EDK Concepts, Tools, and Techniques

A Hands-On Guide to Effective Embedded System Design

XTP013 EDK 10.1

Xilinx is disclosing this user guide, manual, release note, and/or specification (the "Documentation") to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU "AS-IS" WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/01/07	9.1i	Book release for EDK 9.1i.
09/05/07	9.2i	Book release for EDK 9.2i.
11/05/07	10.1	Book release for ISE Unified 10.1 release.
9/18/08	10.1	Book release for ISE v10.1 SP3 release.

Table of Contents

Chapter 5: Creating Your Own Intellectual Property (IP)

Chapter 6: The Software Platform and SDK

Chapter 7: Introduction to Simulation in XPS

Chapter 8: Implementing and Downloading Your Design

Chapter 9: Debugging the Design

Appendix A: More About BFM Simulation

Appendix B: Glossary

Schedule of Figures

Chapter 1: Introduction

Chapter 2: Creating a New Project

Chapter 3: Xilinx Platform Studio

Chapter 4: The Embedded Hardware Platform

Chapter 5: Creating Your Own Intellectual Property (IP)

Chapter 6: The Software Platform and SDK

Chapter 7: Introduction to Simulation in XPS

Chapter 8: Implementing and Downloading Your Design

Chapter 9: Debugging the Design

Appendix A: More About BFM Simulation

Appendix B: Glossary

About This Guide

This guide explains the basics of the EDK embedded design flow, tools architecture, and concepts behind the EDK design process. It also provides an opportunity for you try out the EDK tools through a series of *Test Drives*, during which you build a sample project.

Guide contents include:

- Chapter 1, Introduction
- Chapter 2, Creating a New Project
- Chapter 3, Xilinx Platform Studio
- Chapter 4, The Embedded Hardware Platform
- Chapter 5, Creating Your Own Intellectual Property (IP)
- Chapter 6, The Software Platform and SDK
- Chapter 7, Introduction to Simulation in XPS
- Chapter 8, Implementing and Downloading Your Design
- Chapter 9, Debugging the Design
- Appendix A, More About BFM Simulation
- Appendix B, Glossary

Additional Resources

To find additional EDK documentation, see:

www.xilinx.com/ise/embedded/edk_docs.htm

To search the Answers Database for silicon, software, and IP questions and answers, or to create a technical support WebCase, see:

www.xilinx.com/support/mysupport.htm

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

This document uses the following typographical conventions:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File > Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets **[ ]**	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*option_name*] *design_name*
Braces **{ }**	A list of items from which you must choose one or more	**`lowpwr ={on|off}`**
Vertical bar **|**	Separates items in a list of choices	**`lowpwr ={on|off}`**

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		`.`
		`.`
		`.`

Horizontal ellipsis **. . .**	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
<u>Blue, underlined text</u>	Hyperlink to a Website (URL)	Go to http://www.xilinx.com for the latest speed files.

Introduction

Welcome

The Xilinx Embedded Development Kit (EDK) is a suite of tools and Intellectual Property (IP) that enables you to design a complete embedded processor system for implementation in a Xilinx Field Programmable Gate Array (FPGA) device.

This guide describes the design flow for developing a custom embedded processing system using EDK. Some background information is provided, but the main focus is on the features of EDK and their use.

Read this document if:

- you need an introduction to EDK and its utilities.
- it has been a while since you've designed an embedded processor system.
- you are installing the Xilinx EDK tools.
- you need a quick reference while designing a processor system.

Note: This guide is written based on Windows operating system. The behavior or the graphical user interface (GUI) on a Linux system may vary slightly.

Take a Test Drive!

Because the best way to learn a software tool is to use it, this document provides opportunities for you to work with the tools described in this guide. Specifications and instructions for creating a sample project are provided in various *Take a Test Drive!* sections as you go along. These test drives also include information about what happens when you run automated functions. Test Drives are indicated by the car icon, shown at start of this paragraph.

Additional Documentation

Additional documentation about the Xilinx EDK is available at: www.xilinx.com/ise/embedded/edk_docs.html

Documentation about the Xilinx Integrated Software Environment® (ISE) is available at: www.xilinx.com/support/software_manuals.htm.

How EDK Simplifies Embedded Processor Design

Embedded systems are somewhat complex. Getting the hardware and software portions of an embedded design to work are projects in themselves. Merging the two design components so they function as one system brings additional challenges. Add an FPGA

design project to the mix, and the situation has the potential to become very confusing indeed.

Components of EDK To simplify the design process, Xilinx offers several sets of tools. It is a good idea to get to know these basic tool names, project file names, acronyms, and abbreviations. To make this easier for you, we have included a Glossary of EDK-specific terms at the end of this guide.

Integrated Software Environment

The Integrated Software Environment (ISE) is the foundation for Xilinx FPGA logic design. Because FPGA design can also be an involved process, Xilinx has provided software development tools that allow you to simplify some of this complexity. Various utilities, such as constraints entry, timing analysis, logic placement and routing, and device programming have all been integrated into ISE. For more helpful information about using the Xilinx ISE® tools for FPGA design see:
www.xilinx.com/support/software_manuals.htm.

Embedded Development Kit

The Embedded Development Kit (EDK) is a suite of tools *and* Intellectual Property (IP) that enables you to design a complete embedded processor system for implementation in a Xilinx FPGA device. Think of it as an umbrella covering all things related to embedded processor systems and their design. The Xilinx ISE software must also be installed to run EDK.

Xilinx Platform Studio

The Xilinx Platform Studio (XPS) is the development environment or GUI used for designing the *hardware* portion of your embedded processor system.

Software Development Kit

Platform Studio Software Development Kit (SDK) is an integrated development environment, complimentary to XPS, that is used for C/C++ embedded software application creation and verification. SDK is built on the Eclipse™ open-source framework, so this software development tool might appear familiar to you or members of your design team.

Other EDK Components

Following is a list of some of the other EDK elements.

- Hardware IP for the Xilinx embedded processors
- Drivers and libraries for embedded software development
- GNU Compiler and debugger for C/C++ software development targeting the MicroBlaze™ and PowerPC™ processors
- Documentation
- Sample projects

The utilities provided with EDK are designed to assist in all phases of the embedded design process.

How Do the Tools Expedite the Design Process?

Figure 1-1 shows the simplified flow for an embedded design.
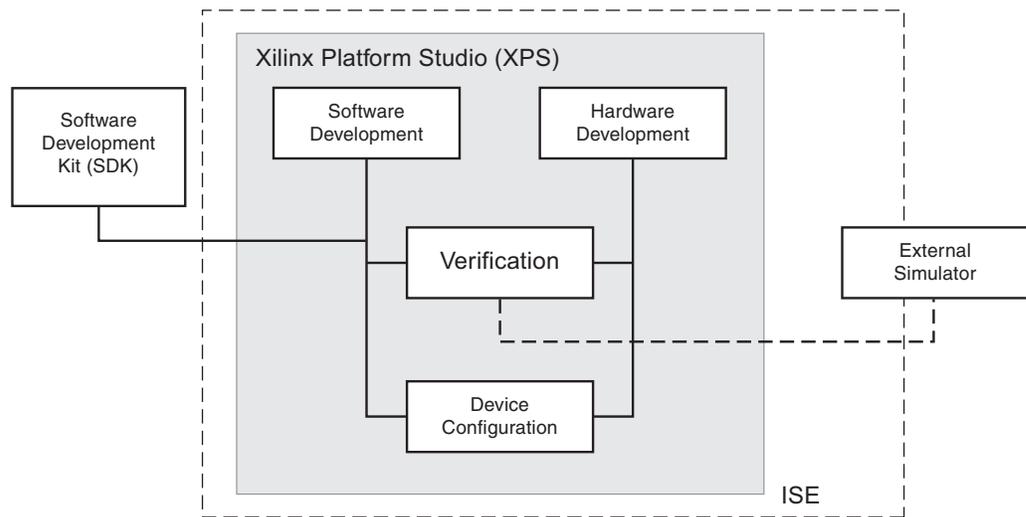
Figure 1-1: **Basic Embedded Design Process Flow**

Embedded System Overview

The following is an overview of how these tools work together to simplify the design process.

- The recommend design flow is to begin with an ISE project, and then to add an embedded processor source to the ISE project.

- XPS is used primarily for embedded processor hardware system development. Configuration of the microprocessor, peripherals, and the interconnection of these components, along with their respective property assignments, takes place in XPS.

- SDK is the recommended software development environment for simple and complex software applications. While basic software development can be accomplished within XPS, this capability will be removed in a future release.

- Verifying the correct functionality of your hardware platform can be accomplished by running the design through a Hardware Description Language (HDL) simulator. XPS facilitates three types of simulation:

  ♦ Behavioral

  ♦ Structural

  ♦ Timing-accurate

  XPS automatically sets up the verification process structure, including HDL files for simulation. You will only have to enter clock timing and reset stimulus information, along with any application code.

For more information about the embedded design process as it relates to XPS, see "Design Process Overview" in the *Embedded Systems Tools Reference Manual,* available at: www.xilinx.com/ise/embedded/edk_docs.htm.

Before Starting

Before we start discussing the tools in depth, it would be a good idea to make sure they are installed properly and that the environments you have set up match those you need to follow the instructions in the *Take a Test Drive* sections in this guide.

Installation Requirements: What You Need to Run EDK Tools

Xilinx ISE

Several utilities in EDK use functionality that is delivered with tools contained in ISE. So, to use the EDK tools, you first need to have ISE installed. Be sure you have also installed the latest ISE service pack as well. To obtain information and product downloads for the ISE software, go to www.xilinx.com/support/download/index.htm.

EDK Installation Requirements

Bash Shell for Linux

If you are running EDK on a Linux platform, you need a bash shell. Also, be sure to check out the supported platforms covered in the Xilinx document *Getting Started with the EDK*, available at: www.xilinx.com/ise/embedded/edk_docs.htm.

Software Registration ID

You'll need a software registration ID to install EDK. You can get one online at: www.xilinx.com/products/design_resources/design_tool/index.htm.

EDK Installation

Exact installation instructions vary, depending upon whether the software was obtained from an electronic download or on a DVD. For detailed installation instructions, please refer to the *ISE 10.1 Release Notes and Installation Guide*.

You can fine more information about EDK installation online at: www.xilinx.com/ise/embedded/edk_docs.htm.

Note: ISE and EDK major versions *must* be the same. For example, if you are installing EDK version 10.1, you must also install ISE version 10.1.

Installation Requirements for Simulation

To perform simulation using the EDK tools, you must have the following steps completed:

1. An IP-Protect capable simulator (ModelSim® PE/SE v6.3c or Mentor Graphics® IUS v6.1) is required for the simulation steps.

2. Install the CoreConnect™ Toolkit. CoreConnect is a free utility provided by IBM®. You can download CoreConnect from the Xilinx website at: www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=dr_pcentral_coreconnect.

   After you make the appropriate selections on the web page to order and register, you will have access to the download.

   Note: The CoreConnect toolkit is only required if you are going to perform Bus Functional Model (BFM) Simulations. If you do not intend to run BFM simulations, you do not need to install the CoreConnect toolkit.

3. Compile the simulation libraries (if you haven't already done so), following the procedure outlined in the EDK help system available in XPS or on the Xilinx web page under "Xilinx Platform Studio Help Topics" at www.xilinx.com/ise/embedded/edk_docs.htm.

   a. If you are opening the help from XPS, select Help > Help Topics.

   b. Navigate to Procedures for Embedded Processor Design > Simulation > Compiling Simulation Libraries in XPS > Compiling Simulation Libraries in XPS.

For additional information about the installation process, see *Getting Started with EDK* at: www.xilinx.com/ise/embedded/edk_docs.htm.

Creating a New Project

Now that you've been introduced to EDK, let's begin looking at how to use these tools to develop an embedded system.

The Base System Builder (BSB)

About BSB

BSB is a wizard that quickly and efficiently establishes a working design, which you can then customize.

At the end of this section, you will have the opportunity to begin your first Test Drive, using BSB to create a project.

Why Should I Use BSB?

Xilinx® recommends using the BSB Wizard to create the foundation for any new embedded design project. BSB may be all you need to create your design, but if more customization is required, BSB saves you a lot of time because it automates basic hardware and software platform configuration tasks common to most processor designs. After running the wizard, you have a working project that contains all the basic elements needed to build a more customized or complex system, should that be necessary.

What You Can Do in the BSB Wizard

Using the BSB Wizard, you can create your project file, choose a board, select and configure a processor and I/O interfaces, add internal peripherals, set up software, and generate a system summary report.

BSB recognizes the system components and configurations on the selected board and provides the options appropriate to your selections.

Creating Your Top-level Project File (*.`xmp`)

The Xilinx Microprocessor Project (.xmp) file*

A Xilinx Microprocessor Project (XMP) file is the top-level file description of the embedded system under development. All XPS project information is saved in the XMP file, including the location of the Microprocessor Hardware Specification (MHS) and Microprocessor Software Specification (MSS) files. The MHS and MSS files are described in detail later in this tutorial.

The XMP file also contains information about C source and header files that XPS is to compile, as well as any executable files that the Software Development Kit (SDK) compiles. The project also includes the FPGA architecture family and the device type for which the hardware tool flow must be run.

File creation includes the option to apply settings from another project you have created with the BSB.

Selecting a Board Type

BSB allows you to select a board type from a list or to create a custom board.

Supported Boards

If you are targeting one of the supported embedded processor development boards available from Xilinx or from one of our partners, BSB lets you choose from the peripherals available on that board, automatically match the FPGA pinout to the board, and create a completed platform and test application that is ready to download and run on the board. Each option has functional default values that are pre-selected in XPS. This base-level project can be further enhanced in XPS, or can be implemented using the Xilinx implementation utilities provided by ISE.

Upon initial installation of EDK, only Xilinx board files are installed. If you want to target a third party board, you must add the necessary board support files. The BSB Select Board screen contains a link that assists you in finding third party board support files. After the files are installed, the BSB drop-down menus display those boards as well.

Custom Boards

If you are developing a design for a custom board, BSB lets you select and interconnect one of the available processor cores (MicroBlaze or PowerPC, depending on your selected target FPGA device) with a variety of compatible and commonly used peripheral cores from the IP library. This gives you a hardware system to use as a starting point. You can add more processors and peripherals if needed. The utilities provided in XPS assist with this, including the creation of custom peripherals.

Selecting and Configuring a Processor

Processors

You can choose a MicroBlaze or PowerPC processor and select:

- Architecture type
- Device type
- Package
- Speed grade
- Reference clock frequency
- Processor-bus clock frequency
- Reset polarity
- Processor configuration for debug
- Cache setup
- Floating Point Unit (FPU) setting

Selecting and Configuring Multiple I/O Interfaces

Multiple I/O Interfaces

BSB understands the external memory and I/O devices available on your predefined board and allows you to select the following, as appropriate to a given device:

- Which devices to use
- Baud rate
- Peripheral type
- Number of data bits
- Parity
- Whether or not to use interrupts

For your convenience, data sheets for external memory and I/O devices can be opened from within the wizard.

Adding Internal Peripherals

BSB allows you to add additional peripherals. There is a caveat, however: the peripherals are supported by the selected board and FPGA device architecture. For a custom board, only certain peripherals are available for general selection and automatic system connection.

Setting Up Software

Standard input and output devices can be specified in BSB, and you can select sample C applications that you would like XPS to generate. Each application includes a linker script. The sample applications from which you can select include a memory test, peripheral test, or both.

Viewing a System Summary Page

After you have made your selections in the wizard, BSB displays a system summary page. At this point, you can choose to generate the project, or you can go back to any previous wizard screen and revise the settings.

It should be noted that while this guide is using the Xilinx ML507 Development Board and targeting the PowerPC 440 processor, other recent boards could just as easily be used. If the board does not have an FPGA with a PowerPC-class processor (PowerPC 440 or PowerPC 405), MicroBlaze can be used for most of the Test Drives in this guide. Some behavior may vary slightly, but the discussions and exercises will still be of value.

Take a Test Drive!

To run BSB Wizard, you need to first start the ISE Project Navigator, and create a project with an embedded processor system as the top level.

1. Start ISE Project Navigator.
2. Select File > New Project. This launches the New Project Wizard.

3. Use the information listed in the table below to make your selections in the Wizard screens.

Wizard Screen	System Property	Setting or Command to Use
Create New Project	• Project Name • Project Location • Top-level source type	• Choose a name for your project (do not use spaces). • Choose a location for your project. • Select **HDL** (default). • Click **Next**.
Device Properties	• Product Category • Family • Device • Package • Speed • Synthesis Tool • Simulator • Preferred Language	• All • Virtex®-5 • XC5VFX70T • FF1136 • -1 • XST (VHDL/VERILOG) • User-specific • VHDL • Accept other defaults and click **Next**.
Create New Source		• Click **New Source**.
Select Source Type	In the menu tree in the left pane, click **Embedded Processor.**	• Enter **system** in File name field • Click **Next** • Click **Finish** • Click **Yes** • Click **Next**
Add existing sources	Do not add anything.	• Click **Next**.
Project Summary		• **Click Finish.**

Note: After running through the ISE Project Navigator new project wizard, it will recognize that you have an embedded processor system, and will start Platform Studio with the message *This project appears to be a blank project. Do you want to create a Base System using the BSB Wizard*? (This can take a few moments.) Click Yes.

Now that the BSB wizard has started, you can create a project using the settings described in the table below.

Note: If no setting or command is indicated in the table below, **accept the default values**.

Wizard Screens	System Property	Setting or Command to Use
Welcome to the Base System Builder	Project type options	• Select the option to create a new design.
Select Board	Board vendor and name	• Choose **Xilinx** as your board vendor. • Select the **Virtex-5 ML507 Evaluation Platform.** The ML507 board contains a Virtex®-5 FXT device, which means BSB allows you to select either a MicroBlaze or PowerPC soft processor core. • Select Board Revision A (default)
Select Processor	Processor type	• Select **PowerPC.**
Configure PowerPC Processor	• Clock frequencies • Processor configuration, Debug I/F	• Use defaults. • Choose **FPGA JTAG**. (default) This means that the JTAG pins will also be used for processor debug. *NOTE: Cache and FPU settings should be disabled (left blank).*
Configure IO Interfaces (four screens)	Xilinx-provided IP selections	• Select **RS232_UART_1** • **RS232_Uart_1, select XPS_UART_16550**. Accept the new defaults that appear. • Select **Push_Buttons_Position_5Bit.** For this project, deselect all other options. *NOTE: IP that must be purchased is displayed with an accompanying lock symbol. You can evaluate the IP for a period of time, but it must be purchased to continue working in your design.*
Add Internal Peripherals	Default is XPS_BRAM_IF_CNTLR with an 8 KB memory size.	• Select 16 KB memory size.

Wizard Screens	System Property	Setting or Command to Use
Software Setup	• Software setup In the BSB software setup screen you specify how you would like to use your system. BSB can also set up any software tests you would like to create. • Boot memory • Memory and peripheral tests The software tests send or receive information to selected peripherals. The microprocessor interprets the status of the peripherals and reports it via the STDIN/STDOUT peripheral	• For the STDIN and STDOUT devices, select **RS232_Uart**. • `Select` **`xps_bram_if_cntlr_1`** • Use default application tests.
Configure Memory Test Application	Instruction, Data, and Stack/Heap memory locations	• For this project, place all of these in **`xps_bram_if_cntlr_1.`** This specifies that the program code operates out of the block RAM contained in the FPGA (`xps_bram`) using the BRAM controller (`_if_cntlr_1`).
Configure Peripheral Test Application	Same as above	Same as above.

Wizard Screens	System Property	Setting or Command to Use
System Created	System summary page	After you have selected and configured all your system components, BSB displays an overview of the system, allowing you to verify your selections. • You should have PowerPC440 running at 125 MHz, buses running at 125 MHz with 16KB of on chip memory. • You should have the following attached components: an xps_bram_if_cntlr, an xps_uart16550, and an xps_gpio. You can go back to any previous wizard dialog and make revisions. BSB creates a default memory map. The memory map cannot be modified inside BSB, but it can be changed after BSB is closed. • After reviewing the system summary (and making any changes needed), click Generate.
Finish	During design generation, the directory structure of your system is created. The HDL and other files are populated with the choices you made in BSB, and connections between the processor, busses, and peripherals are handled, and any additional logic is instantiated.	• Click Finish. A message dialog will appear with information about how to manage the constraints generated for this system. After reading this message, click OK to close the dialog.

Note on BSB and Custom Boards

If you plan to create a project that includes a custom board, you must create a Xilinx Board Description file (`*.xbd`) for your custom board library and place it in the `$XILINX_EDK/board` location. For more information, see "Xilinx Board Description (XBD)" in the *Platform Specification Format Reference Manual*, available at www.xilinx.com/support/documentation/dt_edk_edk10-1.htm.

What's Next?

In the next chapter, you will learn how you can view and modify your new project in XPS.

Xilinx Platform Studio

Now that you have created a baseline project with BSB, it's time to take a look at the options available in Xilinx Platform Studio (XPS). Using XPS, you will be able to build on the project you created with BSB. This chapter takes you on a tour of XPS. Subsequent chapters in the document discuss how to use XPS to modify your design.

Note: Taking the tour of XPS provided in this chapter is recommended. It will enable you to more easily follow the rest of this book and other documentation on XPS.

What is XPS?

XPS includes a graphical user interface (GUI), along with a set of tools that aid in project design. This chapter describes the XPS GUI and some of the most commonly used tools.

The XPS GUI

From the XPS GUI, you can design a complete embedded processor system for implementation within a Xilinx FPGA device. The XPS main window is shown in Figure 3-1.

Using the XPS User Interface

Note that the XPS main window is divided into these three areas.

- Project Information Area
- System Assembly View
- Console Window

Optional Test Drives are provided in this chapter so you can explore the information and tools available in each of the XPS main window areas.
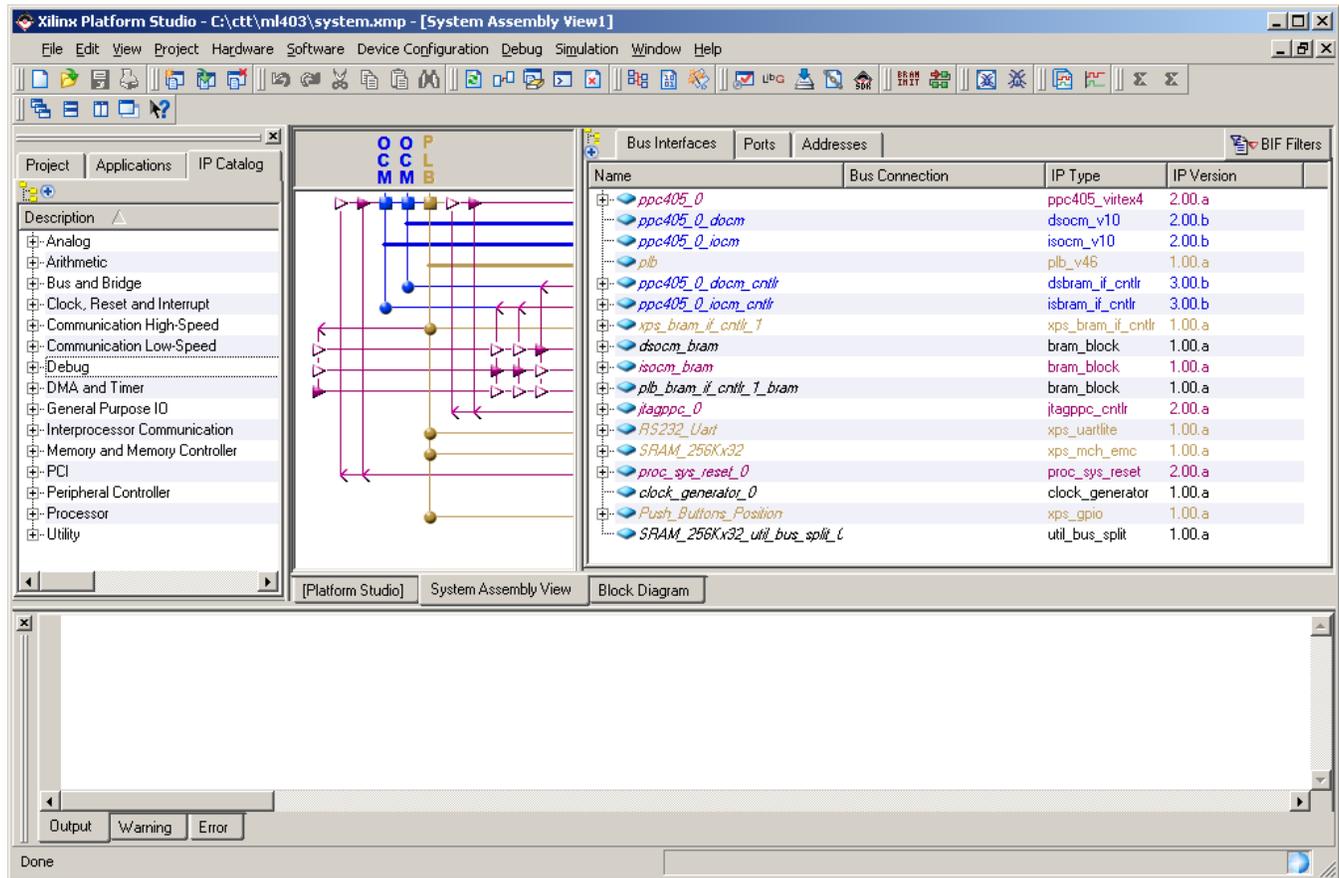
Figure 3-1: **Xilinx Platform Studio User Interface**

Project Information Area

The Project Information Area offers control over and information about your project. The Project Information Area includes Project, Applications, and IP Catalog tabs.

Project Tab

The Project Tab, shown in Figure 3-2, lists references to project-related files. Information is grouped in the following general categories:

- Project Files

  All project-specific files such as the Microprocessor Hardware Specification (MHS) files, Microprocessor Software Specification (MSS) files, User Constraints File (UCF) files, iMPACT Command files, Implementation Option files, and Bitgen Option files.

- Project Options

  All project-specific options, such as Device, Netlist, Implementation, Hardware Description Language (HDL), and Sim Model options.

- Reference Files

  All log and output files produced by the XPS implementation processes.
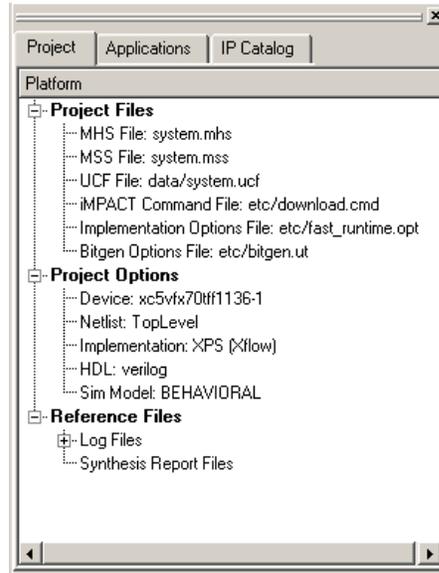
Figure 3-2: **Project Information Area, Project Tab**

Applications Tab

The Applications tab, shown in Figure 3-3, lists all software application option settings, header files, and source files that are associated with each application project. With this tab selected, you can:

- create and add a software application project, build the project, and load it to the block RAM.

- set compiler options.

- add source and header files to the project.

Note: While XPS allow you to create and manage software projects, SDK is the recommended tool for all software development. The software development capabilities in XPS will be removed in a future release.
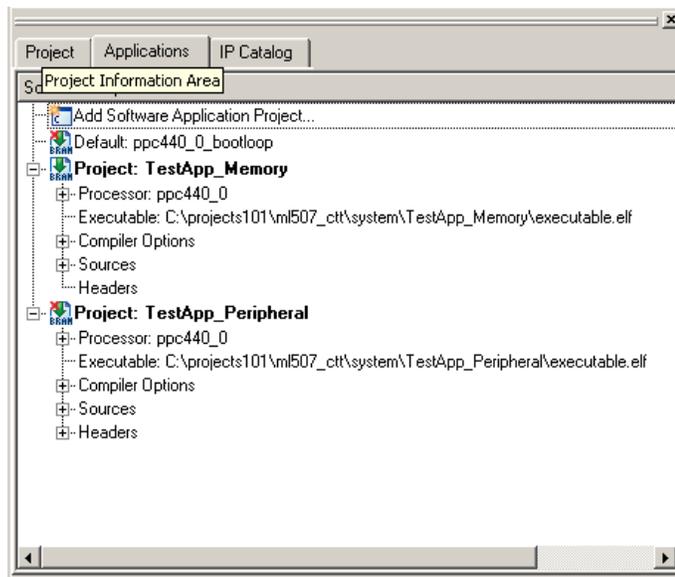
Figure 3-3: **Project Information Area, Applications Tab**

IP Catalog Tab

The IP Catalog tab (Figure 3-4), lists all the EDK IP cores and any custom IP cores you created.

If a project is open, only those IP cores that are compatible with the target Xilinx device architecture are displayed. The catalog lists information about the IP cores, including release version, status (active, early access or deprecated), lock (not licensed, locked, or unlocked), processor support, and a short description.

Additional details about the IP core, including the version change history, data sheet, and the Microprocessor Peripheral Description (MPD) file, are available from the right-click menu. By default, the IP cores are grouped hierarchically by function.

XILINX®

Figure 3-4: **Project Information Area, IP Catalog Tab**

Note: You may have to click and drag to expand the pane to view all IP details.

Take a Test Drive!

1. In the XPS GUI, Click the Project tab.

   Notice that right-clicking an item under Project Files lets you open it in XPS and that right-clicking an item under Project Options allows you to open the Project Options dialog.

   You can close the open project file by right-clicking the tab for that file, and selecting **Close**.

2. Click the Applications tab.

   a. Collapse the Project: TestApp_Memory (using the +/- box) entry.

   b. Expand the four sub-headers below Project: TestApp_Peripheral.

      - Under Processor: ppc440_0, note the `xparameters.h` file.

        The `xparameters.h` file contains the system address map and is an integral part of the Board Support Package (BSP). If you have been following the previous Test Drive steps, the BSP has not been generated yet, so this file is unavailable.

      - Under Compiler Options and Sources, note that both a linker script and test application executables were automatically generated by the BSB Wizard when the selected test applications were created.

3. Click the IP Catalog tab.

   a. Find the Communication Low-Speed IP category and expand it.

b. Locate the XPS_UART (16550-Style) peripheral and right-click to view the PDF data sheet for the XPS_UART (16550-Style).

Note the option to select a flat or hierarchical view.

c. Click the directories icon circled in Figure 3-5 to switch between the two views.

System Assembly View

The System Assembly View allows you to view and configure system block elements. If the System Assembly View is not already maximized in the main window, click the System Assembly View tab at the bottom of the pane to open it.

Bus Interface, Ports, and Address Filters

XPS provides Bus Interface, Ports, and Addresses tabs in the System Assembly View (Figure 3-5), to organize information about your design and allow you to more easily edit your hardware platform.
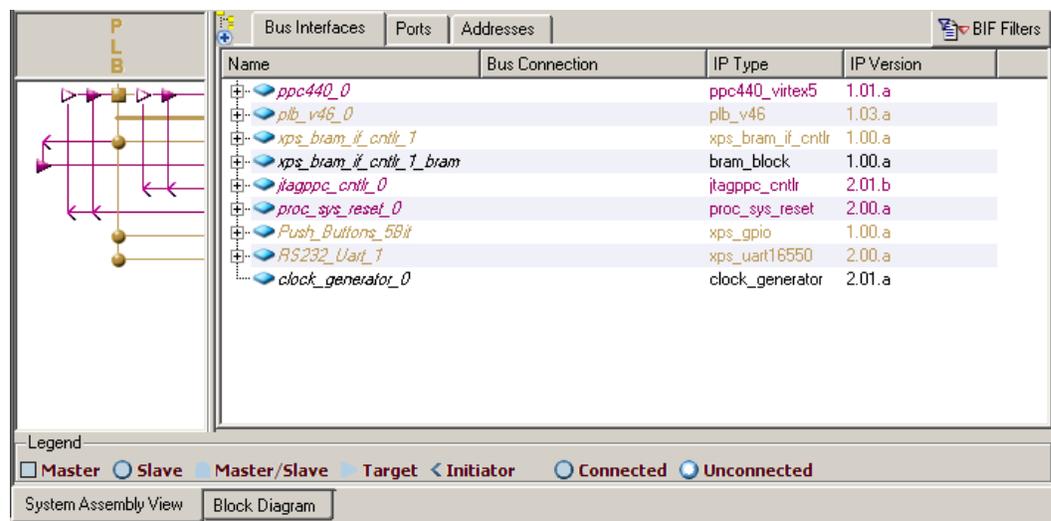
Figure 3-5: **System Assembly View Contents**

Connectivity Panel

With the Bus Interfaces tab selected, you'll see the Connectivity Panel, (labeled in the figure above). The Connectivity Panel is a graphical representation of the hardware platform interconnects.

*   A vertical line represents a bus, and a horizontal line represents a bus interface to an IP core.

*   If a compatible connection can be made, a connector is displayed at the intersection between the bus and IP core bus interface.

*   The lines and connectors are color-coded to show bus compatibility.

*   Differently shaped connection symbols indicate whether IP blocks are bus masters or bus slaves.

*   A hollow connector represents a connection that you can make, and a filled connector represents a connection made. To create or disable a connection, click the connector symbol.

Information Viewing and Sorting

To allow you to sort information and revise your design more easily, the System Assembly View provides two view options: hierarchical view and flat view.

Hierarchical and Flat Views

Hierarchical view is the default in the System Assembly View. In the hierarchical view, the information about your design is based on the IP core instances in your hardware platform and organized in an expandable or collapsible tree structure.

When you click the directory structure icon (circled in Figure 3-5), the ports are displayed either hierarchically or in a flattened, or flat, view. The flat view allows you to sort information in the System Assembly View alphanumerically by any column.

Expanded or Collapsed Nodes

The +/- icon expands or collapses all nets or buses associated with an IP to allow quick association of a net with the IP elements.

Take a Test Drive!

In System Assembly View, click the Ports tab (located at the top of the screen).

1. Expand the External Ports category to view the signals that are present outside the FPGA device.

2. Note the signal names in the Net column and find the signals related to the RS232_Uart_1. (You may need to drag the right side of the Net column header to see its entire contents.) These are referenced in the next step. Collapse this category when finished.

3. Scroll down to locate the RS232_Uart peripheral and expand it.

   Note the Net names and how they correspond to the names that were present as external signals. The RX and TX net from the UART are name-associated with the external ports.

4. Right-click the RS232_Uart_1 peripheral icon and select Configure IP to launch the RS232_Uart_1:xps_uart16550_v2_00_a parameters dialog. You can use the parameters dialog for any peripheral to adjust various settings available for the IP. Take a moment and observe what happens when you hover the cursor over a parameter name. Note the three top buttons and the tabs available for this core. Close this dialog when finished.

5. Click the directories icon (circled in Figure 3-5), and switch between the hierarchical and flat views.

Platform Studio Tab

In the same space as the System Assembly View, there is a **Platform Studio** tab. The Platform Studio tab display (Figure 3-6) provides an embedded design flow diagram, with links to related help topics.

If at any point you are not sure what to do next, or need more information on how to perform a process, you can refer to this diagram for a quick update.
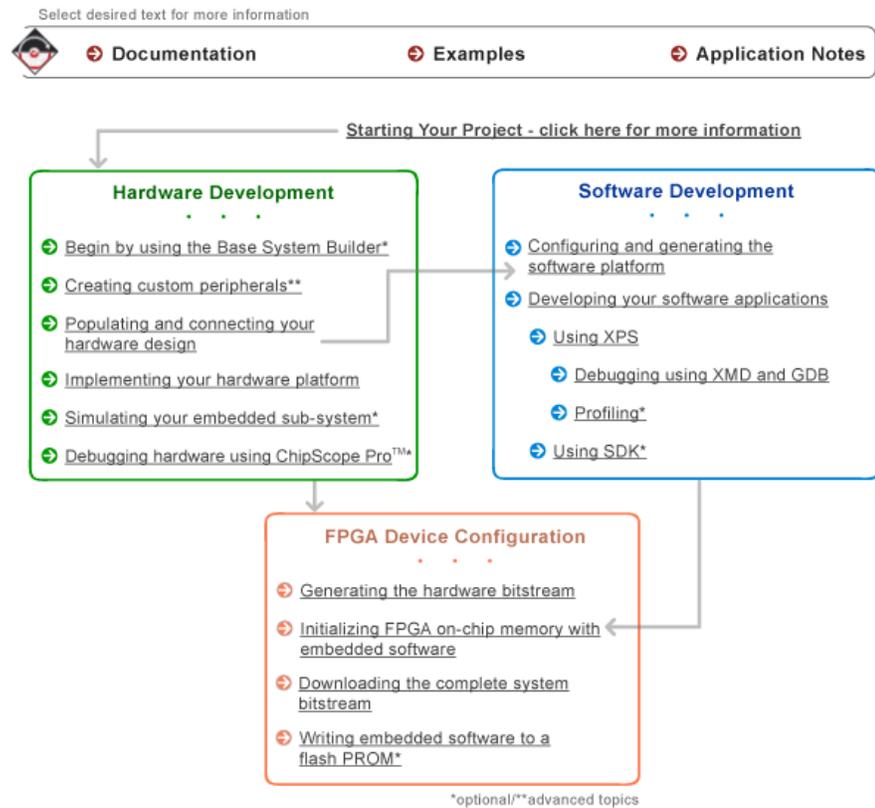
Figure 3-6: **XPS Startup Flow Diagram**

Take a Test Drive!

Note: If you can't see the Platform Studio tab, select Help > View Startup Flow Diagram.

1. With the Platform Studio tab selected, try clicking the Software Development, Hardware Development, and FPGA Device Configuration headings.

    You may find it interesting to read the help-topic overviews for these parts of the design flow. You can navigate the HTML using the green arrows in the tool bar. Or, you can go to View > Toolbars > HTML Browser.

2. Try clicking the Hardware Development topic *Begin by using the Base System Builder*.

    This presents material with which you might now be familiar, after reading Chapter 2, "Creating a New Project."

Console Window

The Console window (Figure 3-2, page 23) provides feedback from the tools invoked during runtime. Notice the three tabs: Output, Warning, and Error.

XPS Tools

In addition to the GUI, XPS includes all the underlying tools needed to develop the hardware and software components of an embedded processor system.

These include the following.

XPS Tools for Building Hardware and Software Components

*   The Base System Builder (BSB) Wizard, for creating new projects. The BSB dialog that appears on XPS start-up is also available from the tool bar.

    Click File > New Project.

*   The Hardware Platform Generation tool (Platgen), for generating the embedded processor system. To start Platgen, click Hardware > Generate Netlist.

*   The Simulation Model Generation tool (Simgen) generates simulation models of your embedded hardware system based either on your original embedded hardware design (behavioral) or finished FPGA implementation (timing-accurate).

    Click Simulation > Generate Simulation HDL Files to start Simgen.

*   The Create and Import Peripheral Wizard helps you create your own peripherals and import them into EDK-compliant repositories or XPS projects.

    To start the wizard, click Hardware > Create or Import Peripheral.

*   The Library Generation tool (Libgen) configures libraries, device drivers, file systems, and interrupt handlers for the embedded processor system, creating a software platform.

    Click Software > Generate Libraries and BSPs to start Libgen.

    Note: Libgen will be removed in a future version of XPS software.

*   The Xilinx Platform Studio Software Development Kit (SDK) is a complementary interface to XPS and provides a development environment for software application projects.

    Click Software > Launch Platform Studio SDK to open SDK. For your convenience, SDK has its own user interface to expedite software design tasks.

Take a Test Drive!

- Take a look at all the options that are available under the Hardware, Software, and Simulation items (located in the top menu bar).

XPS Directory Structure

For the Test Drive design you have begun, BSB has automated the project directory structure setup and started what can be considered a simple but complete project. The time savings that BSB provides during platform configuration can be negated, however, if you don't understand what the tools are doing behind the scene. Let's take a look at the directory structure BSB created and see how it could be useful as project development progresses.

Note: These files are stored in the location where you created your project file.

Directories

BSB automatically creates four primary directories, listed below. These directories are also shown in Figure 3-7.

`_ _xps`	Contains intermediate files generated by XPS and other tools for internal project management. You will not use this directory.
`data`	Contains the user constraints file (UCF). For more information on this file and how to use it, see the ISE UCF help topics at: www.xilinx.com/support/software_manuals.htm.
`etc`	Contains files that capture the options used to run various tools. This directory is empty because no actions outside of BSB have been performed.
`pcores`	Used for including custom hardware peripherals.

There are two directories that contain the BSB-generated test application C-source code, header files, and linker scripts, which were explored in an earlier Test Drive.

Underneath the main project directory you will also find a few files. Those of interest are shown in Figure 3-7 and are described as follows.

`system.xmp`	This is the top-level project design file. XPS reads this file and graphically displays its contents in the XPS user interface.
`system.mhs`	The system microprocessor hardware specification, or MHS file, captures textually the system elements, their parameters, and connectivity. The MHS file is the hardware foundation for your project.
`system.mss`	The system microprocessor software specification, or MSS file, captures the software portion of the design, describing textually the system elements and various software parameters associated with the peripheral. The MSS file is the software foundation for your project.

The MHS and MSS files can be thought of as the main products of the XPS GUIs. Your entire hardware and software system are represented by these two files.
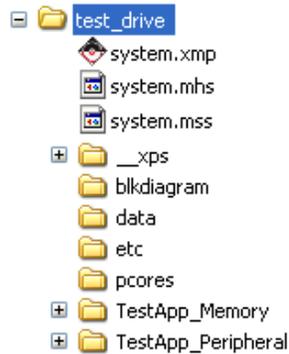
Figure 3-7: **BSB Wizard-Created Directories and Files**

Take a Test Drive!

1. Using a file explorer utility (such as Internet Explorer), navigate to the top-level directory for your project.

2. Open the various subdirectories and become familiar with the basic file set.

What's Next?

Now that you know your way around XPS, you're ready to begin working with the project you started in Chapter 2, "Creating a New Project." We will begin with the hardware platform.

The Embedded Hardware Platform

What's in a Hardware Platform?

The embedded hardware platform includes one or more processors, along with a variety of peripherals and memory blocks. These blocks of IP use an interconnect network to communicate. Additional ports connect to the "outside world." The behavior of each processor or peripheral core can be customized. Implementation parameters control optional features and specify what is ultimately implemented in the FPGA. The implementation parameters also define the addresses for your system.

Hardware Platform Development in Xilinx Platform Studio

About the Microprocessor Hardware Specification file

XPS provides an interactive development environment that allows you to specify all aspects of your hardware platform. XPS maintains your hardware platform description in a high-level form, known as the Microprocessor Hardware Specification (MHS) file. The MHS, an editable text file, and is the principal source file representing the hardware component of your embedded system. XPS synthesizes the MHS source file into Hardware Description Language (HDL) netlists ready for FPGA place and route.

The MHS File

The MHS file is integral to your design process. It contains all peripheral instantiations along with their parameters. The MHS file defines the configuration of the embedded processor system and includes information on the bus architecture, peripherals, processor, connectivity, and address space. For more information about the MHS file, see the "Microprocessor Hardware Specification (MHS)" chapter of the *Platform Specification Format Reference Manual*, available at www.xilinx.com/ise/embedded/edk_docs.htm.

Because of its importance, let's take a quick tour of the MHS file that was created when you ran the BSB Wizard.

Take a Test Drive!

1. Select the **Project** tab in the Project Information Area.

   Look under the Project Files heading to find MHS File: system.mhs, as shown in Figure 4-1.

2. Double-click the file name to open it.
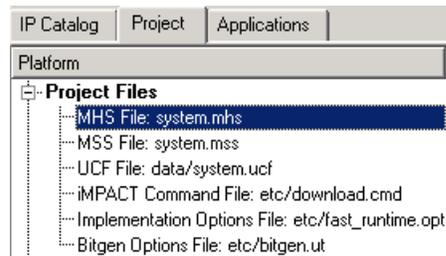
Figure 4-1: **MHS File**

3. Search for `xps_uart16550` the `system.mhs` file.

   Notice how the peripherals, their ports, and their parameters are configured in the MHS file.

4. Take some time to review other IP cores in your design.

5. When you are finished, close the `system.mhs` file.

The Hardware Platform in System Assembly View

The System Assembly View in XPS displays all hardware platform IP instances using an expandable tree and table format.

XPS provides extensive display customization, sorting, and data filtering capability so you can easily review your embedded design. The IP elements, their ports, properties, and parameters, which are configurable in the System Assembly View, are written directly to the MHS file.

Using the System Assembly View

Editing a port name or setting a parameter takes effect when you press Enter or click OK. XPS automatically writes the system modification to the hardware database, which is contained in the MHS file. The recommended method for editing the MHS file is to use the System Assembly View.

Note: Adding, deleting, and customizing IP are discussed in Chapter 5, "Creating Your Own Intellectual Property (IP)."

Generating the Hardware Platform

Generating the hardware platform is a two step process. First a netlist is generated, then a bitstream is produced. The bitstream is the configuration file that gives the FPGA its personality. The netlist is generated within XPS, the bitstream is built from within the ISE Project Navigator GUI.

Netlist and Bitstream Generation

Netlist Generation

When XPS is instructed to generate the netlist, it invokes the platform building tool, Platgen, which performs the following.

♦ Reads the design platform configuration MHS file.

♦ Generates an HDL representation of the MHS file written to `system.[vhd|v]` along with a `system_stub.[vhd|v]`. The system file is your MHS description written in HDL format.

   The file `system_stub` is a top-level HDL template file that is useful should you want to instantiate your processor system as a component in a larger, HDL-based

design. For more information about this process, see Appendix A, "Embedded Submodule Design with ISE."

♦ Synthesizes the design using Xilinx Synthesis Technology (XST).

♦ Produces a netlist file.

More information about PlatGen is available in the "Platform Generator (PlatGen)" chapter in the *Embedded System Tools Reference Manual*, available at www.xilinx.com/ise/embedded/edk_docs.htm.

Bitstream Generation

On successful completion of the Platgen process, the ISE Project Navigator GUI is used to generate the bitstream. This GUI reads the netlist that was created and in conjunction with the User Constraints File (UCF), they produce a BIT file containing the hardware design. Software patterns, if any, are not part of that bitstream (they are added later in SDK). When you use the BSB Wizard to create your initial hardware platform, it generates a UCF in the XPA project data folder.

What's Next?

Now you can start to customize your design. In the next chapter, you'll add your own IP to the Test Drive project.

Creating Your Own Intellectual Property (IP)

Adding Custom Logic to Your Design

So far, it has been fairly easy to develop an embedded system using XPS. Everything you have done up to this point has amounted to a series of mouse clicks because XPS has automated the process for you. Invariably, however, you will want to add some degree of customization to achieve your design goals. But this doesn't mean the process has to become hopelessly complex and slow. Even when customizing a system, XPS allows you to automate many steps that would otherwise be error-prone and time-consuming. That said, adding some custom logic (IP) to your Test Drive system would be a good next step. Let's get into some real design!

IP Creation Overview

If you think back to the XPS overview (see Figure 3-1, page 22 and Figure 3-6, page 28), the Bus Interface tab in System Assembly View shows connections among busses, processor, and IP. Any piece of IP you create must be compliant with the system you are designing. To ensure compliance, the following must occur:

1. The interface required by your IP must be determined.

   The bus to which your custom peripheral will attach must be identified. For example:

   Processor Local Bus (PLB)

   a. Processor Local Bus (PLB) version 4.6. The PLB provides a high-speed interface between the processor and high-performance peripherals. PLB v4.6 is used in both PowerPC and MicroBlaze processor systems.

   Fast Simplex Link (FSL)

   b. Fast Simplex Link (FSL). The FSL is a point-to-point "FIFO-like" interface. It can be used in MicroBlaze designs, but generally is not used in PowerPC systems.

2. Functionality must be implemented and verified.

   Your custom functionality must be implemented and verified, with awareness that common functionality available from the EDK peripherals library can be reused. Your stand-alone core must be verified. Isolating the core ensures easier debug in the future.

3. The IP must be imported to EDK.

   Your peripheral must be copied to an EDK-appropriate directory, and the Platform Specification Format (PSF) interface files (MPD and PAO) must be created, so other EDK tools can recognize your peripheral.

4. Your peripheral must be added to the processor system created in XPS.

How to Do It: Use the CIP Wizard!

You are probably saying to yourself, "This sounds complicated. How do I use XPS to make all this happen?" Fortunately, XPS offers another useful wizard, the Create and Import Peripheral (CIP) Wizard. The CIP Wizard assists with steps two and three above by walking you through the IP creation process. It sets up a number of templates for you to populate with proprietary logic. In addition to creating HDL templates, the CIP Wizard creates a peripheral core (pcore) verification project for Bus Functional Model (BFM)

verification. The templates and the BFM project creation are great for jump starting your IP development as well as ensuring your IP will comply with the system you created or will create.

The Create and Import Peripheral Wizard

By asking a few simple questions, the Create and Import Peripheral (CIP) Wizard greatly simplifies your custom peripheral creation process. Let's walk through creating a blank template for a piece of proprietary IP that you will design. For simplicity, most steps will accept default values, but you will have a chance to see all the possible selections you can make.

What You Need to Know Before Running the CIP Wizard

The wizard can create four types of PLB v4.6 peripherals using predefined IP interface (IPIF) libraries.

Supported PLB v4.6 Peripherals

- PLB v4.6 Slave for single data beat transfer
- PLB v4.6 Slave for burst data transfer
- PLB v4.6 Master for single data beat transfer
- PLB v4.6 Master for burst data transfer

You can also enable the legacy PLB v3.4 and OPB buses in the CIP wizard by checking the box Enable OPB and PLB v3.4 interfaces at the bottom of the Create Peripheral - Bus Interface wizard screen.

PLB Bus

To learn more about the PLB v4.6 interface, review the following documents appropriate to the bus to which your IP will connect:

```
$XILINX_EDK\doc\usenglish\mg_ug.pdf

$XILINX_EDK\doc\usenglish\sp026.pdf
```

Data Sheets

An easy way to find data sheets for a given element in the IP catalog is to right-click the IP element and select View PDF Data Sheet.

Take a Test Drive!

From the XPS menu bar, select Hardware > Create or Import Peripheral.

Create your new peripheral so that it has the characteristics described in the table below. **When in doubt about which value to enter, use the default value**.

Wizard Screen	Wizard Requested Input	Value to Enter
Create Peripheral Peripheral Flow (Screen 1)	Select Flow.	• Select the **Create templates for a new peripheral** option to begin creating your new IP. (default)
Repository or Project (Screen 2)	Specify the location to which you want to save the peripheral.	• Select the **To an XPS Project** option and browse to the location of the current project. (This will likely be pre-selected.)
Name and version (Screen 3)	Peripheral name and version number.	• Give the new peripheral the name `test_ip`. • Use the default version 1_00_a.
Bus Interface	Bus type.	Select Processor Local bus, PLB v4.6 (default).
IPIF Services	IPIF services requested.	• Under **Slave service and configuration** enable all options, *except* **User logic memory space**. • Under **Master service and configuration** leave the **User logic master** unchecked.
Slave Interface	Burst and cache-line support.	Leave the burst and cache-line support option unchecked.
FIFO Service	FIFO services requested.	Use defaults.
Interrupt Service	Configure interrupt handling.	Use defaults.
User S/W Register	Software register configuration.	Use defaults.
IP Interconnect (IPIC)	IP interconnect (IPIC) signals.	Use defaults. (The defaults should all checked except the 3rd, 4th and 5th boxes: Bus2IP_Addr, Bus2IP_CS and Bus2IP_RNW.) *Note:* You can click a signal name to view information about a signal you might want to adjust.

Wizard Screen	Wizard Requested Input	Value to Enter
Peripheral Simulation Support	Bus functional model simulation.	• Click **Generate BFM simulation platform for <Your Simulator>** where Your Simulator will be either ModelSim or NCSim. ***Note:*** If either of the two following conditions is not met, skip this step. • You must have the BFM toolkit installed, or you won't be able to select the BFM option. • You must have ModelSim or NCSim installed.
Peripheral Implementation Support	Peripheral implementation support.	Use defaults.
Finish	Create Peripheral, Finish.	• Review the details contained in the wizard screen text box. Note the interrupt address range given. • Click **Finish.**

Note: For more information about the Create and Import Peripheral Wizards, see the XPS help system (Help > Help Topics > Procedures for Embedded Processor Design > Creating and Importing Peripherals).

You can get IP interface documentation at www.xilinx.com/ise/embedded/edk_ip.htm.

What Just Happened?

The wizard worked! But you're probably not sure what it really produced. Let's stop for a moment and examine some concepts and the resulting output.

Intellectual Property Interface (IPIF)

EDK uses what is called PLB slave and burst peripherals to implement common functionality among various processor peripherals. These PLB slave and burst peripherals can act as bus masters or bus slaves. In the Bus Interface and IPIF Services Panel, the CIP Wizard asked you to define the target bus and what services the IP would need. The purpose here was to determine the PLB slave and burst peripheral elements your IP would require.

The IP Interface The PLB slave and burst peripherals are verified, optimized, and highly parameterizable interfaces. They also give you a set of simplified bus protocols. This is all IP Interconnect (IPIC), which is much easier to work with when compared to operating on the PLB or FSL bus protocols directly. Using the PLB slave and burst peripherals with parameterization that suits your needs greatly reduces your design and test effort because you don't have to reinvent the wheel. Figure 5-1 illustrates the relationship between the bus, a simple PLB slave peripheral, IPIC, and your user logic.
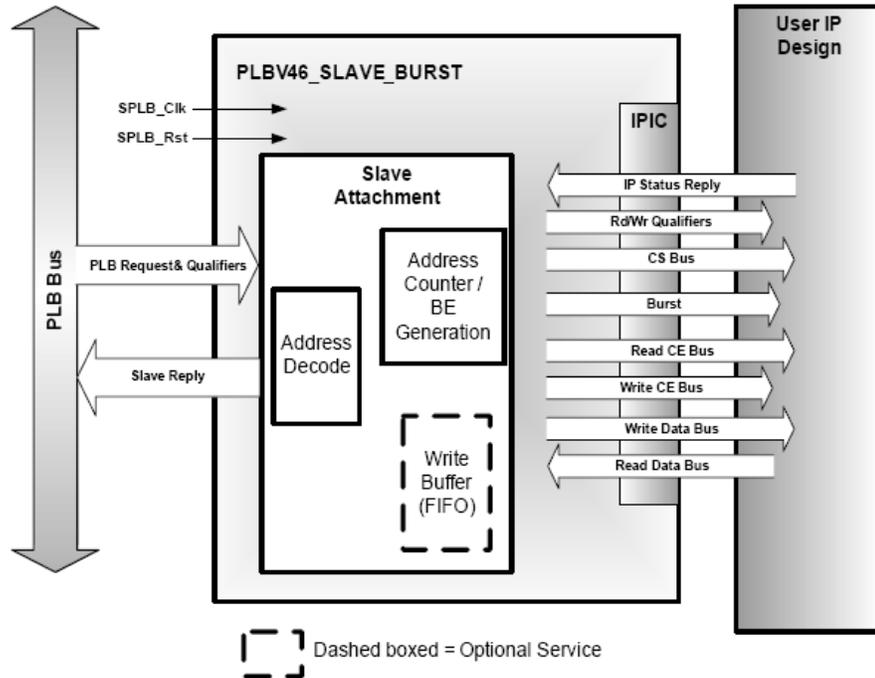
Figure 5-1: **PLB Slave/Burst Module in a Custom Peripheral**

Now, let's draw a parallel between what the wizard created and the boxes shown in Figure 5-1. The CIP Wizard created two template files that assist in IP connection. The top-level file is given the name you entered: `test_ip.vhd`. The second file, `user_logic.vhd`, is where your custom logic is to be connected.

A review of the directory structure and files that were created by the wizard reveals where the above-mentioned and other key files reside. See the `pcores` directory in your example project directory, shown in Figure 5-2.

Note: The list of files shown in Figure 5-2 is a partial list, and do not represent the full list of directory files.

Directory Structure and Files Created by the CIP Wizard
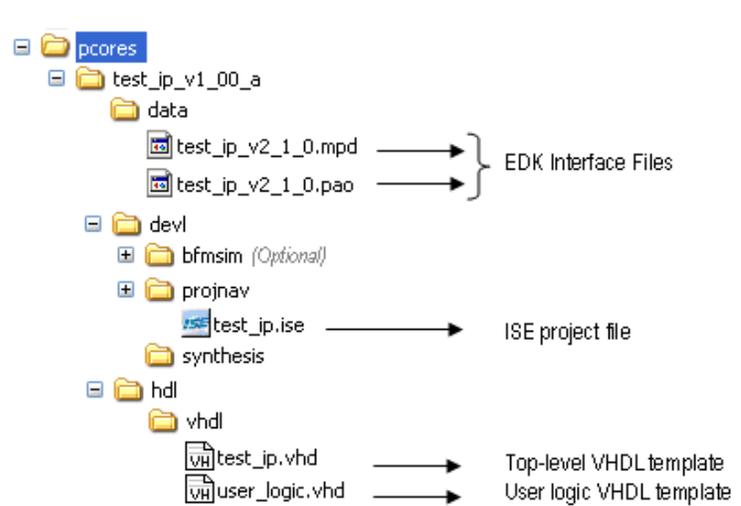
Figure 5-2: **Directory Structure Generated by the CIP Wizard**

Let's focus our attention on the two VHDL template files created by the wizard, `test_ip.vhd` and `user_logic.vhd`, shown in Figure 5-2.

The `user_logic` file makes the connection to the PLB v4.6 bus via the PLB slave/burst cores configured in `test_ip.vhd`. The `user_logic.vhd` file is equivalent to the "Custom Functionality" block and the `test_ip.vhd` file is equivalent to the PLB slave/burst blocks.

Figure 5-3 illustrates the relationship between the block diagram shown in Figure 5-1 and the generated files shown in Figure 5-2.
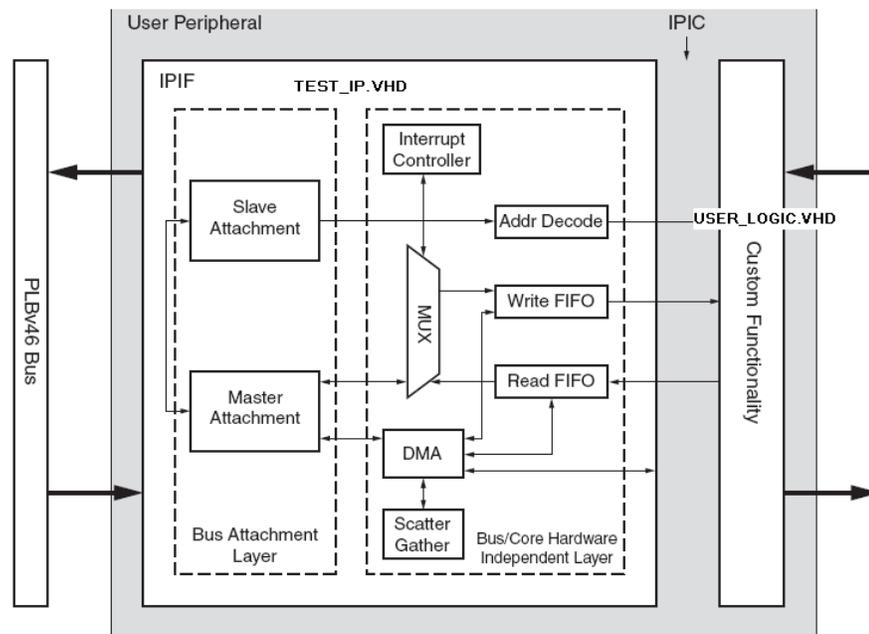
Figure 5-3: **Relationship of IP Module to Generated Files**

What's still lacking from both files is your proprietary logic.

Create and Import Peripheral Wizard Template Files

This summary of the interface provides the background you need to create some usable proprietary logic. Let's take a Test Drive to review the template files the Wizard has created for you.

Take a Test Drive!

1. In XPS, select File > Open and navigate to the `pcores\test_ip_v1_00_a\hdl\vhdl` directory.

   Here you will find the `test_ip.vhd` file and the `user_logic.vhd` file. (See Figure 5-2.)

2. Open the `user_logic.vhd` file.

3. Search for the value **entity user_logic** and find the occurrence that appears as shown in Figure 5-4.

   Note: A quick way to search for this value on a Windows platform is to use the search function keys (Ctrl + F).

```
111  entity user_logic is
112    generic
113    (
114      -- ADD USER GENERICS BELOW THIS LINE ---------------
115      --USER generics added here                                    ← Insert User Value
116      -- ADD USER GENERICS ABOVE THIS LINE ---------------
117
118      -- DO NOT EDIT BELOW THIS LINE --------------------
119      -- Bus protocol parameters, do not add to or delete
120      C_SLV_AWIDTH                  : integer              := 32;
121      C_SLV_DWIDTH                  : integer              := 32;
```

Figure 5-4: **User_logic.vhd Template File**

Wherever user information is required in the two template files (`<ip core name>.vhd` and `user_logic.vhd`), you will find comments indicating the type of information required and where to place it.

Because the templates create CoreConnect-compliant structures, you will not add any additional logic to your Test Drive project. However, it would be a good idea to view the bare interface setup and operation for future understanding.

Intellectual Property Bus Functional Model Simulation

Note: This is optional, but recommended.

Note: If you made no selections in the wizard screen for BFM simulation (see "Peripheral Simulation Support," page 40), skip to the Test Drive section "Running the CIP Wizard to Re-import test_ip into Your XPS Project," page 47.

The best thing you can do to understand BFM Simulation options is to explore the BFM project created for you by the CIP Wizard. So, let's take another Test Drive.

Take a Test Drive!

Note: If, in the CIP Wizard, you selected the check box to create the BFMs, you must close your XPS project before proceeding with the following steps.

If you elected to create the BFMs, the CIP Wizard created a sub-directory to your `\pcores\test_ip_v1_00_a\devl\` directory called `bfmsim`, in which it saved the XPS BFM simulation project called `bfm_system.xmp`.

Open the project `bfm_system.xmp` from XPS. What you see will be similar to what is shown in Figure 5-5.
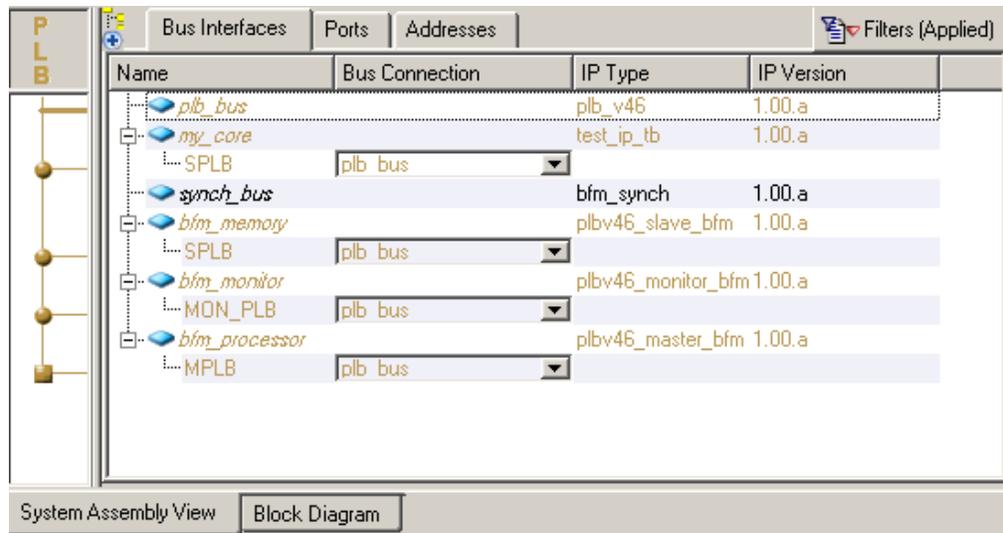
Figure 5-5: **XPS BFM User PCORE Simulation Project**

1. Select Project > Project Options and click the HDL and Simulation tab.

2. Select the HDL format in which you would like to simulate. We will use the default, VHDL.

3. Select the simulator you are using, either ModelSim or NCSim. We will use the default, ModelSim.

4. You should have your EDK simulation libraries compiled and pointing to the proper locations.

   a. If so, open the Application Preferences dialog (select Edit > Preferences > Application Preferences) and verify the location for the EDK and ISE libraries.

   b. If you have not compiled these, click Simulation > Compile Simulation Libraries and follow the steps given in the Simulation Library Compilation Wizard.
   For more information regarding simulation library compilation, refer to the XPS Help topic, Procedures for Embedded Processor Design > Simulation > Compiling Simulation Libraries in XPS.

5. BFM only offers Behavioral Simulation, so leave the Simulation Model selection set to its default.

6. Select OK when your have finished setting up the simulation options.

Simulation Model Generator (Simgen)

7. Select Simulation > Generate Simulation HDL Files to run the Simulation Model Generator (Simgen) for this test project.

Simgen creates a `simulation\behavioral` directory structure under the `bfmsim` directory. The behavioral directory contains the HDL wrapper files along with the DO script files needed to run a behavioral simulation.

8. Click Custom Button 1 ![icon] in the XPS GUI tool bar. The CIP Wizard configures this tool bar button when it creates the BFM simulation project. Custom Button 1 initiates the following:

a. Launches a bash shell to run a make file.

Bus Functional Compiler (BFC)

b. Using the previously set simulation options properly calls the CoreConnect Bus Functional Compiler (BFC) to operate on a `sample.bfl` file (see `<project name>\pcores\` `test_ip_v1_00_a\devl\bfmsim\scripts\sample.bfl` for more detail).

c. Invokes the simulator with the BFC output command files (`INCLUDE` or `DO` files) depending on the simulator to execute the commands in the `sample.bfl` file. The simulator waveform result will be similar that shown in Figure 5-6.
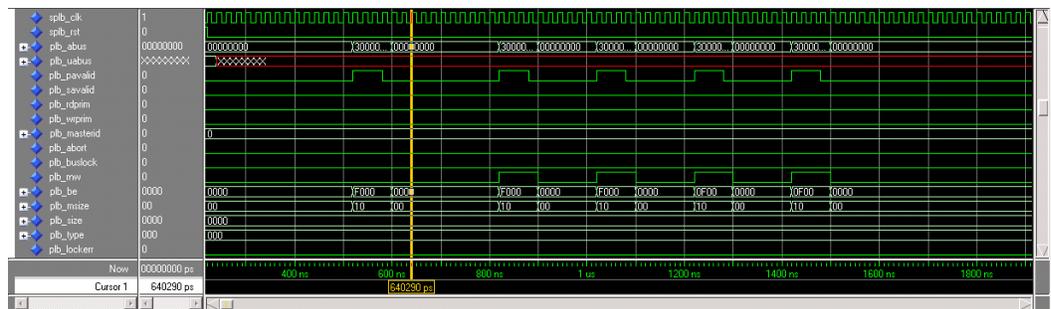
Figure 5-6: **BFM Waveform Simulation Results for sample.bfl @ t=640 ns**

What Just Happened?

The XPS tools just automated a lot of steps for you! Assuming this is your first time through the process, however, it may seem confusing. Let's quickly review what just happened.

1. The CIP Wizard created a set of HDL template files in the `<project_name>\pcores\test_ip_v1_00_a\hdl\vhdl` directory.

2. The CIP Wizard created a test project, which isolates your PCORE and allows you to verify its functionality with the bus before hooking it to a larger system. This project resides in the `<project_name>\pcores\test_ip_v1_00_a\devl\bfmsim` directory.

   This test project makes use of several BFMs supplied by the CoreConnect ToolKit. In this case, there is a model of the processor, bus, memory, and bus monitor, all connected to your core under development. The clear benefit is that you not only avoided having to create these models yourself, but XPS also made all the correct connections automatically. This saved you considerable time.

3. After generating the simulation platform, you can use Custom Button 1 to automate several, otherwise tedious steps in the simulation process. These steps run the `sample.bfl` through the CoreConnect Bus Functional Compiler, and must be performed to generate the command file the simulator uses. To find more information associated with these buttons, select Project > Customize Buttons and use the F1 help on the topic. The location of the make file used is given below.

In addition to compiling the `BFL`, the `make` file executed by Custom Button 1 calls the simulator with the command files to start simulation, simplifying the simulation launch and compilation process to a single button click.

How Can I Modify IP Created with the CIP Wizard?

The next logical question is how to make future adjustments, given that you will not be developing IP blocks without additional logic for very long. So, let's try making some alterations to your test IP.

Take a Test Drive!

1. In XPS, select **File > Open**, navigate to the `<project name>\pcores\test_ip_v1_00_a\devl\bfmsim\scripts` directory, and display all files.

2. Open the `sample.bfl` file.

Bus Functional Language (BFL)

BFM Script files

Roughly the first 160 or so lines of code set command aliases, making the command lines more readable. Source and destination memory is populated, and the various core features are tested. You can add or subtract commands to various sections as your core requires, or create a completely new `BFL` command file.

Note: If you create a new `BFL` file, you must also adjust the `bfm_sim_xps.make` file under the `bfmsim` directory to reflect your desired command file. For more information on the BFL commands, look in your EDK install area for the file `$XILINX_EDK\third_party\doc\xxxToolkit.pdf`, where `xxx` corresponds to a desired bus.

In addition to the `BFL` file, the CIP Wizard creates a corresponding `PCORES` directory under the `BFMSIM` project. Here you'll find a template for the BFM test bench. You can add to the template test bench as your core logic requires. This guide doesn't go into description on how to add test bench signals and stimulus to this file.

Now that you have a general understanding of how the BFM project can be used, and of its associated control files, it's time to add the validated PCORE to the overall system.

3. Close XPS (File/Exit) and relaunch XPS with your original system from the ISE Project Navigator by double-clicking on the XPS icon in the Sources Window (Figure 5-7).
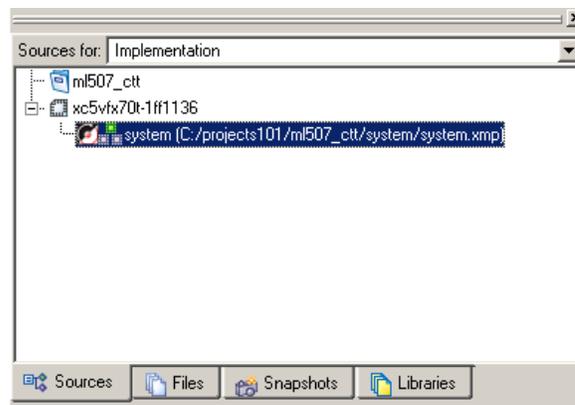

Figure 5-7: **Relaunching XPS from the ISE Project Navigator**

Next, you will add the new IP to the previously created embedded system.

Adding IP to Your Processor System

Not having generated any additional logic, you haven't changed the peripheral top-level interface. This guide will treat the `test_ip` core as if additional user ports were added. Why? Because additional logic signals are, more often than not, required for use of the PCORE.

With the assumption that you have added user ports, you should now re-run the CIP Wizard in the *import mode* to re-generate the correct EDK interface files (MPD and PAO). Doing this includes the newly added user ports and ensures that the `test_ip` peripheral can be used in XPS.

Take a Test Drive!

Before taking this test drive, let's do a quick review of where we are in the IP creation process.

The first time you ran the CIP wizard, you created the `test_ip` peripheral, set up the bus interface, and generated template files for it. Then (if you opted to do so) you ran BFM simulation to verify the basic design of your new peripheral.

Now you will add `test_ip` to your project, again using the CIP wizard. In the process, `test_ip` will be imported to an XPS-appropriate directory and the Platform Format Specification files (MPD and PAO) will be generated.

For more information platform specification format files, see the *Platform Specification Format Reference Manual* at www.xilinx.com/ise/embedded/edk_docs.htm.

Running the CIP Wizard to Re-import test_ip into Your XPS Project

- Open the CIP Wizard (Hardware > Create or Import Peripheral) and click Next.

  The values to enter for each wizard screen are shown in the table below. **When in doubt, use the default values**.

Wizard Screen	Value to Enter
Peripheral Flow	Select **Import existing peripheral**.
Repository or Project	Select **To an XPS project.**
Name and version	• Select **test_ip** from the drop-down list. • Enable the **Use version** option and accept 1.00.a. • If the peripheral already exists, a dialog pops up asking if you would like to overwrite it. Click Yes.
Source File Types	Indicate the types of files that make up the peripheral. Enable the HDL source files option.
HDL Source Files	• Select **Use existing Peripheral Analysis Order file (*.pao)** as the way to specify the HDL source files. • Browse to the `test_ip_v1_00_a\data\test_ip_v1_1_0.pao` file location, and open the file.

Wizard Screen	Value to Enter
HDL Analysis Information	This screen shows all the dependent library files and HDL source files that are needed to compile your peripheral, as well as corresponding logical libraries into which those files will be compiled. • You would click **Add Files** or **Add Library** if you want to add more files. For this custom peripheral, the wizard automatically infers all files required, based on the PAO file.
Bus Interfaces	• Select **PLBV46 Slave (SPLB)**.
SPLB : Port	This panel allows you to specify additional connections to the SPLB Bus Connector. Were it necessary to connect additional signals, you would do it here. • Because this template design is still empty, click **Next**.
The SPLB : Parameter	This screen defines any special bus interface parameters for your peripheral.
Identify Interrupt Signals	In this screen, you can specify any additional interrupts for your core to use, along with their signal characteristics. • Accept the defaults specified.
Parameter Attributes	Allows you to change parameter settings in the VHDL files produced earlier by the CIP Wizard. • Click **Next**.
Port Attributes	Allows you to change port settings in the VHDL files produced earlier by the CIP wizard.
Finish	Click **Finish**.

Updating User Repositories to Include test_ip

1. Select Project > Rescan User Repositories.

   After XPS completes the scan, a Project Local pcores category appears in the IP Catalog.

2. Expand the Project Local pcores listing in the IP catalog and double-click the `test_ip` peripheral core to add it to the system.

3. With the Bus Interface tab selected in System Assembly View, click the hollow bus connection symbol to complete the connection to the PLB.

4. Click the Addresses tab in System Assembly View.

5. Click the Generate Addresses button.

6. Generate the system netlist by clicking Hardware > Generate Netlist.

This completes the hardware portion of adding IP to your system.

What's Next?

You are now ready to create your software platform. The next chapter explains how EDK handles the software elements of your system and what files it uses to manage and store data about your embedded applications.

The Software Platform and SDK

Board Support Package

The Board Support Package (BSP) is a collection of files that defines the hardware elements of your system for each processor. The BSP contains the various embedded software elements, such as software driver files, selected libraries, standard I/O devices, interrupt handler routines, and other related features. Consequently, it is easiest to have SDK generate the BSP after the hardware system is populated with its processors and peripherals and after the address map is defined.

As with the hardware assembly, SDK allows you to specify all aspects of your software platform and manage your software applications.

MSS File and Other Software Platform Elements

Microprocessor Software Specification (MSS)

The hardware portion of your Test Drive project uses the MHS file to describe the hardware elements in a high-level form. XPS creates an analogous software system description in the Microprocessor Software Specification (MSS) file. The MSS file, together with your software applications, are the principal source files representing the software elements of your embedded system.

This collection of files, used in conjunction with EDK installed libraries and drivers, and any custom libraries and drivers for custom peripherals you provide allows SDK to compile your applications. The compiled software routines are available as an Executable and Linkable Format (ELF) file. The ELF file is the binary ones and zeros that are run on the processor hardware. Figure 6-1 shows the files and flow stages that generate the ELF file.
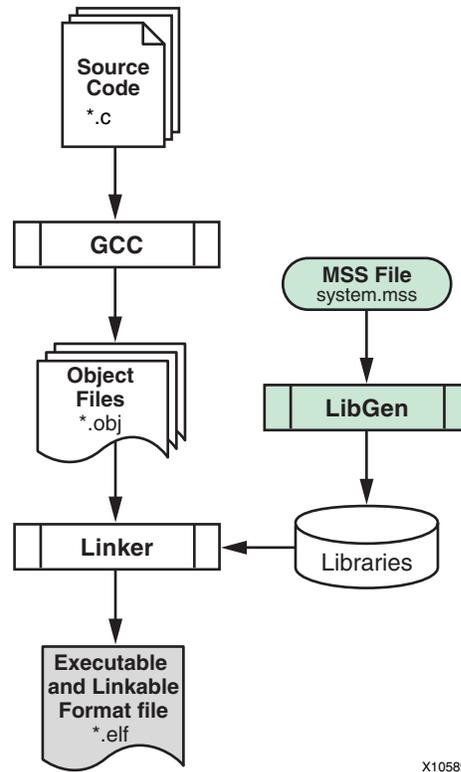
Figure 6-1: **Elements and Stages of ELF File Generation**

Platform Studio Software Development Kit

The Platform Studio Software Development Kit (SDK) was designed to facilitate the development of embedded software application projects. SDK has its own GUI and is based on the Eclipse open-source tool suite. The Platform Studio SDK is a complementary program to XPS; that is, from SDK, you can develop the software that the peripherals and processor(s) elements connected in XPS use.

SDK Overview

You must create an SDK project for each software application. The project directory contains your C/C++ source files, executable output file, and associated utility files such as the `make` files used to build the project. Each SDK project directory is typically located under the XPS project directory tree for the embedded system that the application targets. Each SDK project produces just one executable file, `<project_name>.elf`. Therefore, you may have more than one SDK project targeting a single XPS embedded system.

Take a Test Drive!

Launch SDK and Import Your Test Applications

For this project, you'll import the applications created earlier, when you ran the BSB Wizard.

1. Click Software > Launch Platform Studio SDK to open SDK.

When SDK opens, the Application Wizard opens (Figure 6-2) to assist in creating a software application project. (If the wizard does not open automatically, click Xilinx Tools > Launch Application Wizard.)

2. Select Import XPS Application Projects and click Next.

    Note: For future reference, notice that you could also choose to create a new SDK application.
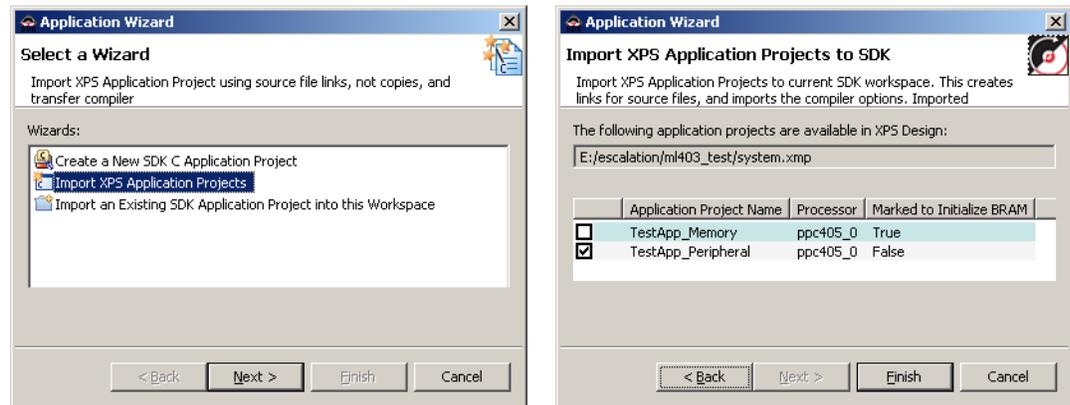
Figure 6-2: **Platform Studio SDK Project Creation Wizard**

The projects available in XPS are listed with check boxes for importation.

3. Select TestApp_Peripheral and click Finish.

    Note: The associated XMP file (top-level XPS project file) tells SDK which processors are present in the hardware platform and provides a pointer to the libraries for each processor. Notice that SDK compiles all libraries and generates the BSP automatically.

Adding Test Software for Your Custom IP

Next let's add some test software for the custom peripheral (test_ip) you created earlier. This entails doing the following steps.

♦ Locating the software test files for the core.

♦ Importing them into your TestApp_Peripheral application project.

♦ Editing the test_ip_selftest.c file to identify the base address for the test_ip core (because the TEST_IP_SelfTest routine requires a base address pointer). To obtain this information, you must refer to the xparameters.h file.

    (Does this seem confusing? Don't worry, you'll see how it works when you perform the steps below.)

♦ Rebuilding your projects. (SDK can be set to do this automatically.)

The following Test Drive takes you through the entire process.

Take a Test Drive!

Locating and Importing the Software Test Files

1. Click the C/C++ Projects tab in the upper left of the SDK main window.

2. In the C/C++ Projects Panel, right-click the TestApp_Peripheral project name and select Import.

3. In the Import dialog, select File system and click Next.

4. Browse to the `system\drivers` directory under your top-level project and locate the `test_ip_v1_00_a\src` directory.

    This is where the CIP Wizard created a few C files and a header file for your `test_ip` core, (Figure 6-3).

Figure 6-3: **Importing test_ip Software Files**

5. Check the boxes for all the source files (`test_*.*`), and click Finish.

Editing the `test_app_peripheral.c` File

1. In the C/C++ Projects tab on the left side of the SDK main window, locate the `test_ip_selftest.c` file. Double-click the file name to open it.

    The `test_ip_selftest.c` file contains the function definition for a `TEST_IP_SelfTest` routine, as shown in Figure 6-4. Notice the parameters this routine requires.

```
36  * @note    Caching must be turned off for this function to work.
37  * @note    Self test may fail if data memory and device are not on the same bus.
38  *
39  */
40 XStatus TEST_IP_SelfTest(void * baseaddr_p)
41 {
```

Figure 6-4: **Sample Software Template Created by the CIP Wizard**

As you can see, the `TEST_IP_SelfTest` routine requires a base address pointer, which you must provide.

You can find the `TEST_IP` base address value in the `xparameters.h` file, by doing the following:

a. In the C/C++ Projects tab, open the `ppc440_0_sw_platform/ ppc440_0/include` directory to display the `xparameters.h` file.

b. Double-click `xparameters.h` to open it in the editing window. Search for `TEST_IP_0_BASEADDR`, using the File Search dialog, shown in Figure 6-5..
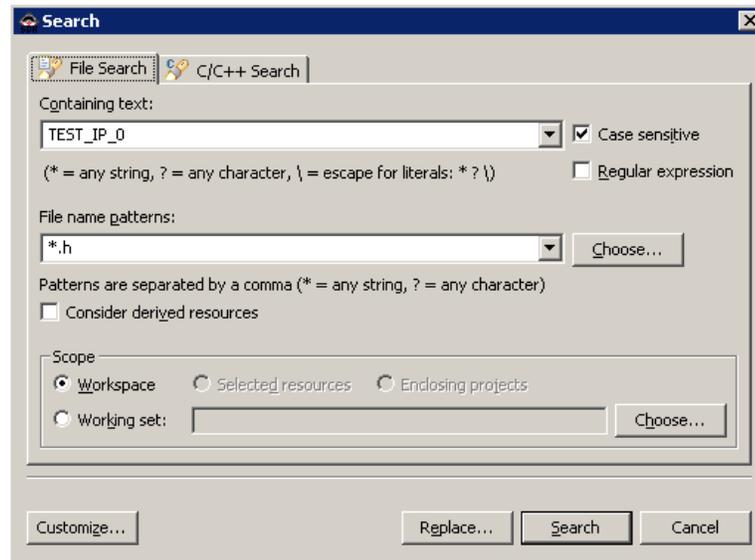
Figure 6-5: **File Search Dialog**

You now have the base address definition information necessary to add the function to the `TestApp_Peripheral.c` file.

2. In the `TestApp_Peripheral.c` file, insert the following line of code before the final print statement:

`TEST_IP_SelfTest(XPAR_TEST_IP_0_BASEADDR);`

Your `TestApp_Peripheral.c` file now looks similar to Figure 6-6.

Figure 6-6: **Code Insertion for TestApp_Peripheral.c File**

Rebuilding Your Projects

If the Build automatically option (in the tool bar under Project) is selected, your projects are updated when you save the `TestApp_Peripheral.c` file. If not, select Project > Build Project.

After the build is complete, note the creation of the `Debug` directory under the `TestApp_Peripheral` project. For now, your working `ELF` file for the project resides here. Note the `ELF` file location. You'll need it later for the Test Drive.

The C/C++ Build configuration settings allow control over the type of project you are building.

You can get more information in SDK by clicking Help > Help Contents and navigating to C/C++ Development User Guide > Reference > C/C++ Project Properties > Managed Make Projects > C/C++ Build > Build Settings.

Returning to XPS to Complete Your Project

This guide provides Test Drives to take your system through both simulation and implementation (the creation of a bitstream). To complete your project, you'll need to return to XPS and perform a few steps. We'll continue the Test Drive from there; but first, we'll provide a little background information.

Having completed some software development work using SDK, you must select a SW project to be loaded into block RAM using the Applications tab in XPS. What you select is determined by the design flow you plan on following.

If you plan to simulate your embedded system, your application needs to be loaded into block RAM (BRAM). You specify this by marking a block RAM for initialization in the Applications tab.

Completing the Project with XPS

Even if you are not planning on simulating your design, you must select a project in XPS to be marked to initialize BRAMs. It is fine to use **boot loop** for this. And, even though you will be using SDK to manage your software projects, this step needs to be performed once. The reason for this may sound a bit involved, but is important to understand. It has to do with the integration of the ISE Project Navigator GUI with SDK.

When a project is marked to initialize BRAMs, the **Update Bitstream with Processor Data** command in ISE Project Navigator copies the bitstream to the implementation subdirectory. When that is done, SDK can then access that bitstream and continue to update it as you modify your software. If you don't have a software project marked to initialize BRAMs, then the **Update Bitstream with Processor Data** command will fail, as will all further attempts to update that bitstream from SDK.

Let's take a look at a another scenario that could potentially pose a problem. Suppose two designers are working on this XPS project, one using XPS and another using SDK. The designer who saves the project last could overwrite the other designer's work. To avoid this situation, XPS identifies the potential conflict and creates a stable file condition that can be used going forward. Because SDK is the preferred software project manager, XPS only needs to know the location of the ELF file so that it can be merged with the FPGA for implementation and simulation.

By looking at the XPS Applications tab information, you can see that in addition to the software project you just worked on, there are additional projects. Quickly confirm that the following are present in your project before taking the following Test Drive:

- The default `ppc440_0_bootloop` project. The boot loop project minimally boots the processor by having it wake up and execute a single "jump to itself" instruction. It spins on this instruction until the debugger takes over.

- Projects created by the BSB Wizard, including `Project: TestApp_Memory` and `Project: TestApp_Peripheral`. You may recall that in the BSB Wizard, you chose to test both memory and other peripherals selected as part of the BSB process.

Perform the steps in the following Test Drive to select and configure the software so it can be simulated or downloaded to the FPGA or board memory device.

Take a Test Drive!

1. In XPS, select the Applications tab.

2. Right-click the **`Project: TestApp_Memory`** application and *deselect* Mark to Initialize BRAMs. (You're going to have the `Project: TestApp_Peripheral` application take care of this.)

3. Right click on the TestApp Peripheral project in XPS. You'll get the dialog shown in Figure 6-7.

    Note: The `TestApp_Peripheral` project was created in SDK, and XPS assumes that you are now actively managing this project from SDK. To work with the `TestApp_Peripheral` project, XPS will ask you to change it to an `ELF`-only project.
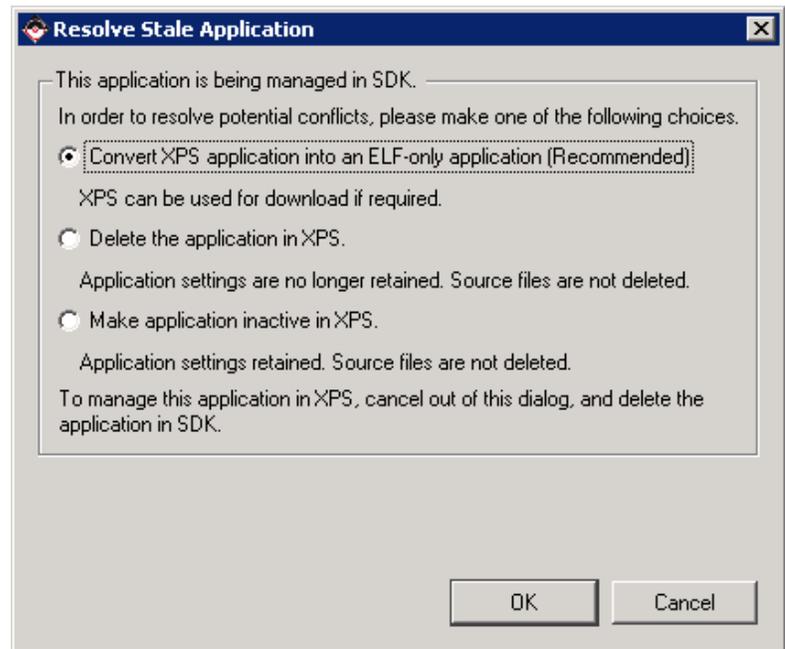
Figure 6-7: **XPS ELF File Management Option**

4. Select the option Convert XPS application into an ELF-only application.

    When you click OK, XPS continues to manage the data with which block RAM is initialized, but it turns the software project management function over to SDK.

5. Right-click this project to select it as the project to initialize block RAMs, as was done in the previous step.

    You should now see a check mark beside Mark to Initialize BRAMs in the right-click menu. Your Applications tab will look similar to the one shown in Figure 6-8.
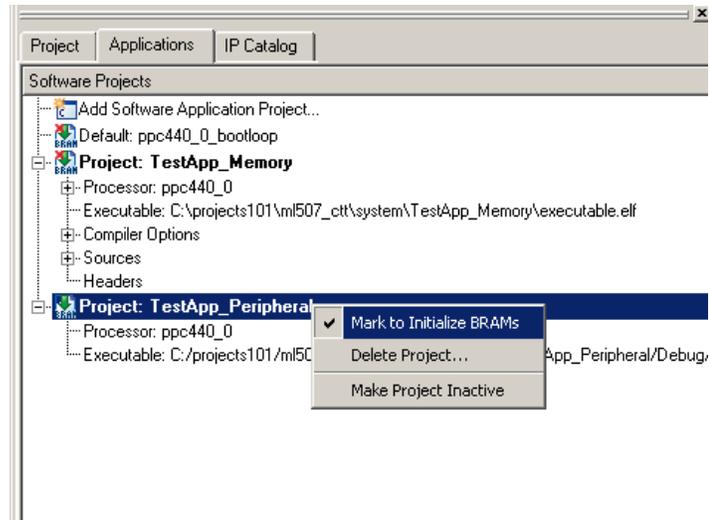
Figure 6-8: **Project Setting for BRAM Initialization**

6. In the Applications tab, right-click the Executable option in the
    `TestApp_Peripheral` project.

7. Browse to your SDK-created ELF file in the
    `\SDK_projects\TestApp_Peripheral\Debug` directory.

Debug and production ELF file locations

Note: A few steps earlier in the Test Drive, the SDK tool placed the `ELF` file in a `Debug` directory, which is for development. When your design moves to a release phase, a different directory (`Release`) can be used (depending on the C/C++ Build Project Properties). You can choose whether or not to use this structure because the file `ELF` location can be reassigned at anytime. Remember, that if you do reassign the build property, you must adjust the `ELF` file location in XPS as well.

What's Next?

Now that the software and hardware elements are created, they must be tested. This can be accomplished by downloading to a demo board or through simulation. Because our system and software application are relatively small, and because not everyone will be using the same demo board, this guide takes an opportunity to describe the simulation process.

Introduction to Simulation in XPS

Before You Begin

Recheck the simulation requirements from "Installation Requirements: What You Need to Run EDK Tools" in Chapter 1 to be sure the following conditions are satisfied.

- A mixed language simulator (Mentor Graphics ModelSim PE/SE or Mentor Graphics IUS) is required for the simulation steps.

  MXE does not support mixed language simulation. A mixed language simulator is required as the PPC440 model uses an IP-Protect model. After compilation, this model will be in a new simulation library called `secureip`.

- You should already have compiled the simulation libraries. If you haven't, just follow the procedure outlined in the XPS help system. To view this help section:

  1. Select Help > Help Topics.

  2. From the resulting HTML page, navigate through Procedures for Embedded Processor Design > Simulation > Compiling Simulation Libraries in XPS > Compiling Simulation Libraries in XPS.

    Note: The XPS help system is also available online at:

    www.xilinx.com/ise/embedded/edk_docs.htm.

Why Simulate an Embedded Design?

What are the benefits of simulating an embedded design? Let's take a look:

- Using simulation, you don't have to wait for hardware to be complete before testing your software. This provides for facilitated software development, which allows you to meet more aggressive project deadlines.

The Benefits of Simulating a Design

- Simulation provides insight into the internal workings of your system. Signals and register values are more accessible in a simulated system than those in a hardware design.

- Simulation also allows for complete control of your system. Conditions that may be difficult to create in a hardware setting are relatively easy to simulate.

As you have seen throughout this guide, XPS automates many mundane design details. So, you probably won't be surprised to learn that it does an excellent job of creating the simulation scripts and Hardware Description Language (HDL) files.

However, for some software designers, it may not be clear how to make use of the final, simulated project data. To help with that, this chapter takes you on a Test Drive through the simulation process.

EDK Simulation Basics

EDK supports simulation of your embedded system on Mentor Graphics® ModelSim® or Cadence IUS® logic simulators. Simulation is accomplished by exporting VHDL or Verilog HDL models of your embedded hardware platform design. The models include block RAM (BRAM) memory peripherals that you can initialize with your embedded software Executable and Linkable Format (ELF) file. EDK can generate your choice of:

- A behavioral model (based on your hardware platform specification alone)
- A post-synthesis structural model (simulating after the Generate Netlist step)
- A complete post-place-and-route, timing-accurate model

Simulation Stages   Verification through behavioral, structural, and timing simulation can be performed at specific points in your design process, as illustrated in Figure 7-1. The simulation model generation tool, Simgen, creates and configures specified HDL design files.
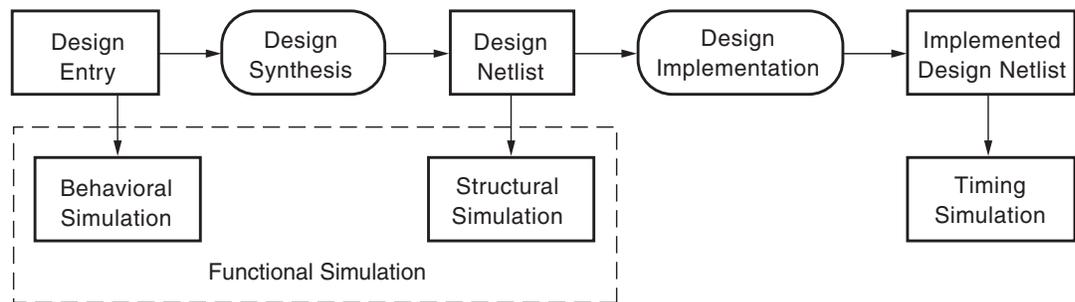
UG111_01_051005

Figure 7-1:   **FPGA Design Simulation Stages**

The simulators that support EDK require you to compile the HDL libraries before you can use them for design simulation. The advantages of compiling HDL libraries include speed of execution and efficient use of memory. It is assumed that your libraries are compiled at this point. If you need to compile the libraries, see "Before Starting" in Chapter 1.

For more information about simulation, including descriptions of behavioral, structural, and timing simulation, see the "Simulation Model Generator (Simgen)" in the *Embedded System Tools Reference Manual* at www.xilinx.com/ise/embedded/edk_docs.htm.

Simulation Considerations

When simulating your design, keep the following points in mind.

- Ensure that certain system values are specified.
- It is advantageous to change some settings to improve the simulation runtime.

Global Settings to Specify

Global reset, tristate nets, and clock signals all must be set to some value. Xilinx Integrated Software Environment (ISE™) tools provide detailed information on how to simulate your VHDL or Verilog design. Refer to the "Simulating Your Design" chapter in the ISE *Synthesis and Simulation Design Guide* for more information. A PDF version of this document is located at:
http://toolbox.xilinx.com/docsan/xilinx10/books/docs/sim/sim.pdf

System Behavior and Improving Simulation Times

You should also be aware of system behavior. HDL simulation is slow when compared with a design running on hardware. To improve the simulation runtime, you can adjust some parameters for simulation-only purposes. For example, our Test Drive system contains an RS232_UART and a DCM. In the Test Drive section of this chapter, you'll see how to improve simulation time by increasing the baud rate for the UART and eliminating the DCM reset. (In a real-life system, you should always reset the DCM. For this example design, we're more concerned with getting illustrative results with the shortest amount of simulation time).

Helper Scripts

Xilinx has put a good deal of effort into making system simulation easier to perform. The tools understand how your system is connected and how all the HDL design files relate behind-the-scenes. The tools also have the ability to create simulator instruction files for the design under test. When you initiate the XPS tool bar command Simulation > Generate Simulation HDL Files, all this capability is enabled automatically.

In addition to this, XPS includes Helper scripts to simplify simulator usage. Helper scripts are generated at the test harness (or test bench) level to set up the simulation. When run, the Helper script performs initialization functions and provides instructions for creating waveform and list windows (Mentor Graphics ModelSim-only) using waveform and list scripts. The top-level scripts invoke instance-specific scripts.

Simplifying Simulation by Using Helper Scripts

Under the `simulation\<simulation type>` directory, you will find several command scripts for running simulation. The `system_setup.do` file is the starting point from which all other scripts are called. Commands in the scripts can be customized as desired. Editing the top-level waveform (`system_wave.do`) or list scripts allows you to select signals for inclusion or exclusion. They are all shown by default. For timing simulations, only top-level ports are displayed.

In XPS under Project > Project Options, in the HDL and Simulation tab there is a selection to instruct XPS to generate a test bench. If this option is enabled, the test bench will have instructions to perform the reset of your system. If the option is not enabled, a helper script is created in `system_setup.do` called `rst` that sets up your simulation clocks and resets the system.

Restrictions

The simulation utility, Simgen, does not provide simulation models for external memories, and it does not have automated support for simulation models. External memory models must be instantiated and connected in the simulation test bench and initialized according to the model specifications.

Take a Test Drive!

This Test Drive takes you through simulating your system and allows you to observe the hardware and software response of the recently created IP block to requests it receives.

Simulation Setup

For the RS232_UART peripheral, simulating at a 9600 baud rate requires extended simulation times, during most of which there is little happening. Accelerating the baud

rate by a factor of 100 reduces the time spent simulating by a similar amount. It also condenses the transition area for data, allowing you to assess the simulated behavior of the system more easily.

For the DCM, we will eliminate the time to acquire lock by tying the relevant input to the `proc_sys_reset_0` block high.

To accelerate the UART baud rate:

Accelerating UART Baud Rate

1. Launch SDK if not already open.
2. Open **TestApp_Periperal.c** (It may be necessary to first select Bus Interfaces.)
3. From the drop-down box, select the highest value possible, 921600, and click **OK**.
4. In the first line of code following main, change the baud rate of 9600 to 921600.
5. Save the file.

To accelerate DCM Reset Behavior:

Accelerating DCM Reset Behavior

1. In XPS System Assembly View, click the **Ports** tab.
2. Expand the `proc_sys_reset_0` block (see Figure 7-2).
3. Modify the connection to the `DCM_Locked` port by connecting it to `net_vcc`.
4. Examine the MHS file to ensure that the change made in the GUI was written into the MHS file.

Figure 7-2: **Ports Tab with proc_sys_reset_0 block Expanded**

Running Simulation

1. Select **Project > Project Options**, and under the HDL and Simulation tab, verify that Behavioral simulation is selected. If not, select it.
2. Select **Simulation > Generate Simulation HDL Files** to launch the Simgen tool and generate the simulation HDL files and helper scripts.

   When you invoke the command to generate the simulation HDL files, XPS creates the `simulation\behavioral` directory structure.
3. Use a file browser to locate and view the contents of the `\behavioral` directory. Here you find two primary file types: DO files and VHDL files.

a. Open the `system.vhd` file in a text editor.

This is your top-level file for the device under test. It contains all the signals and port mappings that comprise the design you are working with at this point. Scan through and familiarize yourself with the content of this file. When finished, close the file.

b. Open the `system_setup.do` file.

The system_setup.do Macro File

This macro file automates many of the steps executed during simulation. You see the results of this file after completing just a few steps. Note that the alias commands call additional `DO` files. You could add your own aliases to this file as well for custom simulation operations. Note the **w** alias for calling the `do system_wave.do` file. You will be asked to edit this file next. When finished with `system_setup.do` close this file and open `system_wave.do`.

c. The `system_wave.do` file displays the signals in your design. Many signals are generated by this file. To provide a little more focus for our simulation, comment out, using the pound (#) sign, the following lines of code:

```
#   do xps_bram_if_cntlr_1_wave.do
#   do xps_bram_if_cntlr_1_bram_wave.do
#   do Push_Buttons_5Bit_wave.do
#   do jtagppc_cntlr_0_wave.do
```

When you're finished with these modifications, save and close this file.

Before starting simulation it would be helpful to know more about the actual software implementation that will occur. A quick disassembly of the previously generated ELF file provides information about the executable address and assembly instructions that run the code.

1. In XPS, select Project > Launch EDK Shell.

The EDK shell is a cygwin-based command window you can use to run EDK specific commands.

2. At the EDK shell command prompt, change your directory:

`cd SDK_projects/TestApp_Peripheral/Debug/`

This is where your ELF file resides.

Tip: You can use the tab key to automatically populate the path.

3. To perform the disassembly, enter the following command:

`powerpc-eabi-objdump -S TestApp_Peripheral.elf >> TestApp_Peripheral.dis`

Disassembly command

This command calls the PowerPC object file display routine (`powerpc-eabi-objdump`) with intermixed source and disassembly information. The output is sent to the `TestApp_Peripheral.dis` file.

4. When the process is complete, close the EDK shell and return to XPS.

For more information about the `powerpc-eabi-objdump` routine, see the "GNU Utilities" appendix in the *Embedded System Tools Reference Manual* at www.xilinx.com/ise/embedded/edk_docs.htm.

Specific information about the switches that `powerpc-eabi-objdump` supports can be found by running **`powerpc-eabi-objdump –H`** on the command line.

5. In XPS, click Simulation > Launch HDL Simulator.

Providing you have an EDK-supported simulator installed, it appears with the `system_setup.do` file invoked. The simulator is now ready to compile and load your design.

6. Assuming you are using ModelSim, at the prompt, enter the following commands:

   c   Compile the designs.

   s   Load the design.

   w   Set up the waveform window.

   rst   Toggle the reset port and set the clock frequency to 100 MHz.

   run 3ms Simulation for 3 ms.

7. While your simulation is running, launch SDK and open the `TestApp_Peripheral.c` file, located in the `SDK_projects\TestApp_Peripheral` directory.

Using the TestApp_Peripheral Files

8. In this file, find the first interaction between the processor and the general purpose I/O. This should be line 69 of the source file where we're reading the GPIO status register (see Figure 7-3).
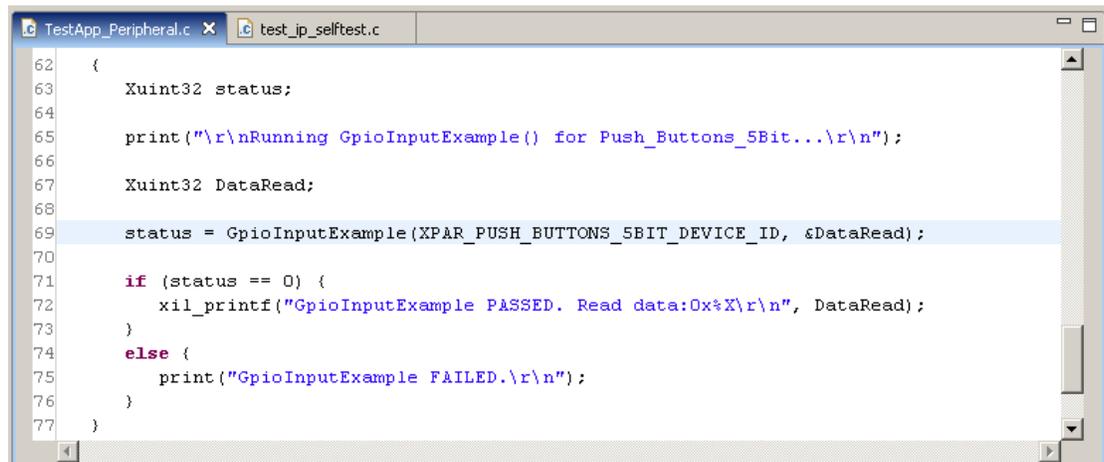

```
62  {
63      Xuint32 status;
64
65      print("\r\nRunning GpioInputExample() for Push_Buttons_5Bit...\r\n");
66
67      Xuint32 DataRead;
68
69      status = GpioInputExample(XPAR_PUSH_BUTTONS_5BIT_DEVICE_ID, &DataRead);
70
71      if (status == 0) {
72          xil_printf("GpioInputExample PASSED. Read data:0x%X\r\n", DataRead);
73      }
74      else {
75          print("GpioInputExample FAILED.\r\n");
76      }
77  }
```

Figure 7-3: **TestApp_Peripheral - Reading GPIO Peripheral**

9. Use a text editor to open the `TestApp_Peripheral.dis` file and search for the same line of code:

   `status = GpioInputExample(XPAR_PUSH_BUTTONS_5BIT_DEVICE_ID, &DataRead);`.

   There you'll find assembly code that appears similar to the following (actual address values may vary):

```
status = GpioInputExample(XPAR_PUSH_BUTTONS_5BIT_DEVICE_ID, &DataRead);
    ffffc3f8:    38  81  00  08      addi  r4,r1,8
    ffffc220:    38  60  00  00      li    r3,0
    ffffc224:    48  00  05  99      bl    ffffc7bc
                                           <GpioInputExample>
```

| C/C++ Source Code

Memory Address for Code Execution   Machine Code Execution Values   Assembly Operands

10. In your simulator, perform a signal search on `FFFFC220`.

Using the PLB signal `plb_abus` for the FFFFC220 value allows you to zoom in on a location at which to begin looking for execution of the `GpioInputExample` subroutine.

Armed with the disassembled source and the simulation waveform output (Figure 7-4), you'll be better able to continue stepping through the design and better able to understand its internal operation, as well as the hardware and software interaction.
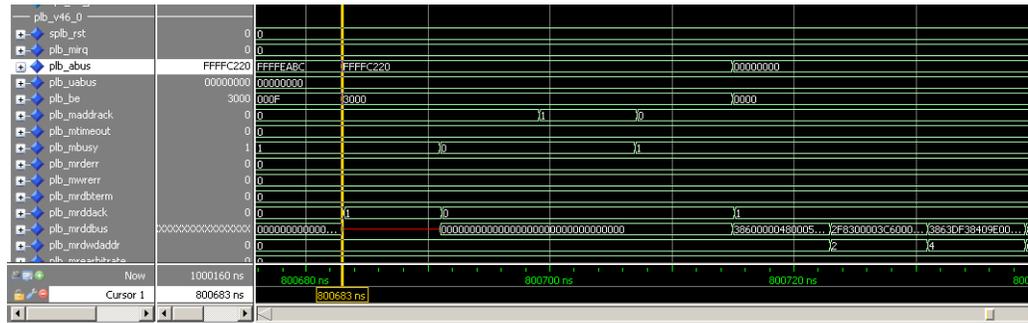
Figure 7-4:   **Simulation Output Results for `test_ip`**

Simulation Output

Now that your simulation is wrapped up, you need to go to XPS and in the Applications Tab mark Default: ppc440_0_bootloop. Initialize the block RAMs by right-clicking, and checking the menu that comes up.

Note: As previously mentioned, ISE needs to have a block RAM initialization file selected so that the.bit file will be copied to the proper location for SDK.

Implementing and Downloading Your Design

Implementing the Design

Having completed the design entry phase, you can now implement your design in hardware. We touched on this subject in Chapter 4, "The Embedded Hardware Platform," and as part of the earlier Test Drives, you generated your hardware netlist (see "Simulation Setup," page 59). As a result, XPS did most of the essential work required for implementation. To take the design from concept to reality, you must perform a few additional steps. This chapter provides information on what the tools have automated and on how to adjust those settings to suit your final design needs.

Netlist Generation Review

The MPD file contains all available ports and hardware parameters for a peripheral

Earlier you were prompted to select the Hardware > Generate Netlist menu item. This command causes the XPS Platform Generator (Platgen) utility to read the design platform information contained in the Microprocessor Hardware Specification (MHS) file, along with the IP attribute settings available from the respective Microprocessor Peripheral Definition (MPD) files. The output files from Platgen are Hardware Description Language (HDL) files, which can be found at `<project name>\hdl\`.

More information about the MPD file can be found in the *Platform Specification Format Reference Manual*, available at www.xilinx.com/ise/embedded/edk_docs.htm.

When you select Hardware > Generate Netlist, the Xilinx Synthesizer Technology (XST) synthesizes these HDL design files to produce the IP netlist (NGC) files, as shown in Figure 8-1.
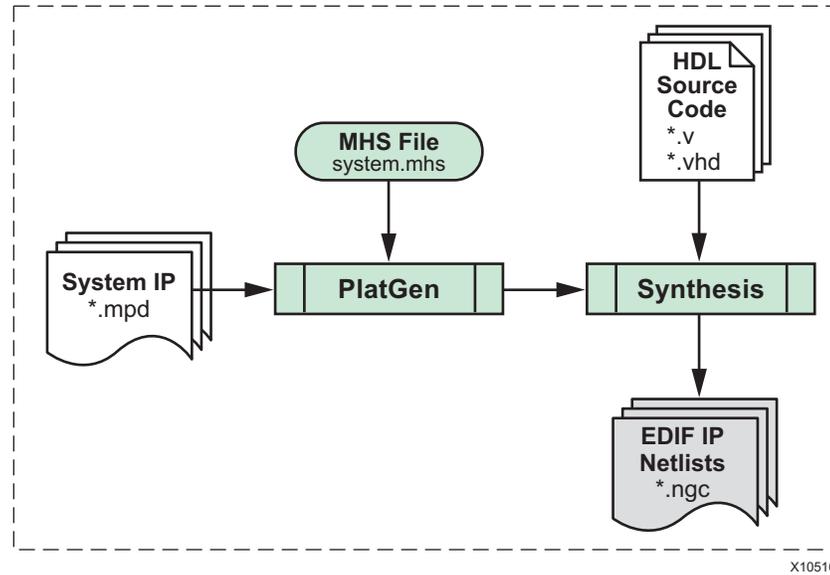
Figure 8-1: **Elements and Stages of Generating a Hardware Netlist**

For your general reference: the resulting NGC files reside at
`<project name>\implementation`. Note that you don't need to change these files.

ISE uses the NGC netlist files during design implementation, which occurs when you invoke the **Generate Programming File** from the ISE Project Navigator.

When you double-click Generate Program File, the following flow takes place, illustrated in Figure 8-2.
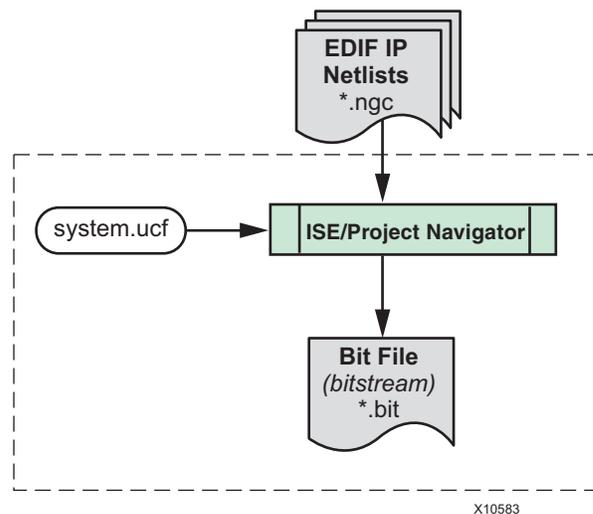
Figure 8-2: **Elements and Stages of Generating a Hardware Bitstream**

The NGC files and system constraints are processed through the remaining ISE tools (NGDBuild, MAP, PAR, and TRACE) and BITGEN from the ISE Project Navigator GUI.

The only Project Navigator file needed, in addition to the top-level processor design, is the User Constraints File (UCF). This file contains your design constraints (see Figure 8-3). If you are not familiar with FPGA design, the use of design constraints enables the tools to

identify and satisfy real-world limitations. Example constraints could be as simple as clock information or pin placement, or they could be complex placement and timing parameters that satisfy critical logic paths.
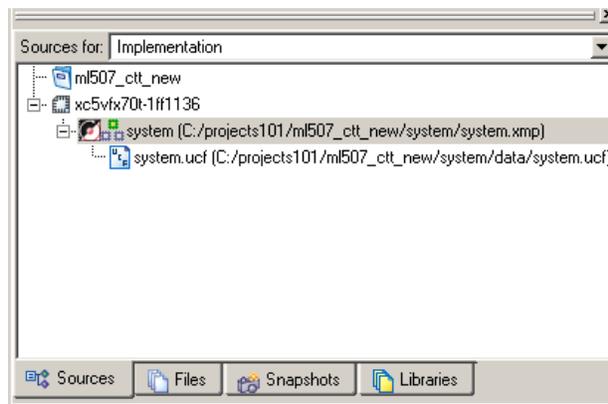
Figure 8-3: **Sample User Constraints File**

The Test Drive project provided in this tutorial is a processor-centric design; that is, it consists only of the embedded processor platform. There is no external logic associated with the processor system. Therefore, only a few constraints need to be added before bitstream generation.

Locating the UCF file   You already have a UCF file because when you ran the BSB Wizard, you selected a small set of constraints that were based on the board you selected. When you completed the Base System Builder (BSB) Wizard steps, the constraints were automatically generated. The UCF is located in the directory `<project name>\system\data`. More information about these constraints can be found in the ISE tools documentation *Constraints Guide*, available at: www.xilinx.com/support/software_manuals.htm.

Take A Test Drive!

Generating the Netlist and Bitstream

1. From SDK, open `TestApp_Peripheral.c` and change the baud rate back to 9600.

2. Save the file.

3. Select **Project/Add Source** in the ISE Project Navigator, and go to the system/data subdirectory.

4. Select the system.ucf file. Click OK in the dialog box, then OK again in the "Adding Source Files . . ." confirmation window.

   Your display should now look similar to the display shown in Figure 8-3.

5. While still in Project Navigator, select **File/Open** . . ." and open the same `ucf` that was just added, so you can view some constraints. Note that it will have both pinout and timing constraints.

6. Double-click **Generate Programming File** and observe the progress of your design's implementation in the Console Window.

7. When the Programming File generation completes, double-click **Update Bitstream With Processor Data** in the ISE Project Navigator GUI.

This sets the file system up appropriately so that SDK can find the bitstream that you will download in the next Test Drive.

FPGA Configuration

To boot up an embedded processor system, both hardware and software system components must be downloaded to the FPGA and program memory, respectively. We will show how that is done in SDK. Figure 8-4 shows the elements and stages of generating the embedded system bitstream, which is executed by selecting Configuration > Bitstream Settings in EDK.

```
Bit File                Executable
(bitstream)             and Linkable
  *.bit                 Format file
                           *.elf

              Data2MEM

          system.bit  JTAG    Hardware
```
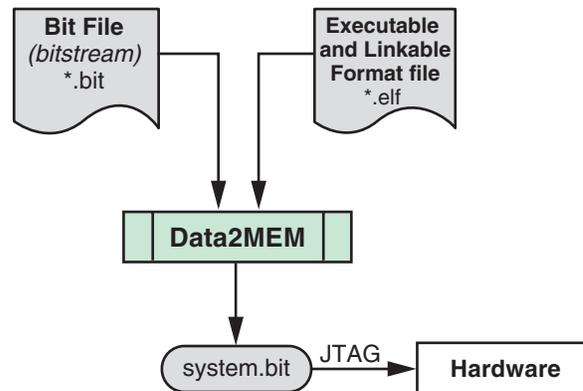
X10584

Figure 8-4: **Generating the Embedded System Bitstream**

Downloading the Hardware and Software System Components

During the prototype or development phase, you can download the hardware bitstream and software Executable and Linkable Format (ELF) file images by connecting a JTAG cable from your host computer to the JTAG port on your development board.

The Device Configuration > Download Bitstream menu command in SDK programs the FPGA with the bitstream. For software downloading: you can initialize software into the bitstream if it fits inside FPGA internal block RAM (BRAM) memory, or you can use the software debug tools, such as the XPS Software Development Kit (SDK), to download your program to the board.

The complete EDK program flow is shown in Figure 8-5.
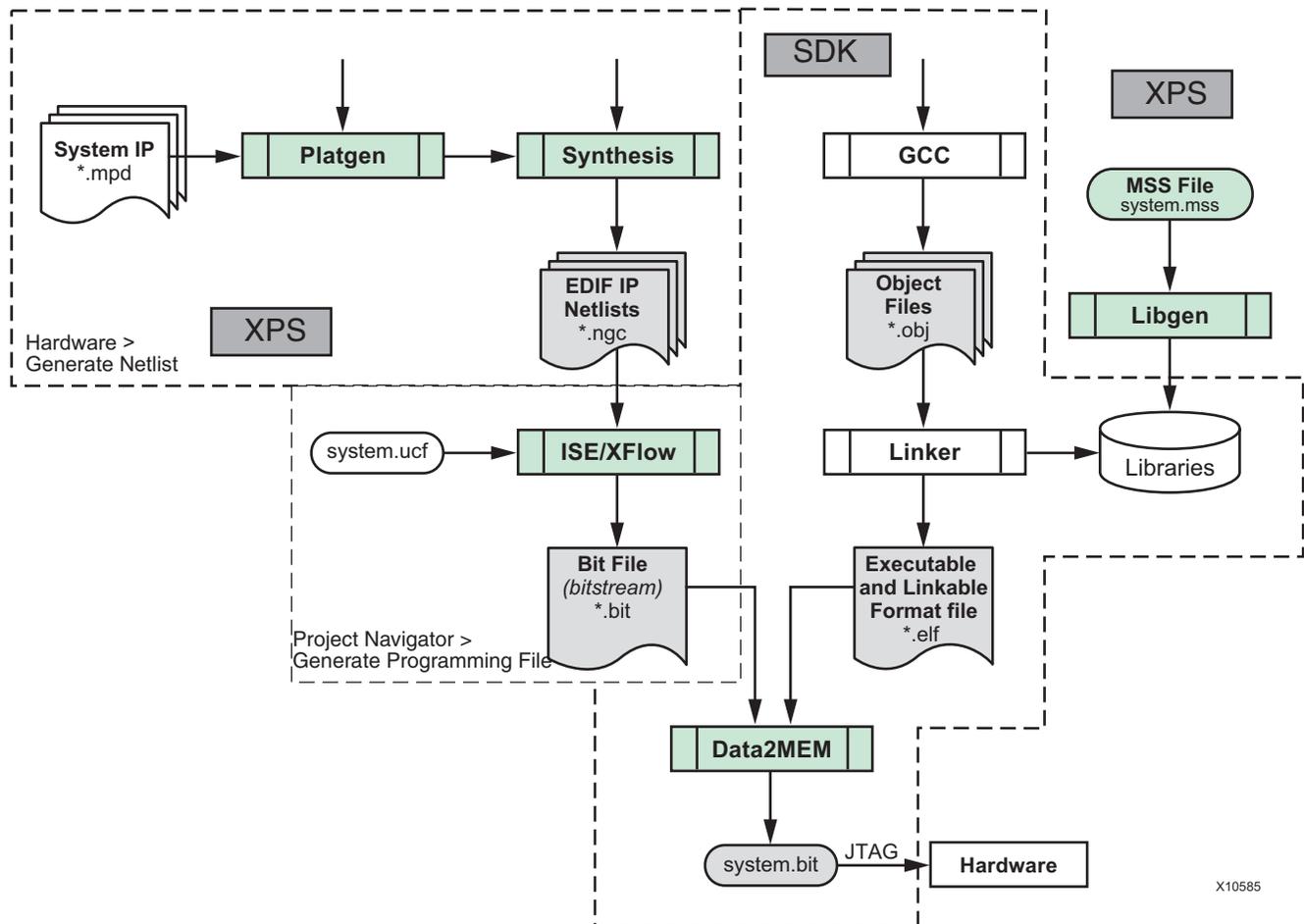
Figure 8-5:   **Elements and Stages of XPS and EDK Leading to FPGA Configuration**

Take a Test Drive!

1. At this point, your design netlists and bitstream are generated, the software is configured in the `TestApp_Peripheral` project.

2. In SDK, Select Device Configuration > Bitstream settings to merge the FPGA bitstream and `TestAppPeripheral.elf` files into a single bitstream file.

3. Ensure that the TestApp_Peripheral is selected. If it is not, select it now.

4. Ensure the serial and JTAG cables are connected, the development board is powered on, and a serial terminal is connected properly and set to the 9600 baud rate.

5. Click Device Configuration > Program FPGA. (Make sure that you executed the **Update Bitstream with Processor Data** command from the ISE Project Navigator.)

   After the bitstream is loaded to the board, an output on your serial terminal window shows the testing status of the peripherals included in your design. An output result from the shell `test_ip` peripheral you created earlier is included also.

```
-- Entering main() --

Running GpioInputExample() for Push_Buttons_Position...
GpioInputExample PASSED. Read data:0x0
*****************************
* User Peripheral Self Test
*****************************

Soft reset test...
    - write 0x0000000A to software reset register
    - soft reset passed

User logic slave module test...
    - write 1 to slave register 0 word 0
    - read 1 from register 0 word 0
    - slave register write/read passed

Packet FIFO test...
    - reset write packet FIFO to initial state
    - reset read packet FIFO to initial state
    - push data to write packet FIFO
      0x00000001
      0x00000002
      0x00000003
      0x00000004
    - write packet FIFO is not full
    - number of entries is expected 4
    - pop data out from read packet FIFO
      0x00000001
      0x00000002
      0x00000003
      0x00000004
    - read packet FIFO is empty
    - number of entries is expected 0
    - write/read packet FIFO passed

Interrupt controller test...
    - IP (user logic) interrupt status : 0x00000000
    - clear IP (user logic) interrupt status register
    - Device (peripheral) interrupt status : 0x00000000
    - clear Device (peripheral) interrupt status register
    - enable all possible interrupt(s)
    - write/read interrupt register passed

-- Exiting main() --
```

Chapter 9

Debugging the Design

So far, the Test Drive system has been fairly simple. However, as additional IP elements are added and more software is written, the system inevitably becomes more complex. In addition, because the system elements are encapsulated inside the FPGA and because the signals necessary for sufficient design analysis are inaccessible, debug could potentially become a challenge. But Xilinx has anticipated these difficulties and offers several methods and tools that allow you good visibility into both the hardware and software portions of your design, such as:

The Xilinx Debugging Tools

- Hardware debug capability using the Xilinx Microprocessor Debugger (XMD).
- Platform Studio Software Development Kit (SDK) software debugger communicates to the target processor through the XMD interface.
- The ChipScope™ Pro tool, which uses integrated logic analyzer hardware cores to communicate with the target design inside most Xilinx devices.

The Xilinx debug capabilities associated with Platform Studio tend to see the greatest level of use but may be the least understood. This chapter provides you with insight into this crucial function.

Xilinx MicroProcessor Debugger

The Xilinx MicroProcessor Debugger (XMD) is a design software utility that facilitates debugging software programs you create. XMD also helps you verify systems that use the microprocessors offered by Xilinx. You can use XMD to debug programs that run on a hardware board or that use the cycle-accurate Instruction Set Simulator (ISS). Figure 9-1 and Figure 9-2 show how XMD interacts with the target processor and the debug (host) software in use.

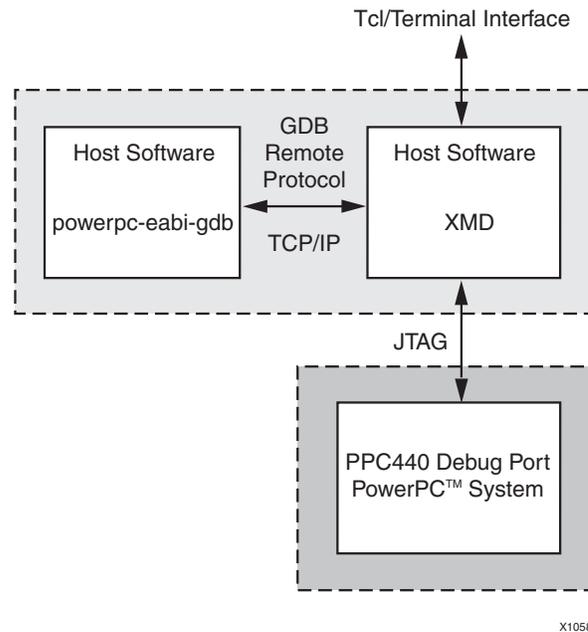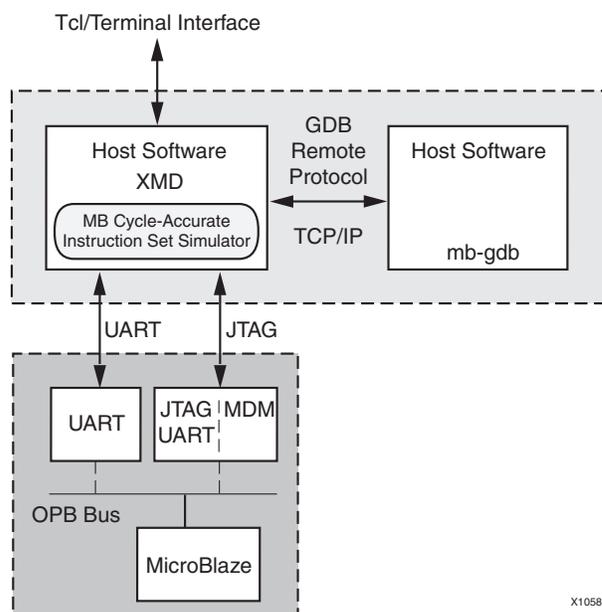*Figure 9-1:* **XMD PowerPC System Connection**

Figure 9-2: **XMD MicroBlaze System Connection**

XMD can stand on its own, but is usually used in conjunction with other utilities, such as the XPS or SDK GUIs. Typically, XMD connects to the target processor through a JTAG connection to the device under test. Communication and control are achieved using the TCP/IP protocol. In the images above, depending on the microprocessor you have selected (PowerPC or MicroBlaze) and on how you configured the system, XMD passes information from the device under test to the GUI (SDK) about its status. XMD also controls the operation of the processor, based on the requests you entered in SDK.

For more information about XMD, see the XMD chapter of the *Embedded System Tools Reference Manual*, available at www.xilinx.com/ise/embedded/edk_docs.htm

SDK Software Debugger

Platform Studio SDK presents an integrated environment for seamless debugging of embedded targets. Both MicroBlaze and PowerPC Executable and Linkable Format (ELF) files can be debugged with SDK.

Software debuggers such as the one provided in SDK enable you to monitor the execution of a program by controlling it through start, stop, and pause (breakpoint) operations. The software debugger may also allow some run-time control over program operation through monitoring and adjustment capabilities of the memory and/or variable values.

ChipScope Pro Tools

ChipScope Pro tools include several utilities that are integrated into a single application.

- ChipScope Pro Analyzer provides device configuration, trigger setup, and trace display for ChipScope Pro cores.
- ChipScope Pro Cores hardware debugging is accomplished through bus and arbitrary signal value monitoring, along with discrete control of inputs and output using the JTAG connection.

  The available cores include:

  - **Integrated Controller Pro (ICON)**: Provides a communication path between the JTAG port of the target FPGA and up to 15 other cores (IBA, VIO, ATC2, or MTC2).
  - **Integrated Logic Analyzer (ILA)**: A customizable logic analyzer core that monitors any internal signal in your design.
  - **Integrated Bus Analyzer (IBA)**: A specialized logic analyzer core designed to debug embedded systems that contain IBM CoreConnect bus interconnects, Processor Local Bus v4.6, Processor Local Bus v3.4, or On-Chip Peripheral Bus.
  - **Virtual Input/Output (VIO)**: A core that can both monitor and drive internal FPGA signals in real time.
  - **Agilent Trace Core 2 (ATC2)**: A debug capture core specifically designed to work with the latest generation Agilent logic analyzers. This core provides an external logic analyzer access to internal FPGA nets.

For more information about ChipScope Pro Tools features, benefits, and associated core descriptions, see the *ChipScope Pro Software and Cores User Guide* available at www.xilinx.com/literature/literature-chipscope.htm.

Platform Debug

Using the Debug Configuration Wizard

Used individually, you can see that the utilities described in the last several sections of this chapter are certainly helpful. However, when they are combined, they provide an even greater advantage—they give you a simultaneous and complete picture of hardware and software interactions within your embedded design. This ability is crucial to isolating the source of a bug.

Overview

The XPS Platform Studio Debug Configuration Wizard automates hardware and software debug configuration tasks common to most designs.

- To open the Debug Configuration Wizard, select Debug > Debug Configuration.

The wizard has the following primary screens.

System Explorer

The System Explorer (No. 1 in Figure 9-3) shows the options for selecting the debug utility that you want to configure. Use these options to navigate through and configure debug features for the available ChipScope cores and processors.

Information Tab

The Information tab (No. 2 in Figure 9-3) contains information about desired operations.

Console Window

The Console Window (No. 3 in Figure 9-3) displays output, warning, error, and information messages from the Debug Configuration wizard.
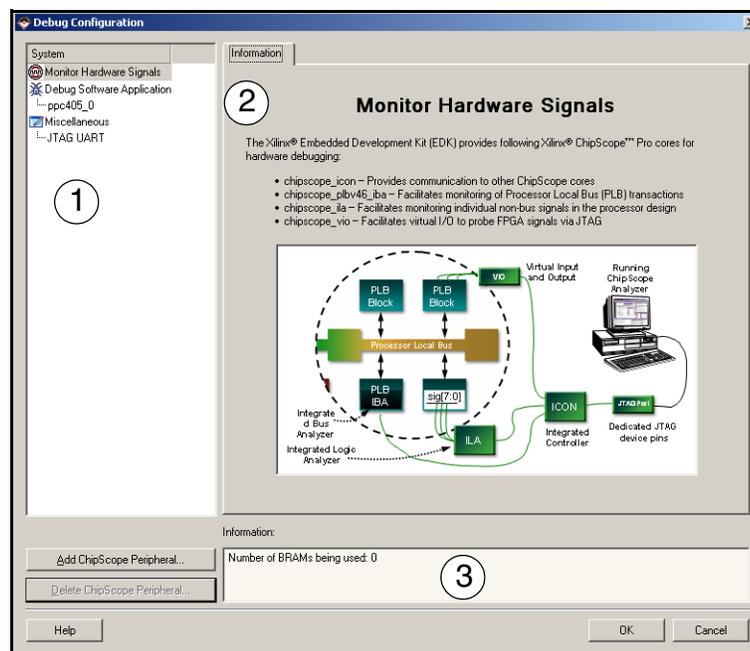

Figure 9-3: **Debug Configuration Wizard**

Hardware and Software Co-Debug

The Debug Configuration Wizard facilitates hardware and software co-debugging to accomplish the following:

- Connects IBA `trig_out` to the processor stop signal so the IBA can place the processor in the debug Halt mode. In short, this ChipScope signal stops processor execution.

  Whenever the processor is halted, the software debugger registers the state of the processor when it was stopped, allowing a hardware trigger to be correlated to the activity in software. Depending on the bus in use, the delay between the processor stop time and the registration of this event in the debugger can be as short as 11 clock cycles. As a result, it is highly likely that the software has stopped during the same subroutine that caused the hardware trigger event.

- Connects the `C405DBCSTOPACK` processor halted signal to one of the many IBA `trig_in` ports so that the halting of the processor can trigger the IBA to record samples. This condition registers with the bus analyzer any time the processor stops its execution.

  A debug event, such as a breakpoint occurrence, forces the processor to halt its execution. When this occurs, the bus analyzer registers the condition and presents all samples gathered up to that point, allowing you to correlate a software event to a state in hardware.

  Note: This applies only to the PPC405. MicroBlaze works in a similar fashion.

- Connects the processor instruction bus `C405DBGWBIAR[0:29]` to the IBA `trig_in` port so that the IBA can record the sequence of instructions. The processor and clock must operate at the same clock frequency. With this setting, a trace review is possible. The depth of the trace buffer is limited by the amount of on-chip BRAM available.

  Note: This applies only to the PPC405. MicroBlaze works in a similar fashion.

Let's take another Test Drive and put these concepts into practice.

Take a Test Drive!

Run the Debug Configuration Wizard in XPS

1. To Launch the Debug Configuration Wizard in XPS, click Debug > Debug Configuration.

2. In the wizard, select Add ChipScope Peripheral (located below the System Explorer pane). The Add New ChipScope Peripheral dialog appears.

3. Select the option To monitor PLB v4.6 bus signals (adding PLB IBA). This adds a ChipScope integrated bus analyzer core for the PLB bus.

4. Click OK.

   The configuration pane changes so you can specify the core configuration.

   a. Ensure the core is set up to monitor the PLB bus control, address, and data read/write signals.

   b. Select the check box to Enable Hardware/Software Co-debug.

   c. Accept the default value of 1024 samples.

5. Click the Advanced tab in the Debug Configuration pane.

   Review (but do not change) the available options. These provide finer control over what the ChipScope logic analyzer monitors and on what it will trigger. For more information on how to use these features and parameters, refer to the *ChipScope Pro Software and Cores User Guide* available at www.xilinx.com/literature/literature-chipscope.htm.

6. Click **OK** to close the wizard.

Review the Results

Notice that the Debug Configuration Wizard has done a lot of work for you. It has made connections appropriate to your design, which you can view in XPS and in your Microprocessor Hardware Specification (MHS) file.

1. Select the **Ports** tab in the System Assembly View and notice the two new cores: `chipscope_plbv46_iba_0` and `chipscope_icon_0`.

   If you expand the ports associated with these cores you'll see that the Debug Configuration Wizard made the necessary connections for you. See Figure 9-4.
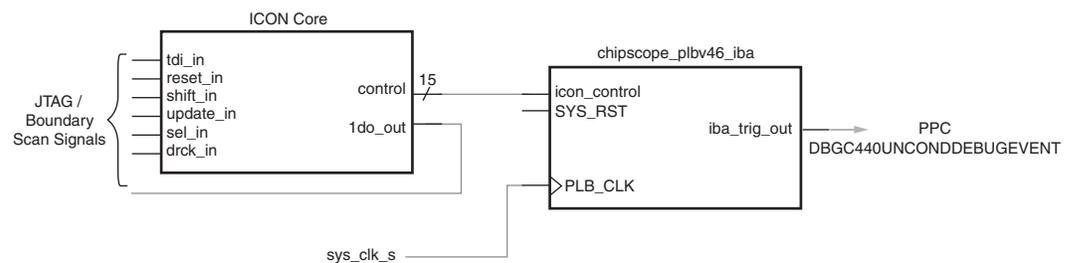
X10886

Figure 9-4:   **Debug Configuration Wizard Automatic Connections**

2. If you click the **Projects** tab and open the MHS file, you cans see that the wizard also added the `chipscope_plbv46_iba` connections:

   ```
   PORT chipscope_icon_control = chipscope_plbv46_iba_0_icon_ctrl
   PORT PLB_Clk = sys_clk_s
   PORT iba_trig_out = ppc440_0_DBGC440UNCONDDEBUGEVENT_chipscope
   ```

   Notice that the output from the IBA core is connected to the `DBGC440UNCONDDEBUGEVENT` input on the PowerPC core. A high pulse on this signal stops the processor.

ChipScope processor stop

   Below is the signal required to create the hardware-software cross triggering capability:

   `DBGC440UNCONDDEBUGEVENT`
   Indicates that external debug logic (in this case the presence of a trigger event in ChipScope) is causing an unconditional debug event.

Generate the Bitstream in XPS and Observe Platform Debugging

After adding the two new cores, you must create a new hardware bitstream.

1. In XPS, choose **Hardware > Generate Netlist**.

2. If you haven't already done so, go to the Applications Tab of XPS and mark Default: ppc440_0_bootloop as the application to initialize block RAMs.

3. In Project Navigator, double-click Generate Programming File.

4. In Project Navigator, double-click Update Bitstream with Processor Data.

5. In SDK, to observe the Platform debugging operation, add a `while(1)` statement to your `TestApp_Peripheral` application so the function runs continuously by doing the following:

   a. Open the `TestApp_Peripheral.c` file.

   b. In the file, look for the `int main (void) {` statement on line 45.

   c. Add the following code to the file (shown in bold type):

   ```
   int main (void) {

   while(1){

     print("-- Entering main() --/r/n");
   ```

   d. Locate the `print("-- Exiting main() --/r/n");` statement near the end of the file.

   e. Add the closing bracket as follows:

   ```
   print("--Exiting main() --/r/n");
     }
   return 0;
   }
   ```

   f. Save your file.

Download the Bitstream and Run Debug in SDK

Note: You must have a board connected to perform the following steps.

1. In SDK, select Device Configuration > Bitstream Settings, and set the specified ELF file to **boot loop**.

2. Select Device Configuration > Program FPGA to download the bitstream.

3. Ensure that your `TestApp_Peripheral` project is selected. From the Run menu select the Debug option. This launches the Debug Configuration dialog.

4. Click the New button at the bottom of the dialog. `TestApp_Peripheral` is automatically populated for the project type, and the C/C++ application can be found at `Debug\TestApp_Peripheral.elf`.

5. Select the Debug button at the bottom of the dialog to download the design to the board and switch to the Debug Perspective.

6. In the Debug Perspective, click on the Resume icon and observe the output in the serial terminal window. The software routine runs in a continuous loop.

ChipScope Pro Setup

1. Launch ChipScope Pro Analyzer.

2. Click the Open Cable/Search JTAG Chain icon, circled in Figure 9-5.

Figure 9-5: **Open Cable/Search JTAG Chain Icon**

A dialog appears, showing all of the devices in the JTAG chain.

3. Click OK.

4. To launch the Signal Import dialog, choose File > Import from the menu.

ChipScope CDC file location

5. Click Select New File and browse to
   `<project directory>\system\implementation\chipscope_plbv46_iba_0_wrapper`

   Here you will find the CDC file `cs_coregen_chipscope_plbv46_iba_0.cdc`.

6. Open this file in the ChipScope Logic Analyzer.

Waveform Window Setup

The ChipScope Logic Analyzer contains four main windows: New Project, Signals: DEV:4 Unit:0, Trigger Setup, and Waveform. In the following steps, you will work in the Waveform window.

1. In the Waveform window, select the first signal. Using the shift key, also select the last signal in the list to highlight all signals.

2. Right-click and select Remove from Viewer.

3. Drag the `PLB_ABUS` signal from the Signals pane to the Waveform pane. Do the same with the `PLB_SrdDBus`, and `PLB_wrDBus` signals.

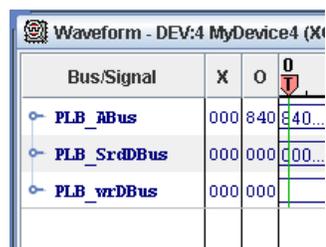   Your completed waveform window will look similar to Figure 9-6.

Figure 9-6: **ChipScope Pro Logic Analyzer Waveform Setup**

4. With this basic configuration setup, click the Trigger Now icon ![T!] in the ChipScope Pro tool bar.

This first instructs the ChipScope Pro Logic analyzer to sample system data for the previously configured signals and then provides a quick check that the ChipScope Pro logic analyzer, the ChipScope Pro IP elements, and the JTAG connection between these two items is capturing data.

5. By expanding the Match Units M0 through M4 in the Trigger setup window, you can see how the debug wizard has defined signals that are useful for advanced debugging. To define complex trigger conditions, you can enter specific PLB address values (match unit M2), PLB write and read data values (match units 3 and 4, respectively).

6. Set up the trigger and match units, as shown in Figure 9-7.

    When you select Apply Settings and Arm Trigger, you trigger on a UART access, which in turn asserts the IBA trigger out, and the processor is halted.
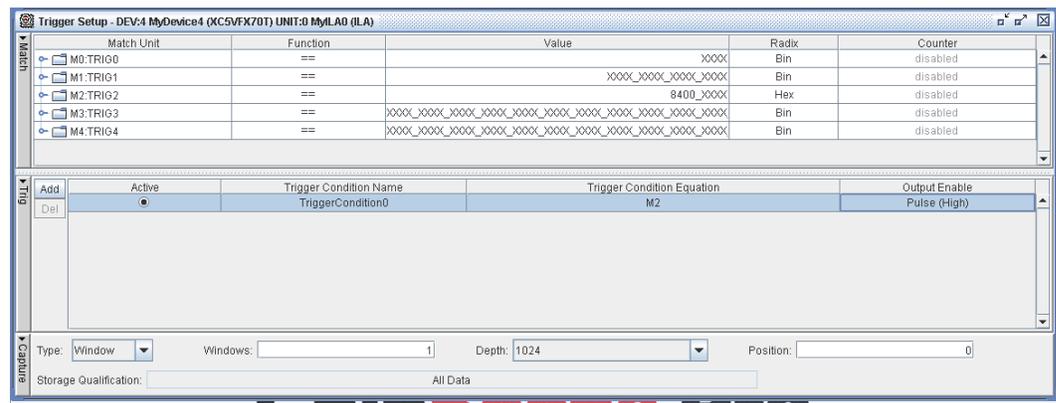
Figure 9-7: **ChipScope Trigger and Match Units**

Platform Debug: Triggering Between Hardware and Software

ChipScope allows you to combine conditions from all or some of the match units, so you can, for example, trigger on a specific data value that has been read or written at a specific address (as described in step 6 in the previous section). When ChipScope recognizes a complex trigger condition, the external trigger can then be passed to the processor core, halting code execution. In this manner, you can use a combination of SDK and ChipScope cross-triggering to help isolate and identify unexpected system behavior.

This concludes the hardware and software debug exercise as well as this version of the *EDK Concepts, Tools, and Techniques* guide.

We hope you have found this guide useful as you progressed from the beginning stages of XPS project development, through simulation, device download, and finally into both hardware and software debug.

More About BFM Simulation

When you took the Bus Functional Model (BFM) Simulation Test Drive in Chapter 5, "Creating Your Own Intellectual Property (IP)," you were asked to click **User Command Button 1** ∑ . The tools then ran through several `make` file scripts, which resulted in the simulation shown in Figure 5-3, page 42. This appendix provides a more detailed look at what happened, as well as information on how you can modify the routines for your own purposes.

Use a file explorer tool and navigate to the `<project_name>\pcores\test_ip_v1_00_a\devl\bfmsim\scripts` directory, shown in Figure A-1. Here you'll find a few scripts with which you should become familiar if you would like to modify the BFM for your own purposes.

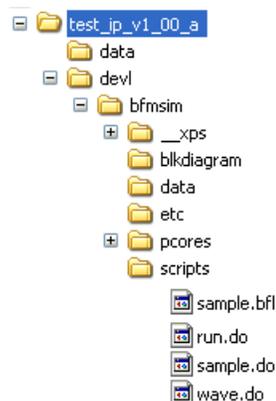*Figure A-1:* **BFM Directory and Files**

XPS created the `sample.bfl` file as part of the Create and Import Peripheral (CIP) Wizard process described in Chapter 5, "Creating Your Own Intellectual Property (IP)," As the name implies, `sample.bfl` is a sample bus functional language (BFL) script file. The `sample.bfl` file is the one you modify or recreate as another file for your own uses. With this understanding, open the `sample.bfl` file in a text editor to review what has been created for you. Again, this file contains some initial alias commands for human readability. Look for them in the BFL file in the order shown below:

1. Byte enable aliases.

2. Unit Under Test (UUT) aliases. These correspond to the same values given in the `drivers\test_ip_v1_00_a\src\test_ip.h` file. This file was also created as part of the CIP Wizard process. Note that although the base address may be different from the one in your actual system, the various register, interrupt and FIFO address values are the same because they are all set relative to the base address in the `test_ip.h` file.

3. Data aliases create readable values for numbers that may be used as part of the BFL.

4. Communication aliases are assigned for common operations in the BFL.

With the aliases set, `sample.bfl` begins to initialize various elements with the following type of command:

```
set_device(path = [string], device_type = [string])
```

5. The `set_device` command selects an Processor Local Bus (PLB) device model to initialize.

6. The `path` string is based on the various `*_wrapper` files created as part of the BFM structure. The string specifies the path of the model within the BFM system and test bench hierarchy.

7. The `device_type` specifies the type of model being initialized (`plb` or `opb _device` or `_arbiter` designations).

Having specified this information, memory is initialized using the `mem_init` command. With memory values initialized, testing of the UUT can be begin. The `sample.bfl` systematically tests the various elements you selected to include as part of the Create and Import Peripheral (CIP) Wizard process. The resultant waveforms first appear at approximately 640 ns in the BFM simulation output. (See Figure 5-3, page 42.)

With this understanding of the BFL, it should be fairly easy to see the connection between the `sample.bfl` and the `sample.do` files.

8. As part of the `make` file script, which was run when User Command Button 1 was invoked earlier, the BFL file is passed through the Bus Functional Compiler (BFC).

9. The BFC translates the input BFL into a simulator command file. Because this file is machine-generated, there is not much need to review the `sample.do` file, other than to note that there is a 1:6 translation (roughly) that occurs from the BFL input commands to the resulting output simulation command file.

    The benefit to you: a substantial time savings compared to manual entry of the simulator commands!

Glossary

BBD file

Black Box Definition file. The BBD file lists the netlist files used by a peripheral.

BFL

Bus Functional Language.

BFM

Bus Functional Model.

BIT File

Xilinx® Integrated Software Environment (ISE™) Bitstream file.

BitInit

The Bitstream Initializer tool. It initializes the instruction memory of processors on the FPGA and stores the instruction memory in blockRAMs in the FPGA.

block RAM (BRAM)

A block of random access memory built into a device, as distinguished from distributed, LUT based random access memory.

BMM file

Block Memory Map file. A BMM file is a text file that has syntactic descriptions of how individual block RAMs constitute a contiguous logical data space. Data2MEM uses BMM files to direct the translation of data into the proper initialization form. Since a BMM file is a text file, it is directly editable.

BSB

Base System Builder. A wizard for creating a complete design in Xilinx Platform Studio (XPS). BSB is also the file type used in the Base System Builder.

BSP

See Standalone BSP.

D

DCM

Digital Clock Manager

DCR

Device Control Register.

DLMB

Data-side Local Memory Bus. See also: LMB.

DMA

Direct Memory Access.

DOPB

Data-side On-chip Peripheral Bus. See also: OPB.

DRC

Design Rule Check.

DSPLB

Data-side Processor Local Bus. See also: ISPLB.

E

EDIF file

Electronic Data Interchange Format file. An industry standard file format for specifying a design netlist.

EDK

Xilinx Embedded Development Kit.

ELF file

Executable and Linkable Format file.

EMC

External Memory Controller.

EST

Embedded System Tools.

F

FATfs (XilFATfs)

LibXil FATFile System. The XilFATfs file system access library provides read/write access to files stored on a Xilinx SystemACE CompactFlash or IBM microdrive device.

FPGA

Field Programmable Gate Array.

FSL

MicroBlaze Fast Simplex Link. Unidirectional point-to-point data streaming interfaces ideal for hardware acceleration. The MicroBlaze processor has FSL interfaces directly to the processor.

G

GDB

GNU Debugger.

GPIO

General Purpose Input and Output. A 32-bit peripheral that attaches to the on-chip peripheral bus.

H

Hardware Platform

Xilinx FPGA technology allows you to customize the hardware logic in your processor subsystem. Such customization is not possible using standard off-the-shelf microprocessor or controller chips. Hardware platform is a term that describes the flexible, embedded processing subsystem you are creating with Xilinx technology for your application needs.

HDL

Hardware Description Language.

I

IBA

Integrated Bus Analyzer.

IDE

Integrated Design Environment.

ILA

Integrated Logic Analyzer.

ILMB

Instruction-side Local Memory Bus. See also: LMB.

IOPB

Instruction-side On-chip Peripheral Bus. See also: OPB.

IPIC

Intellectual Property Interconnect.

IPIF

Intellectual Property Interface.

ISA

Instruction Set Architecture. The ISA describes how aspects of the processor (including the instruction set, registers, interrupts, exceptions, and addresses) are visible to the programmer.

ISC

Interrupt Source Controller.

ISE

Xilinx ISE Project Navigator project file.

ISPLB

Instruction-side Peripheral Logical Bus. See also: DSPLB.

ISS

Instruction Set Simulator.

J

JTAG

Joint Test Action Group.

L

Libgen

Library Generator sub-component of the Xilinx Platform Studio technology.

LibXil Standard C Libraries

EDK libraries and device drivers provide standard C library functions, as well as functions to access peripherals. Libgen automatically configures the EDK libraries for every project based on the MSS file.

LibXil File

A module that provides block access to files and devices. The LibXil File module provides standard routines such as open, close, read, and write.

LibXil Profile

A software intrusive profile library that generates call graph and histogram information of any program running on a board.

LMB

Local Memory Bus. A low latency synchronous bus primarily used to access on-chip block RAM. The MicroBlaze processor contains an instruction LMB bus and a data LMB bus.

M

MDD file

Microprocessor Driver Description file.

MDM

Microprocessor Debug Module.

MFS

LibXil Memory File System. The MFS provides user capability to manage program memory in the form of file handles.

MHS file

Microprocessor Hardware Specification file. The MHS file defines the configuration of the embedded processor system including buses,peripherals, processors, connectivity, and address space.

MLD file

Microprocessor Library Definition file.

MPD file

Microprocessor Peripheral Definition file. The MPD file contains all of the available ports and hardware parameters for a peripheral.

MSS file

Microprocessor Software Specification file.

N

NCF file

Netlist Constraints file.

NGC file

The NGC file is a netlist file that contains both logical design data and constraints. This file replaces both EDIF and NCF files.

NGD file

Native Generic Database file. The NGD file is a netlist file that represents the entire design.

NGO File

A Xilinx-specific format binary file containing a logical description of the design in terms of its original components and hierarchy.

NPI

Native Port Interface.

O

OCM

On Chip Memory.

OPB

On-chip Peripheral Bus.

P

PACE

Pinout and Area Constraints Editor.

PAO file

Peripheral Analyze Order file. The PAO file defines the ordered list of HDL files needed for synthesis and simulation.

PBD file

Processor Block Diagram file.

Platgen

Hardware Platform Generator sub-component of the Platform Studio technology.

PLB

Processor Local Bus.

PROM

Programmable ROM.

PSF

Platform Specification Format. The specification for the set of data files that drive the EDK tools.

S

SDF file

Standard Data Format file. A data format that uses fields of fixed length to transfer data between multiple programs.

SDK

Software Development Kit.

SDMA

Soft Direct Memory Access

Simgen

The Simulation Generator sub-component of the Platform Studio technology.

Software Platform

A software platform is a collection of software drivers and, optionally, the operating system on which to build your application. Because of the fluid nature of the hardware platform and the rich Xilinx and Xilinx third-party partner support, you may create several software platforms for each of your hardware platforms.

SPI

Serial Peripheral Interface.

Standalone BSP

Standalone Board Support Package. A set of software modules that access processor-specific functions. The Standalone BSP is designed for use when an application accesses board or processor features directly (without an intervening OS layer).

SVF File

Serial Vector Format file.

U

UART

Universal Asynchronous Receiver-Transmitter.

UCF

User Constraints File.

V

VHDL

VHSIC Hardware Description Language.

VP

Virtual Platform.

VPgen

The Virtual Platform Generator sub-component of the Platform Studio technology.

X

XBD File

Xilinx Board Definition file.

XCL

Xilinx CacheLink. A high performance external memory cache interface available on the MicroBlaze processor.

Xilkernel

The Xilinx Embedded Kernel, shipped with EDK. A small, extremely modular and configurable RTOS for the Xilinx embedded software platform.

XMD

Xilinx Microprocessor Debugger.

XMK

Xilinx Microkernel. The entity representing the collective software system comprising the standard C libraries, Xilkernel, Standalone BSP, LibXil MFS, LibXil File, and LibXil Drivers.

XMP File

Xilinx Microprocessor Project file. This is the top-level project file for an EDK design.

![Xilinx logo]

XPS

Xilinx Platform Studio. The GUI environment in which you can develop your embedded design.

XST

Xilinx Synthesis Technology.

Z

ZBT

Zero Bus Turnaround™.

www.xilinx.com