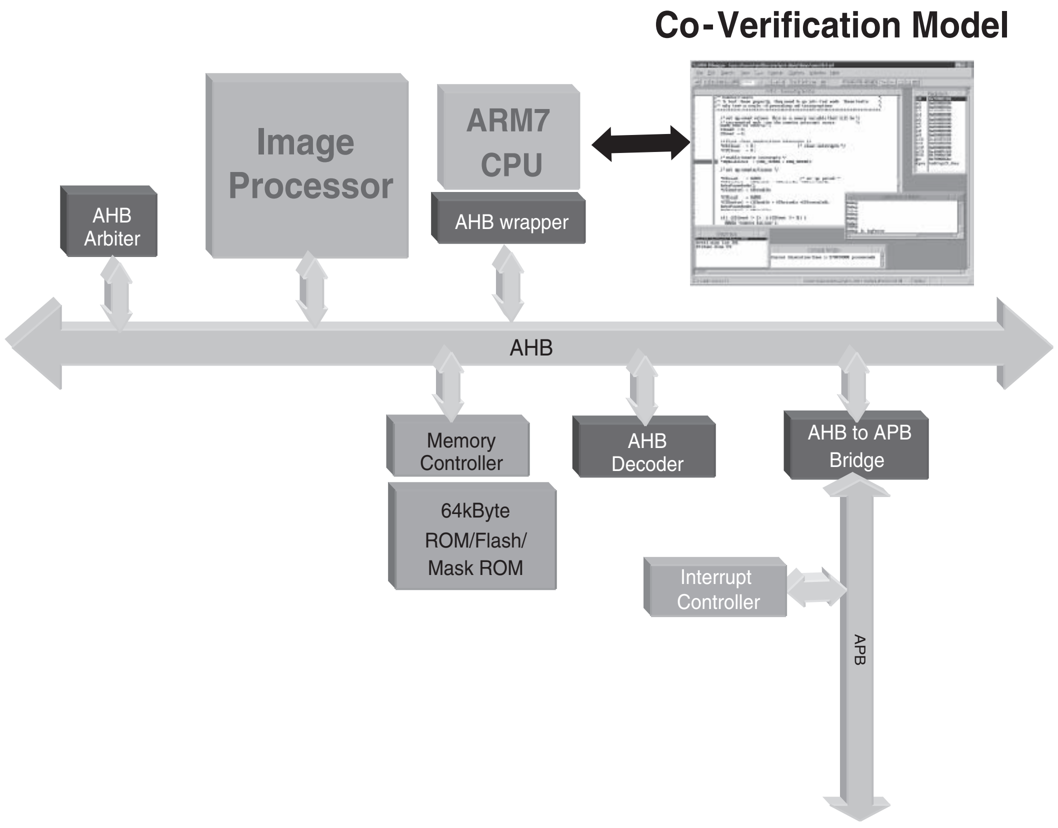


Figure 7-8: Co-verification for device drivers

System Software Testing

Application software requires the highest level of performance. This means emulation or prototyping, depending on the specifics of the project situation. Application software assumes there are no bugs in the hardware. Engineers writing this type of software like a development environment that is as close to the real system as possible. This means JTAG debugging for the ARM software. It also means connection to real hardware and test equipment for design stimulus. This type of software is one of the most popular reasons why projects look at in-circuit emulation. A diagram of JTAG debugging with ICE is shown in Figure 7-9.

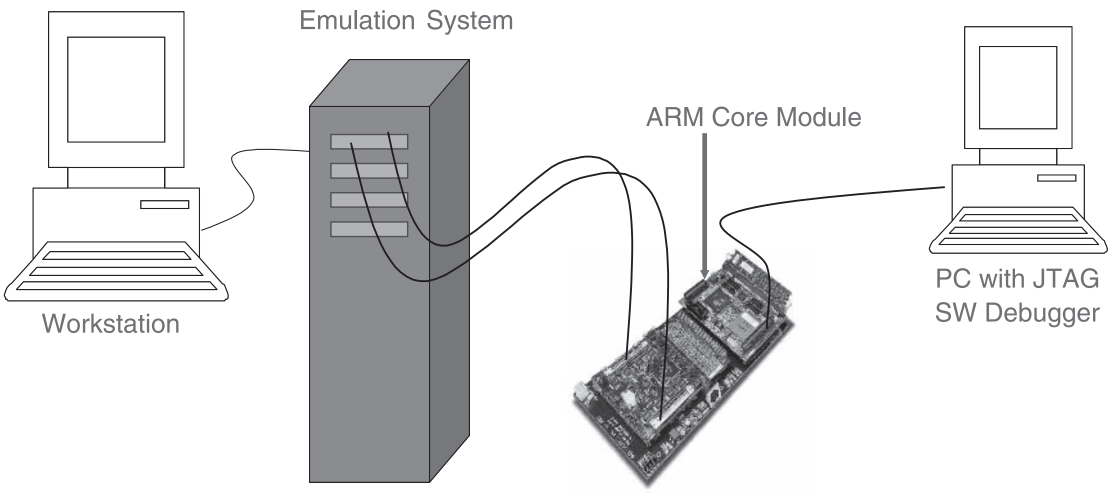


Figure 7-9: JTAG debugging with in-circuit emulation

The key to success with application software and ICE hinges on all of the previous steps that were done to get to this point. For a team of software engineers to come in with an RTOS and applications and expect to just plug in a JTAG probe to an emulator is not realistic. A disconnect can occur when the needs of software engineers for emulation are great, but the hardware team decides they can get by with logic simulation and a farm of Linux machines to accomplish verification. Even if the software team can recruit one or two engineers to help with emulation bring-up and configuration, it can be difficult to realize of the full benefits compared to a complete methodology built around the smooth transition from logic simulation, to simulation acceleration, to emulation. I'm always worried when a team that is primarily driven by software engineers starts deciding on what kind of emulator to use based on PowerPoint slides and the idea that they all pretty much work and have a JTAG connector.

Emulation can use either an RTL model of the CPU or a board with a testchip. Both models can interface to JTAG hardware. A verification scenario is shown in Figure 7-10 that uses an emulation model for the ARM CPU and other in-circuit emulation interfaces such as Ethernet to generate real network traffic.

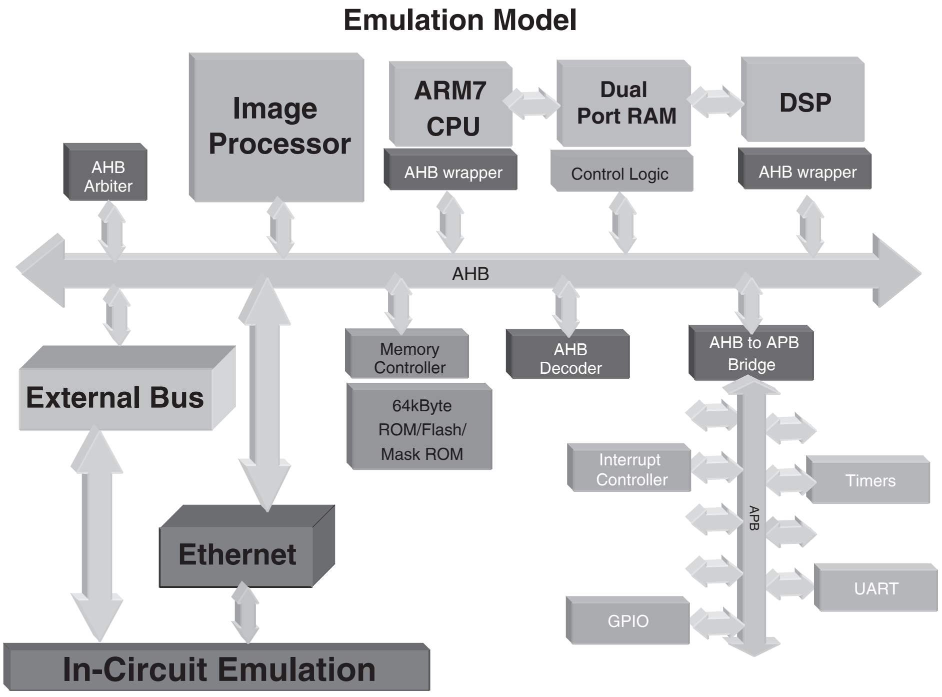


Figure 7-10: Testing application software with in-circuit emulation

The Co-Verification Engineer

The concept of the hardware verification engineer is common in ASIC and SoC projects. Project teams have design engineers with the responsibility to design and code hardware functions in Verilog and VHDL. Most project teams also have a set of engineers called *verification engineers* with the responsibility to make sure the design works correctly. Many tools and techniques used by verification engineers have been described so far. Verification engineers succeed by finding as many bugs as possible in the hardware design. Even if there are no hardware bugs, they succeed by proving to the extent possible that there are no bugs that will show up after the chip or system is constructed. An entire industry has been built around the concept of verification with between 50 and 75 companies providing tools and services related to verification. Verification providers are certain to see continued demand for new products since projects continue to get more complex and more difficult to verify.

For many years the companies providing verification products have been trying to expand the market by selling products that target the embedded system software that accompanies the verification projects they already serve. It sounds like a logical opportunity for market expansion. Unfortunately, efforts in this area have been difficult for various reasons:

- Software teams have much smaller budgets to spend on tools; they often use free or open source software.
- Software is viewed as less critical since it can always be changed at any time with none of the costs associated with changing hardware.

While these obstacles are certainly true, the main reason companies have trouble creating products targeting embedded software is that there is no concept of a *software verification engineer*. Software engineers certainly try to produce high quality software using techniques such as code reviews and code inspections by project peers. We know that software products like Microsoft Word or your favorite logic simulator have a QA team that tracks bugs and works to automate testing. There are even tools for applications with a GUI to record mouse sequences to automate testing of commands. I always imagine that when a new version of Windows is being prepared, there are about 1000 test engineers somewhere that try to stress it and make it crash.

Given all this, there is still no concept of a software verification engineer for the embedded software associated with the SoC projects we have been discussing. The role of the software verification engineer is much more than testing, it involves the same concepts that are applied to hardware verification. These are constrained random generation to create multiple test scenarios and collecting and measuring functional coverage. The job of the software verification engineer is to find bugs in the software. Very few such engineers exist in the world of SoC design, and hence the reason why verification companies have such trouble selling verification tools to software engineers.

Even if the software verification engineers did exist, they would have a tough time doing software verification in SoC projects before tapeout because the verification environment is hardware centric. The solution is a new kind of engineer called a *co-verification engineer*. The co-verification engineer is skilled in the hardware verification tools and environment and has the ability to define methodology to best run different types of embedded software and actually try to find bugs and verify it. The co-verification engineer does not design hardware or write embedded software, but has the skills to figure out when one or the other is not working and is focused not only on finding bugs, but in developing metrics to prove the bugs in both hardware and software are eliminated. To date I have never seen any advertisement for a software verification engineer, much less a co-verification engineer, but creating such roles on SoC project teams is the best way to improve SoC verification. Who knows, some day there could even be a market to sell products to co-verification engineers to help verify embedded system software.

Conclusion

Congratulations! You have persisted this far and are still interested in co-verification. One of the conclusions that should be clear by now is there are no easy answers to the integration problem of hardware and software. There are many ways to do it, each with a different set of pros and cons to be considered. There are many tradeoffs related to performance, cost, ease of use, flexibility, and debugging. This is the reason why there is no single commercial product that has emerged as a clear winner. In fact, the fragmentation of different approaches and products is such that there is not even a clean market segment that defines the market and the market share of each product. A defined market exists for a subset of co-verification products such as those based on logic simulation and ISS models, but while this technique is useful it solves only a part of the co-verification problem. I often wonder if co-verification will ever have a standard set of tools and flow that all, or most, or even many projects use. Will there ever be a day where it becomes like the logic simulator or the software debugger? The answer is probably not until the concepts of software verification and the co-verification engineer become reality. For now, the best strategy is to learn as much as possible about the options and apply the ones that provide the best value to minimize project risk. There is a definite trend for project teams to find a common platform and methodology that is shared by hardware and software teams. The approach of hardware engineers focusing on logic simulation and software engineers focusing on separate efforts to create high-level C models or prototypes is becoming more and more difficult. At the same time, there is still a struggle to meet the needs of hardware verification, true co-verification, and application software. The following story is a good summary of the state of co-verification.

Methodology Gridlock

I want to share a story as evidence of the confusion that can plague project teams. I once visited a company designing a chip with an ARM926EJ-S core and a DSP for mobile phone applications. I spent the afternoon talking about tools and techniques to do simulation acceleration, in-circuit emulation, co-verification, and system integration. It seemed that no matter what types of tools and methodology I proposed they had a reason why it did not meet their requirements. It seemed we just went around and around in circles. As I reflected on the meeting and traced back through the discussion I realized it was more like a Three Stooges "who's on first?" routine. I relate the story only to illustrate that is not easy to meet the requirements of all of the different types of engineers involved in the project. In fact, the various requirements can conflict with each other. The story went something like this.

User: We started by verifying our chip using a logic simulator and the RTL code of the ARM926. As our tests get long and our regression suite gets bigger we found our simulation environment runs too slow and we have no good way to debug software, only waveforms and log files.

Me: We have benchmarked your design and found that by using simulation acceleration on our emulation system you can achieve a 40X speedup in your simulation environment with little or no change.

User: Sounds good, but to address performance and software debugging, we recently started to use co-verification with time and memory optimization. For tests that require little or no interaction with the hardware design we can finish them 100X faster, but for tests where the DSP is active and more hardware interaction is needed the benefit is more like 5X. Of course, memory optimization achieves speedup by skipping simulation so we debug the tests using co-verification and then re-run them on the RTL model of the ARM to simulate the entire test and make sure everything works.

Me: We have integrated an ARM926 co-verification model with the emulation system to provide software debugging and it runs 20X faster than your original simulation environment and simulates all of the bus activity.

User: That sounds good, but co-verification will be more than 20X faster on some tests with memory optimization on.

Me: But if you have to re-run the tests on the RTL model don't the runtimes become too long.

User: We have a farm of workstations so we don't worry about it.

Me: What if a test that takes 10 hours fails? How will you debug software? How will you get waveforms? You will probably have to rerun the tests with waveform dumping on.

User: Let's not think about that. What we really need is something that runs 500,000 instructions/sec so we can run the software stack for making phone calls. We find co-verification with logic simulation really only addresses the low-level software, drivers and below.

Me: We have an in-circuit emulation solution that's not quite that fast, but close. It also offers JTAG software debugging, just like you would debug software when you get your chip back and put it on a board.

User: The performance sounds great, but it looks like only one software engineer at a time can use it and this is an expensive emulator.

Me: We have developed a way to use the emulator in a batch environment that allows software engineers to use post-processing debugging methods to trace software execution. This way software engineers can debug without tying up the emulator.

User: That sounds useful, but then we can't change the program, memory, or registers in the middle of the test and see the results. Our idea to replicate something for more than one software engineer was to develop a SystemC model of the chip and pass out copies to software engineers. We think it should run about 200k instruction/sec.

Me: How will you model all of the custom hardware in SystemC and ensure it is modeled correctly? It looks like you already have RTL code for most of it and the design is pretty complex.

User: Good point, that's why we started a project to build a custom prototype of our chip that runs at 20 MHz so we can make real phone calls, but now we are not sure if we can get it run successfully due to FPGA synthesis and timing issues.

Me: Wow, 20 MHz sounds good, but what if there is a hardware bug? How can you get any visibility into what the hardware is doing since you can't debug what is happening inside the prototype FPGAs.

User: Not sure, I guess we will go back to the simulation environment and try to reproduce it.

Me: I thought your simulation environment was pretty slow and does not have any good way to debug software?

User: That's right, we can try to reproduce the problem using our co-verification with memory and time optimization.

Me: It may work, but the timing will be much different, you may not get the same problem.

User: Probably true since the software stack has a lot of timing dependencies, in that case we will run the test on the emulation system and JTAG debugger to reproduce it.

Me: But I thought you didn't want to use emulation because only one software engineer can use it.

User: Well, at this point we only have one problem so emulation seems to be a good fit. Let's call it quits for today.

Me: Good idea.

User: I need to go start some long simulation jobs before I go home for the night; I'm hoping to have some results when I come in on Monday morning.

Me: Good luck with your simulations.

This page intentionally left blank

Afterward

This book has provided the necessary background and detailed understanding of the issues facing projects dealing with the integration of hardware and software. This understanding enables you to apply these techniques to your own projects. Putting theory into practice requires learning even more about specific tools and technologies and making a plan based on a methodology best suited to your specific project. A good place to start is with the products and methods developed by Verisity. As Yoav Hollander stated in the foreword, there are many companies trying to sell verification tools and I encourage you to learn about all of them, but I believe Verisity is as good a place to start as any to get both a broad and detailed understanding of verification. Here's a summary of the related products you can use as a starting point on your journey.

- **Specman Elite** testbench automation provides constraint-driven random test generation, data and temporal checking, and functional coverage analysis. Specman Elite uses the ***e*** functional verification language.
- **SpeXsim** combines the testbench automation capabilities and proven processes of Specman Elite with mixed-language simulation technology to provide an integrated verification system for block and chip-level verification.
- **SureCov** Automatic FSM, expression, and code coverage.
- ***e* Verification Components (*e*VCs)** provide a complete verification environment including stimulus generation, protocol checking, and functional coverage for common protocols such as AHB.
- **Xtreme** provides simulation acceleration and in-circuit emulation for improved verification performance and serves as a platform for co-verification. **SpeXtreme** high-performance chip and system-level verification system combines Specman Elite, Xtreme and ***e*** synthesis technology.

- **Xpert** provides co-verification using software models (ISS) and works with logic simulation and simulation acceleration.
- **XoC** provides co-verification using RTL models and in-circuit emulation models of microprocessor cores. XoC also provides transaction-level models suitable for emulation.
- ***e*RM** (***e*** Reuse Methodology) provides functional verification productivity gains for advanced ASICs and SoCs through it's comprehensive set of best-known methods for ***e*** environment and ***e***VC development practices.
- ***s*VM** (System Verification Methodology) is a prepackaged verification knowledge-transfer system that provides automation for verifying full systems, including hardware-software systems for SoC and system-level designs. ***s*VM** includes multi-channel generation based on Verisity's Specman Elite test-bench automation.
- ***v*Manager** automates the management of verification projects and drives the overall process from executable plan to achieve total coverage and verification closure

If you would like to discuss the details of your own project, feel free to contact me. I always enjoy learning more about what engineers are using and how things are working. Maybe someday one of you will call me or meet me and identify yourself as a "co-verification engineer."

Jason Andrews
jason@verisity.com

Index