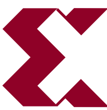
Embedded System Tools Reference Manual

Embedded Development Kit EDK 6.3i

UG111 (v3.0) August 20, 2004

"Xilinx" and the Xilinx logo shown above are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, and XC5210 are registered trademarks of Xilinx, Inc.



The shadow X shown above is a trademark of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, NanoBlaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, RocketIO, SelectIO, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Virtex-II EasyPath, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

All other trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx, Inc. devices and products are protected under U.S. Patents. Other U.S. and foreign patents pending. Xilinx, Inc. does not represent that devices shown or products described herein are free from patent infringement or from any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx, Inc. will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

The contents of this manual are owned and copyrighted by Xilinx. Copyright 1994-2004 Xilinx, Inc. All Rights Reserved. Except as stated herein, none of the material may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted 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. Any unauthorized use of any material contained in this manual may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Embedded System Tools Reference Manual
UG111 (v3.0) August 20, 2004

The following table shows the revision history for this document.

	Version	Revision
06/24/02	1.0	Initial Xilinx EDK (Embedded Processor Development Kit) release.
08/13/02	1.1	EDK (v3.1) release.
09/02/03	1.3	EDK 6.1 release.
01/30/04	1.4	EDK 6.2i release
03/12/04		Updated for service pack release.
03/19/04	2.0	Updated for service pack release.
08/20/04	3.0	EDK 6.3i release.

About This Guide

Welcome to the Embedded Development Kit. This kit is designed to provide designers with a rich set of design tools and a wide selection of standard peripherals required to build embedded processor systems using MicroBlaze, the industry's fastest soft processor solution, and the new and unique feature in Virtex-II Pro, the IBM ® PowerPC ® CPU.

This guide provides information about the Embedded System Tools (EST) included in the Embedded Development Kit (EDK). These tools, consisting of processor platform tailoring utilities, software application development tool, a full featured debug tool chain and device drivers and libraries, allow the developer to fully exploit the power of MicroBlaze and Virtex-II Pro.

Guide Contents

This guide contains the following chapters:

- Chapter 1, "Embedded System Tools Architecture"
- Chapter 2, "Xilinx Platform Studio (XPS)"
- Chapter 3, "Base System Builder"
- Chapter 4, "Create/Import Peripheral Wizard"
- Chapter 5, "Platform Generator"
- Chapter 6, "Simulation Model Generator"
- Chapter 7, "Library Generator"
- Chapter 8, "Platform Specification Utility"
- Chapter 9, "Format Revision Tool"
- Chapter 10, "Bitstream Initializer"
- Chapter 12, "GNU Compiler Tools"
- Chapter 13, "GNU Debugger"
- Chapter 14, "Xilinx Microprocessor Debugger (XMD)"

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

Resource	Description/URL
EDK Home	Embedded Development Kit home page, FAQ and tips. http://www.xilinx.com/edk
EDK Examples	A set of complete EDK examples. http://www.xilinx.com/ise/embedded/edk_examples.htm
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx solution records http://support.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes
Data Sheets	Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://support.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment http://www.support.xilinx.com/xlnx/xil_tt_home.jsp
GNU Manuals	The entire set of GNU manuals http://www.gnu.org/manual

Conventions

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

Typographical

The following typographical conventions are used in this document:

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	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *`design_name`*
	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.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II Handbook*.
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Table of Contents

Preface: About This Guide

Guide Contents 5
Additional Resources 6
Conventions 6
Typographical 7
Online Document 8

Chapter 1: Embedded System Tools Architecture

Tool Architecture Overview 19
Tool Flows 20
Hardware Platform Creation 20
Verification Platform Creation 21
Software Platform Creation 21
Software Application Creation and Verification 22
Some Useful Tools 23
Xilinx Platform Studio 23
Base System Builder 23
Create/Import IP Wizard 23
Platform Generator 24
Simulation Model Generator 24
Library Generator 24
Bitstream Initializer 24
Format Revision Tool 24
GNU Compiler Tools 24
MicroBlaze 24
PowerPC 25
Compiling with Optimization 25
Setting the Stack Size 25
Software Debugging 25
Dumping an Object/Executable File 25
Verifying Tools Setup 26
Tools Directory Path 26
For Solaris or Linux 26
For PC 26
Xilinx Alliance Software 26

Chapter 2: Xilinx Platform Studio (XPS)

Processes Supported 27
Tools Supported 28
Features 29
Project Management 29
Creating a New Project 29
Opening an Existing Project 30
Getting Help 30

XPS Interface 31

Editor Workspace 31

System Tab 32

Applications Tab. 32

Transcript Window (Output) 32

Platform Management. 33

Add/Edit Cores (Dialog) 33

Simulation Models 33

View MPD 33

View MDD 33

S/W Settings 33

Software Platform. 33

Processor and Driver Parameters 34

Library and O/S Parameters. 34

Software Application Management. 34

Adding Files 34

Deleting Files from Project 34

Editing Files 35

Mark Application for Downloading to BRAMs 35

Application to be Compiled Outside the XPS Environment. 35

Bootloop Software Applications 35

Xmdstub Software Applications 35

Compiler Options 36

Generating Linker Scripts 37

Flow Tool Settings and Required Files. 38

Compiler Options 38

Project Options 38

Required Files 40

Tool Invocation 40

Software Flow 40

Hardware Flow. 41

Merging Hardware and Software Flows and Downloading. 41

ISE Project Navigator Interface 41

Debug and Simulation 42

PBD Editor. 42

PBD Editor Interface 42

PBD Editor Workspace 43

System Tabs 44

Creating the Hardware Block Diagram 44

Adding a Component Instance to the System. 44

Naming an Instance 45

Setting Component Instance Parameters 45

Setting Symbol Properties. 46

Connecting a Component Bus Pin to a Bus. 46

Connecting Ports. 47

Viewing and Editing System Ports 47

Viewing and Editing All of the Ports in the System 48

Viewing and Editing Interrupts 48

Editing the Block Diagram 49

Selecting Objects 49

Viewing Object Information 50

Zooming in the Workspace. 50

Drawing Non-Electrical Objects 50
XPS “No Window” Mode 51
Available Commands 51
Creating a New Empty Project 52
Creating a New Project With Given MHS 53
Opening an Existing Project 53
Reading an MSS File 53
Saving Files and Your Project 53
Setting Project Options 53
Executing Flow Commands 54
Reloading an MHS File 55
Adding a Software Application 55
Deleting a Software Application 56
Adding a Program File to a Software Application 56
Deleting a Program File from a Software Application 56
Setting Options on a Software Application 56
Settings on Special Software Applications 57
Closing a Project and Exiting 58
Limitations and Workarounds 58
MSS Changes 58
XMP Changes 58

Chapter 3: Base System Builder

BSB Flow 59
Invoking BSB 59
Selecting a Starting Point 60
Selecting a Target Development Board 61
Selecting a Processor 62
Configuring Processor and System Settings 63
Selecting External Memories and I/O Devices: 64
Adding Internal Peripherals 67
Configuring Software Settings 67
Generating the System and Address Map 69
Output Files 69
Exiting BSB 71
Limitations 72

Chapter 4: Create/Import Peripheral Wizard

Invoking the Wizard 73
Creating New Peripherals 76
Identifying the Physical Location of Your Peripheral 77
Identifying Module and Version. 78
Select Bus Interface 79
Select IPIF Services 80
Generate Optional Files 88
Generating the Files Representing Your Design 91
Review EDK Peripheral Design Flow 91
Importing an Existing Peripheral 92
Identifying the Physical Location of Your Peripheral 93
Identifying Module and Version. 93
Select Source File Types 93

HDL Source Files 94
HDL Analysis Information 95
Bus Interfaces 98
Identifying Bus Interface Ports and Parameters 99
Interrupt Signals 100
Advanced Attributes on Ports and Parameters. 101
Netlist Files 103
Documentation Files 104
Finishing Peripheral Import 104

Organization of Generated Files 105

Limitations 107
Create Peripheral Mode 107
Import Peripheral Mode 107

Chapter 5: Platform Generator

Tool Requirements 109

Tool Usage 109

Tool Options 110

Load Path 111

Output Files 111
HDL Directory 111
Implementation Directory 112
Synthesis Directory 112

About Memory Generation 112
BMM Policy 113
BMM Flow 114

Reserved MHS Parameters 114

Synthesis Netlist Cache 115

Current Limitations 115

Chapter 6: Simulation Model Generator

Overview 117

Simulation Basics 118
Behavioral Simulation 118
Structural Simulation 118
Timing Simulation 118

Simulation Libraries 119
Xilinx Libraries 119
UNISIM Library 119
SIMPRIM Library 119
XilinxCoreLib Library 119
EDK Library 119

COMPEDKLIB Utility Tool 120
Usage 120
COMPEDKLIB Command Line Examples 120
Use Case I: Compiling HDL Sources in the Built-In Repositories in the EDK. 120
Use Case II: Compiling HDL Sources in Your Own Repository 120
Other Details 121
Changes for EDK 6.3 121

Simulation Models . . . 121
Behavioral Models . . . 121
Structural Models . . . 122
Timing Models . . . 122
Single and Mixed Language Models . . . 123
SimGen Syntax . . . 123
Requirements . . . 123
Options . . . 123
Help . . . 123
Version . . . 124
Options File . . . 124
HDL Language . . . 124
Log Output . . . 124
Library Directories . . . 124
Simulation Model Type . . . 124
Mixed Language . . . 124
Output Directory . . . 124
Target Part or Family . . . 125
Processor Elf Files . . . 125
Simulator . . . 125
Source Directory . . . 125
Top-Level Instance . . . 125
Top-Level Module . . . 125
Top-Level . . . 125
EDK Library Directory . . . 125
Xilinx Library Directory . . . 126
Output Files . . . 126
Memory Initialization . . . 126
VHDL . . . 126
Verilog . . . 127
Simulating Your Design . . . 127
Current Limitations . . . 127

Chapter 7: Library Generator

Overview . . . 129
Tool Usage . . . 129
Tool Options . . . 130
-**h**, -help (Help) . . . 130
-**v** (display version information) . . . 130
-**log** logfile[.log] . . . 130
-**p** part_name (architecture family) . . . 130
-**od** output_dir (specify output directory) . . . 130
-**sd** source_dir (specify source directory) . . . 130
-**lp** library_path (specify library path for user peripherals and drivers repositories) . . . 130
-**mhs** mhsfile.mhs (specify MHS file to be used) . . . 130
-**lib** . . . 131
Load Path . . . 131
Output Files . . . 133
include directory . . . 133
lib directory . . . 133
libsrc directory . . . 134

code directory 134
Libraries and Drivers Generation 134
Basic Philosophy 134
MDD/MLD and Tcl 134
MSS Parameters 135
Drivers 135
Libraries 135
OS 136
Interrupts and Interrupt Controller 136
Importance of Instantiation 136
Interrupt Controller Driver Customization 137
XMDSTUB Peripherals (MicroBlaze Specific) 137
STDIN and STDOUT Peripherals 137

Chapter 8: Platform Specification Utility

Tool Options 139
Overview of the MPD Creation Process 140
Detailed Use Models for Automatic MPD Creation 140
Peripherals with a Single Bus Interface 141
Signal Naming Conventions 141
Invoking PsfUtility 141
Peripherals with Multiple Bus Interfaces 141
Non-Exclusive Bus Interfaces 141
Exclusive Bus Interfaces 142
Peripherals with TRANSPARENT Bus Interfaces 142
BRAM PORTS 142
DRC Checks in PsfUtility 142
HDL Source Errors 142
Bus Interface Checks 142
HDL Peripheral Definitions 143
Bus Interface Naming Conventions 143
Naming Conventions for VHDL Generics 143
Reserved Parameters 145
C_BUS_CONFIG 145
C_FAMILY 145
C_INSTANCE 145
C_OPB_NUM_MASTERS 145
C_OPB_NUM_SLAVES 145
C_DCR_AWIDTH 145
C_DCR_DWIDTH 145
C_DCR_NUM_SLAVES 146
C_LMB_AWIDTH 146
C_LMB_DWIDTH 146
C_LMB_NUM_SLAVES 146
C_OPB_AWIDTH 146
C_OPB_DWIDTH 146
C_OPB_NUM_MASTERS 146
C_OPB_NUM_SLAVES 146
C_PLB_AWIDTH 146
C_PLB_DWIDTH 146

C_PLB_MID_WIDTH 146
C_PLB_NUM_MASTERS 147
C_PLB_NUM_SLAVES. 147
Signal Naming Conventions 147
Global Ports. 147
LMB - Clock and Reset 148
OPB - Clock and Reset 148
PLB - Clock and Reset. 148
Slave DCR Ports 148
DCR Slave Outputs. 148
DCR Slave Inputs 148
Slave LMB Ports 149
LMB Slave Outputs. 149
LMB Slave Inputs 149
Master OPB Ports 149
OPB Master Outputs. 150
OPB Master Inputs 150
Slave OPB Ports 150
OPB Slave Outputs 151
OPB Slave Inputs 151
Master/Slave OPB Ports 151
OPB Master/Slave Outputs 152
OPB Master/Slave Inputs. 152
Master PLB Ports 153
PLB Master Outputs 153
PLB Master Inputs 153
Slave PLB Ports. 154
PLB Slave Outputs 154
PLB Slave Inputs. 154

Chapter 9: Format Revision Tool

Revup to EDK 6.3 157
Tool Usage. 157
Limitations 158
Revup from EDK 3.2 to EDK 6.1 158
Tool Usage. 158
Limitations 158

Chapter 10: Bitstream Initializer

Overview 159
Tool Usage. 159
Tool Options. 159

Chapter 11: Programming Flash Memory

Overview 161
Prerequisites. 161
Supported Flash Hardware 161
Using the Program Flash Memory Dialog. 162
File to Program 162

Download Mode 163
Processor Instance 163
Flash Memory Properties 163
Instance Name 163
Program At Offset. 163
Scratch Pad Memory Properties 163
Instance Name 163
Program Flash. 163
Customizing Flash Programming 163
Using Flash Memory 164
Sample Bootloader 165

Chapter 12: GNU Compiler Tools

GNU Compiler Framework. 167
Compiler Usage and Options 168
Usage 168
Quick Reference 168
Compiler Options. 169
-g 169
-gstabs. 169
-O*n* 170
-v. 170
-save-temps. 170
-o Filename 170
-Wp,option 170
-Wa,option 170
-Wl,option. 170
--help. 171
Library Search Options 171
Header File Search Option 171
Linker Options 172
-defsym _STACK_SIZE=value 172
Linker Scripts 172
Search Paths 172
On Solaris 172
On Windows Xygwin Shell. 173
File Extensions. 173
File Types and Extensions 173
Libraries. 174
Compiler Interface 174
Input Files 174
Output Files 174
MicroBlaze GNU Compiler. 175
Quick Reference 175
MicroBlaze Compiler. 175
-mxl-soft-mul 175
-mno-xl-soft-mul. 175
-mxl-soft-div 176
-mno-xl-soft-div 176
-mxl-stack-check 176
-mxl-barrel-shift 176

-mxl-gp-opt . . . 176
-xl-mode-executable . . . 176
-xl-mode-xmdstub . . . 176
-xl-mode-xilkernel . . . 177
MicroBlaze Assembler . . . 177
MicroBlaze Linker . . . 178
-defsym _TEXT_START_ADDR=value . . . 178
-relax . . . 178
-N . . . 179
Initialization Files . . . 179
crt0.o . . . 179
crt1.o . . . 179
crt4.o . . . 179
Command Line Arguments . . . 180
Interrupt Handlers . . . 180
_interrupt_handler attribute . . . 180
_save_volatiles attribute . . . 180
PowerPC GNU Compiler . . . 181
Compiler Options . . . 181
-mhard-float . . . 181
-msoft-float . . . 181
Linker Options . . . 181
-defsym _START_ADDR=value . . . 181
Initialization Files . . . 182

Chapter 13: GNU Debugger

Overview . . . 183
Tool Usage . . . 183
Tool Options . . . 183
Debug Flow using GDB . . . 184
MicroBlaze GDB Targets . . . 184
Remote Targets . . . 185
Simulator Target . . . 185
Hardware Target . . . 185
Virtual Platform Target . . . 185
Compiling for Debugging on MicroBlaze Targets . . . 185
PowerPC Targets . . . 186
Console Mode . . . 186
GDB Command Reference . . . 187

Chapter 14: Xilinx Microprocessor Debugger (XMD)

XMD Usage . . . 190
Options: . . . 190
XMD Command Reference . . . 191
Connect Command Options . . . 194
PowerPC Target . . . 195
PowerPC Hardware Connection . . . 195
PowerPC Target Requirements . . . 197
Example Debug Sessions . . . 198
PowerPC Simulator Target . . . 201

Running PowerPC ISS 201
Example Debug Session for PowerPC ISS Target 202
MicroBlaze Processor Target 203
Microblaze MDM Hardware Target 204
MicroBlaze MDM Target Requirements 205
Example Debug Sessions 208
MicroBlaze Stub Hardware Target 213
MicroBlaze Stub-JTAG Target Options 213
MicroBlaze Stub-Serial Target Options 214
Stub Target Requirements. 215
MicroBlaze Simulator Target 216
Simulator Target Requirements 216
MDM Peripheral Target 217
MDM Target Requirements 217
Virtual Platform Microblaze Target 217
XMD Internal Tcl Commands 217
Program Initialization 217
Register/Memory 218
Program Control 218
Program Trace/Profile 219
Miscellaneous Commands 220
XMD TCP Socket Interface 220
Sending Commands to XMD 220
Return Types. 220

Chapter 1

Embedded System Tools Architecture

This chapter describes the Embedded System Tools (EST) architecture and flows for the Xilinx embedded processors, PowerPC 405 and MicroBlaze. The chapter contains the following sections.

- "Tool Architecture Overview"
- "Tool Flows"
- "Some Useful Tools"
- "Verifying Tools Setup"

Tool Architecture Overview

Figure 1-1 depicts the embedded software tool architecture. Multiple tools based on a common framework allow the user to design the complete embedded system. System design consists of the creation of the hardware and software components of the embedded processor system, and optionally, a verification or simulation component as well. The hardware component consists of an automatically generated hardware platform that can be optionally extended to include other hardware functionality specified by the user. The software component of the design consists of the software platform generated by the tools, along with the user designed application software. The verification component consists of automatically generated simulation models targeted to a specific simulator, based on the hardware and software components.

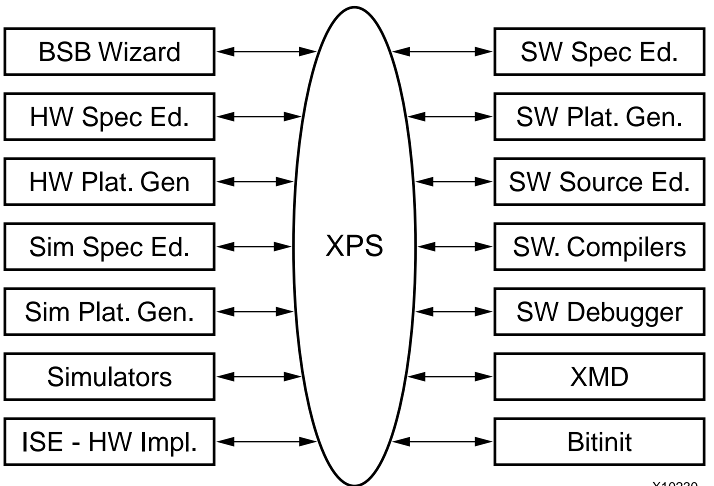


Figure 1-1: **Embedded Software Tool Architecture**

Tool Flows

A typical embedded system design project involves the following phases:

- hardware platform creation,
- hardware platform verification (simulation),
- software platform creation,
- software application creation, and
- software verification (debugging).

Xilinx provides tools to assist in all the above design phases. These tools play together with other, third-party tools such as simulators and text editors that may be used by the designers.

Hardware Platform Creation

Xilinx Platform Studio provides the Base System Builder Wizard for creating the hardware platform (see Chapter 3, “Base System Builder,” for more information about the wizard). Details of hardware platform creation are depicted in Figure 1-2.

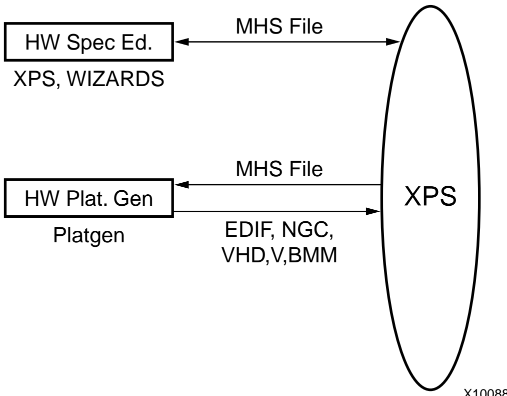


Figure 1-2: **Hardware Platform Creation**

The hardware platform is defined by the MHS (Microprocessor Hardware Specification) file (see Chapter 2, “Microprocessor Hardware Specification (MHS),” in the *Platform Specification Format Reference Manual* for more information). The hardware platform consists of one or more processors and peripherals connected to the processor buses. Several useful peripherals are usually supplied by Xilinx, along with the EDK tools. Users can define their own peripherals and include them in the MHS by following the guidelines in the *Platform Specification Format Reference Manual*. The MHS file is a simple text file and any text editor can be used to create this file. The XPS tool provides graphical means to create the MHS file.

The MHS file defines the system architecture, peripherals and embedded processors. The MHS file also defines the connectivity of the system, the address map of each peripheral in the system and configurable options for each peripheral. Multiple processor instances connected to one or more peripherals through one or more buses and bridges can also be specified in the MHS.

The Platform Generator tool (PlatGen) creates the hardware platform using the MHS file as input. PlatGen creates netlist files in various formats (NGC, EDIF), as well as support files

for downstream tools, and top level HDL wrappers to allow users to add other components to the automatically generated hardware platform. See Chapter 5, "Platform Generator," for more information.

Note: After running PlatGen, FPGA implementation tools (ISE) are run automatically to complete the implementation of the hardware. See ISE documentation for more info on the ISE tools. At the end of the ISE flow, a bitstream is generated to configure the FPGA. This bitstream includes initialization information for BRAM memories on the FPGA chip. If user code or data is required to be placed on these memories at startup time, the Bitinit tool is used to update the bitstream with code/data information obtained from the user's executable files, which are generated at the end of the "Software Application Creation and Verification" flow.

Verification Platform Creation

The verification platform is based on the hardware platform. The MHS file is processed by the Simgen tool to create simulation files (VHDL, Verilog or various compiled models) along with some command files for specific simulators supported by the tool. See Chapter 6, "Simulation Model Generator" for more information. As in the case of the hardware platform, these simulation files may be edited by the user to add other components to the automatically generated verification platform. The entire process of generating the verification platform is depicted in Figure 1-3. If the software application that runs on the hardware platform is available in executable format, it can be used to initialize memories in the verification platform. Details of this process are provided in later chapters.

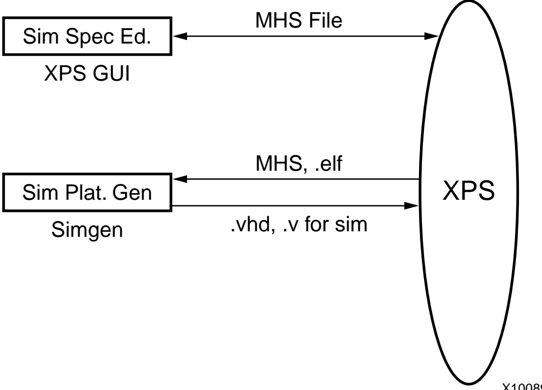


Figure 1-3: **Verification Platform**

Software Platform Creation

The software platform is defined by the MSS (Microprocessor Software Specification) file (see Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual* for more information). The MSS file defines driver and library customization parameters for peripherals, processor customization parameters, standard input/output devices, interrupt handler routines, and other related software features. The MSS file is a simple text file and any text editor can be used to create this file. The XPS tool (see Chapter 2, "Xilinx Platform Studio (XPS)" for more information) provides a graphical user interface for creating the MSS file.

The MSS file is an input to the Library Generator tool (LibGen) for customization of drivers, libraries and interrupt handlers. See Chapter 7, "Library Generator" for more information. The entire process of creating the software platform is shown in Figure 1-4.

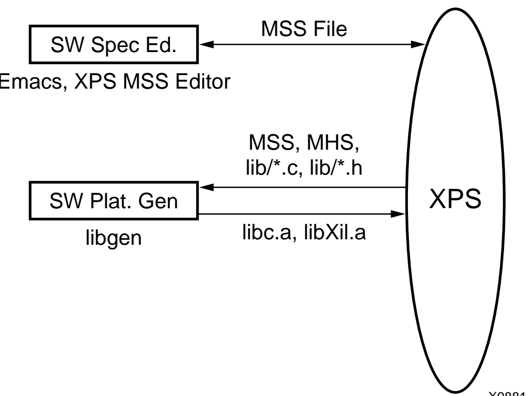


Figure 1-4: **Software Platform**

Software Application Creation and Verification

The software application is the code that runs on the hardware and software platforms. The source code for the application is written in a high level language such as C or C++, or in assembly language. XPS provides a source editor for creating these files, but any other text editor may be used here. Once the source files are created, they are compiled and linked to generate executable files in the ELF (Executable and Link Format) format. GNU compiler tools (see Chapter 12, "GNU Compiler Tools" for more information) for PowerPC and MicroBlaze are used by default but other compiler tools that support the specific processors used in the hardware platform may be used as well. XMD and the GNU debugger (GDB) are used together to debug the software application. XMD provides an instruction set simulator, and optionally connects to a working hardware platform to allow GDB to run the user application. This entire process is depicted in Figure 1-5. See Chapter 14, "Xilinx Microprocessor Debugger (XMD)" for more information on XMD and Chapter 13, "GNU Debugger" for more information on GDB. The Eclipse development environment is provided as an alternative to XPS for software application development.

Eclipse has its own built in source code editor and invokes the same compiler and debugger tools as XPS.

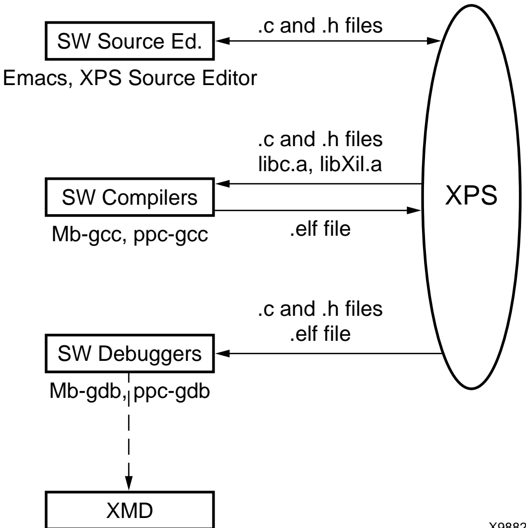


Figure 1-5: **Software Application Creation and Verification**

Some Useful Tools

Xilinx Platform Studio

The Xilinx Platform Studio (XPS) tool provides a GUI for creating the MHS and MSS files for the hardware and software flow. XPS also provides source file editor capability and project and process management capability. XPS is used for managing the complete tool flow, that is, both hardware and software implementation flows. See Chapter 2, "Xilinx Platform Studio (XPS)" for more information.

Base System Builder

The Base System Builder (BSB) wizard is a software tool that help users quickly build a working system targeted at a specific development board. BSB is invoked by XPS when the user wants to create a new system. See Chapter 3, "Base System Builder," for more information.

Create/Import IP Wizard

The Create/Import Peripheral Wizard helps you create your own peripherals and import them into EDK compliant repositories or Xilinx Platform Studio (XPS) projects. This wizard uses the PsfUtility tool to create the necessary Platform Specification files. See Chapter 4, "Create/Import Peripheral Wizard," for more information on the wizard, and see Chapter 8, "Platform Specification Utility," for more information of PsfUtility.

Platform Generator

The embedded processor system in the form of hardware netlists (HDL and EDIF files) is customized and generated by the Platform Generator (PlatGen).

See Chapter 5, "Platform Generator" for more information.

Simulation Model Generator

The Simulation Platform Generation tool (simgen) generates and configures various simulation models for the hardware. It takes a Microprocessor Hardware Specification (MHS) file as input.

See Chapter 6, "Simulation Model Generator" for details.

Library Generator

XPS calls the Library Generator tool for configuring the software flow.

The Library Generator (LibGen) tool configures libraries, device drivers, file systems and interrupt handlers for the embedded processor system. The input to LibGen is an MSS file.

Please see Chapter 7, "Library Generator" for more information. For more information on Libraries and Device Drivers please refer to Chapter 2, "Xilinx Microkernel (XMK)," in the *EDK OS and Libraries Reference Manual* and the "Device Driver Programmer Guide" chapter in the *Processor IP Reference Guide*.

Bitstream Initializer

The Bitstream Initializer tool initializes the instruction memory of processors on the FPGA. The instruction memory of processors are stored in BlockRAMs in the FPGA. This utility reads an MHS file, and invokes the Data2MEM utility provided in ISE to initialize the FPGA BlockRAMs. See Chapter 10, "Bitstream Initializer," for more information.

Format Revision Tool

The Format Revision Tool (revup) updates an existing EDK 6.1 or EDK 6.2 project to an EDK 6.3 project. Note that if you open a project from 6.1 or 6.2 in XPS 6.3, then it will automatically revup the project to the new release. See Chapter 9, "Format Revision Tool," for more information.

GNU Compiler Tools

XPS calls GNU compiler tools for compiling and linking application executables for each processor in the system.

Given a set of C source files, a Microprocessor executable is created as follows.

MicroBlaze

```
mb-gcc file1.c file2.c
```

This command compiles and links the files into an executable that can run on the MicroBlaze processor. The output executable is in **a.out**. The -**o** flag can be used to specify a different file name for the output file.

In order to initialize memories in the hardware bitstream with this executable, the file name should have an **elf** extension.

For further information on compiler options, **mb-gcc** -**help** can be run on the command line. See Chapter 12, "GNU Compiler Tools" for more information.

PowerPC

```
powerpc-eabi-gcc file1.c file2.c
```

This command compiles and links the files into an executable that can run on the PowerPC processor. The output executable is in **a.out**. The -**o** flag can be used to specify a different file name for the output file.

In order to initialize memories in the hardware bitstream with this executable, the file name should have an **elf** extension.

For further information on compiler options, **powerpc-eabi-gcc** --**help** can be run on the command line. See Chapter 12, "GNU Compiler Tools" for more information.

Compiling with Optimization

Once you are satisfied that your program is correct, recompile your program with optimization turned on. This will reduce the size of your executable, and reduce the number of cycles it needs to execute. This is achieved by the following:

```
mb-gcc -O3 file1.c file2.c
```

Setting the Stack Size

By default, the EDK tools build the executable with a default stack size of 0x100 (256) bytes.

The stack size can be set at compile time by using:

```
mb-gcc file1.c file2.c -Wl,defsym -Wl,_STACK_SIZE=0x400
```

This will set the stack size to 0x400 (1024) bytes.

Software Debugging

You can debug your program in software (using an instruction set simulator or virtual platform), or on a board which has a Xilinx FPGA loaded with your hardware bitstream. See Chapter 14, "Xilinx Microprocessor Debugger (XMD)," for more information.

Dumping an Object/Executable File

The mb-objdump utility lets you see the contents of an object (.o) or executable (.out) file.

To see your symbol table, the size of your file, and the names/sizes of the sections in the file, run the following:

```
mb-objdump -x a.out
```

To see a listing of the (assembly) code in your object or executable file, use

```
mb-objdump -d a.out
```

To get a list of other options, use the following command:

```
mb-objdump --help
```

Verifying Tools Setup

The environment variable **`XILINX_EDK`**, needs to be set at the level of the hierarchy where the directories **doc**, **hw**, and **bin** reside.

Tools Directory Path

Ensure that the GNU tools are in your path.

For Solaris or Linux

Check the executable search path. Your path must include the following:

- ${XILINX_EDK}/gnu/microblaze/sol/bin
- ${XILINX_EDK}/gnu/powerpc-eabi/sol/bin
- ${XILINX_EDK}/bin/sol

For PC

Check the executable search path.

- %XILINX_EDK%\gnu\microblaze\nt\bin
- %XILINX_EDK%\gnu\powerpc-eabi\nt\bin
- %XILINX_EDK%\bin\nt

Xilinx Alliance Software

The system should be set up to use the Xilinx Development System. Please verify that the system is properly configured. Consult release notes and installation notes included in the Xilinx ISE software package for more information. The EDK 6.3 release requires Xilinx ISE 6.3 Tools.

Chapter 2

Xilinx Platform Studio (XPS)

This chapter describes the Xilinx Platform Studio (XPS) IDE for the Xilinx Embedded Processors, MicroBlaze and PowerPC.

Xilinx Platform Studio (XPS) provides an integrated environment for creating the software and hardware specification flows for an Embedded Processor system. It also provides an editor and a project management interface to create and edit source code. XPS offers customization of tool flow configuration options. It also provides a graphical system editor for connection of processors, peripherals and buses. XPS is available on both Windows and Solaris platforms. There is also a batch mode invocation of XPS available.

This chapter contains the following sections.

- "Processes Supported"
- "Tools Supported"
- "Project Management"
- "XPS Interface"
- "Platform Management"
- "Software Application Management"
- "Flow Tool Settings and Required Files"
- "Tool Invocation"
- "Debug and Simulation"
- "PBD Editor"
- "XPS "No Window" Mode"

Processes Supported

XPS supports the creation of the MHS (refer to Chapter 2, "Microprocessor Hardware Specification (MHS)," in the *Platform Specification Format Reference Manual*) and MSS file, (refer to Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual*) files needed for embedded tools flow. The MVS file used in EDK 3.2 has been discontinued and that information is stored in XPS project files. XPS also aids users in creating an MHS (refer to Chapter 2, "Microprocessor Hardware Specification (MHS)," in the *Platform Specification Format Reference Manual*) through a dialog based editor and bus connection matrix, or through a graphical block diagram editor (referred to as the Platform Block Diagram editor). It supports customization of software libraries, drivers, interrupt handlers and compilation of user programs. Source management of C source files and header files for user applications is also provided by XPS. Users can also choose the simulation mode for the complete system. Users can begin a project by either importing an existing MHS file or by starting with an empty MHS file

and then adding cores to it. It performs process management and dependency checking between the hardware, software and simulation tool flows by calling the tools in the correct order using the makefile mechanism. Figure 2-1 provides a detailed view of processes supported by XPS.

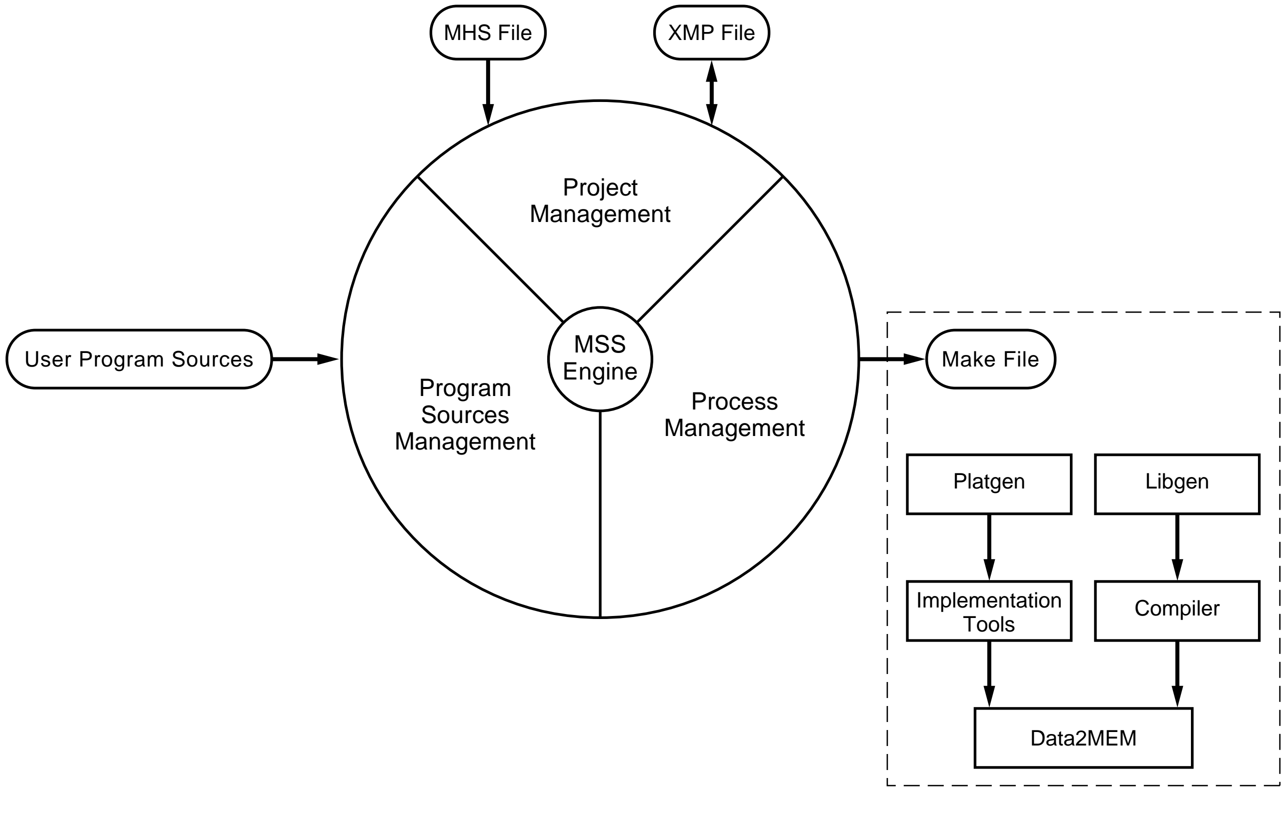


Figure 2-1: **XPS Process**

Tools Supported

Table 2-1 describes the tools that are supported in the XPS.

Table 2-1: **Tools supported in XPS**

Tool	Function	Reference/Notes
Library Generator (LibGen)	Customizes software libraries, drivers and interrupt handlers	The Library Generator Documentation
GNU Compiler Tools	Preprocess, compile, assemble and link programs	GNU tools Documentation
Platform Generator (PlatGen)	Allows user to customize various options. Runs platgen with the options and the MHS file	The Platform Generator Document
Simulation Model Generator (SimGen)	Generates the hardware simulation model and the compilation script file for the complete system.	The Simulation Model Generator
Makefile	Generates a Makefile, which provides targets to run various hardware and software flow tools.	Uses gmake on Unix platforms.

Table 2-1: **Tools supported in XPS**

Tool	Function	Reference/Notes
System ACE	Generates SystemACE file	Not supported on Solaris
XMD	Opens an XMD terminal for the user for on-board debug.	XMD Documentation
Project Navigator Export and Import	Export and Import design to Project Navigator for synthesis and implementation of design.	Flow is an alternative to the XFlow mechanism in XPS.

Features

XPS has the following features

- Adding cores, editing core parameters, and making bus and signal connections to generate a Microprocessor Hardware Specification (MHS)
- Generation and modification of the Microprocessor Software Specification (MSS)
- Support for all the tools described in Table 2-1.
- Graphical Block Diagram View and Editor.
- Multiple User Software Applications support
- Project management
- Process and tool flow dependency management

Project Management

Project information is saved in a Xilinx Microprocessor Project (XMP) file. An XMP file consists of the location of the MHS file, the MSS file, and the C source and header files that need to be compiled into an executable for a processor. The project also includes the FPGA architecture family and the device type for which the hardware tool flow needs to be run.

Creating a New Project

A new project is created using the **New Project** menu option in the **Project** submenu of the main menu. The **Base System Builder Wizard** in the **New Project** menu can be used to invoke the wizard to create a basic system. Please refer to Chapter 3, "Base System Builder" for more information. The **Platform Studio** option can be used to create a new project using XPS. The **New Project** toolbar button can also be used.

For creating a new project, users need to specify the location of the **xmp** file. The name of the xmp file is take to be the project name and the directory where the xmp file resides is considered to be the project directory. All tools are invoked from the project directory. All relative paths are assumed to be relative to the project directory. Optionally, users can also specify an MHS file to be used for the project if the project is created using Platform Studio. If the specified MHS file does not exist in the project directory or does not have same name as the project name, XPS copies it into the project directory with same base name as the project name. XPS always modifies the local copy of the MHS and never refers to the original MHS.

The target architecture *must* be set before running any tool. However, choosing the device size, the package and the speed grade can be deferred till implementation of the design. These options can also be set/changed later in the **Set Project Options** dialog box in **Options** → **Project Options** menu.

Users *must* specify all **Search Path** directories before loading the project **if**

- The MHS uses a peripheral which is not present either in the Xilinx EDK installation area or in **pcores** directory of the XPS project directory.
- The MSS uses a driver which is not present either in the Xilinx EDK installation area or in the **drivers** directory of the XPS project directory.

The concept of a Search Path directory, and its subdirectory structure is explained in detail in Platform Generator and Library Generator chapters. This corresponds to the -**lp option** of the tools. Please note that all the tools automatically look into the **pcores,** and **drivers** directories in the project directory and that the project directory itself should **not** be specified as the Search Path. Multiple directories can be specified as part of search path by specifying a semicolon (;) separated list of directories.

Opening an Existing Project

An existing XPS project can be opened by using the **Open Project** menu option (**File** menu) or using the Open Project button on the toolbar and specifying the existing XMP file corresponding to that project.

New source files and header files can be created, added, and deleted as described in the Source Code Management section of this chapter.

XPS does not allow multiple projects to be open simultaneously. Any open project must be closed before another project can be opened.

Getting Help

The main menu in XPS has a Help menu item. A link to the EDK documentation is provided in the **Help** submenu. The **EDK Examples** menu item is a link to the EDK examples web page at Xilinx. Many example designs are updated in this web site for users to download and use.

XPS Interface

Figure 2-2: **XPS (Xilinx Platform Studio)**

Figure 2-2 shows a screenshot of XPS. XPS opens three main windows by default.

Editor Workspace

The main editor workspace appears on the right in XPS in Figure 2-2. The workspace opens PBD (Platform Block Diagram) file and allows graphical editing of the system. The main workspace also functions as a C source and header file editor of XPS. Users can also view and edit other text files in the main window. Any number of text files can be opened simultaneously in the XPS main window. The PBD file can be opened by double clicking on the PBD file in the system tree view, or through the **Project** →**View Schematic** menu item.

The PBD editor is described in more detail later in this chapter (see “PBD Editor,” page 42).

System Tab

This tab is one of the four tabs that appear on the left in the XPS window in Figure 2-2. The system tab shows the system in a tree format. There are three sub-trees in this view:

- The **System BSP** tree shows system components (various cores) by their instance names. Each core can have its own sub-tree which displays information corresponding to that instance (for example base address and high address). Source and header files corresponding to a processor are listed in the sub-tree for that processor instance.
- The **Project Files** tree shows the MHS. MSS, PBD, UCF and other files corresponding to the project. Users can double-click on any of the file names to open it in the XPS main window. Some of these files must be created by the user in order to implement the design.
- The **Project Options** tree shows the current value set for various project options. Users can double-click or do a Right-click on any of the fields shown in this tree to bring up the **Set Project Options** dialog box.

Applications Tab

This tab shows all user software application projects. Users can create a number of software application projects that are associated with the processors in their design.

A software application project consists of a unique project name, a set of source and header files that the users can create to design their application. The source files can be built into executables (one executable per application project) that can be downloaded onto the FPGA.

If users have multiple applications, but the current design is only going to require a subset of those applications, they should mark the other applications as "Inactive". XPS engine will ignore all the "Inactive" applications. This enables users to preserve software applications and does not force them from deleting those applications.

Each active application project can be specified with a set of compiler options. A right click on the application projects tree view brings up a context menu. The menu items can be invoked to set compiler options, view files, open files, associate different processors with the project and so on. Each project can also be marked for initialize BRAMs. If a user application resides completely in BRAM memory and the user wants to download that ELF file as part of the bitstream, then those applications must be "Marked to initialize BRAMs". XPS will use data2mem to update the bitstream with those ELF files.

For every processor in the design, an application project called <*processor instance*>`_bootloop` is created by default. This is a predefined bootloop that can be downloaded to the BRAMs so that the processor is in a valid state on wakeup. A **View Source** on the bootloop project will open the source file with more comments explaining the importance of the bootloop. For more information please see the Software Application Management Section of this chapter.

Transcript Window (Output)

The transcript window is the bottom window in Figure 2-2. This window acts as a console for output, warning and error messages from XPS and from other tools invoked by XPS.

Platform Management

In order to change the system specification, software settings, and simulation options, XPS supports the following features and processes.

Add/Edit Cores (Dialog)

A **Right click** on **System BSP** item in the System View tab gives a menu option to **Add Cores (dialog)** to the system. Selecting it brings up a tabbed dialog box that lists all the cores which can be instantiated in the design. Multiple cores can be selected at a time for adding to the design by using the 'Shift' or 'Ctrl' key. The tabs can be used to add and connect buses, connect BRAMs to BRAM controllers, add ports and connect using net names and set parameters on cores. Please refer to the MPD and MHS document for parameter information. Also the IP documentation includes parameters that can be changed for each IP.

Simulation Models

A **Right click** on **System BSP** item in the System View tab gives a menu option to set the **Simulation Model** for the system. User can choose between **Behavioral**, **Structural**, and **Timing** modes of simulation. The currently selected model has a check mark against it. This information is stored in XMP file.

View MPD

Right click on an instance name give users the option to **View MPD** for that core. If selected, the MPD file for that core is opened in the main window. If the MPD file is already open, focus is set on the file. MPD files are opened in read-only mode and can not be edited.

View MDD

Right click on an instance name gives users the option to **View MDD** for driver assigned to that core instance. This option is disabled if no driver is assigned to that core. If selected, the MDD file for that core's driver is opened in the main window. If the MDD file is already open, focus is set on the file. MDD files are opened in read-only mode and can not be edited.

S/W Settings

In the System BSP tree, a **double click** on an instance name opens a dialog window displaying configurable software platform options for all peripherals. This window can also be brought up by doing a Right click on peripheral instance name and choosing the menu item **S/W Settings**. This dialog has multiple tabs and is used to set all the software platform related options in the design. The tabs and their significance are detailed as follows:

Software Platform

This tab shows three tables: **Drivers**, **Libraries** and **Kernel and Operating Systems**.

The **Drivers** table displays peripherals used in the design and users can assign drivers for these peripherals. Drivers may already be assigned by default, and users have the ability to change the default drivers.

The **Libraries** table shows all the libraries that are included in the EDK and each library can be included in the design by checking the **Use** column.

The **Kernel and OS** table can be used to select an OS for the processor system in the design. A **standalone** OS is selected by default.

Please see the Microprocessor Software Specification (MSS) for more information.

Processor and Driver Parameters

This tab shows two tables, **Processor Parameters** and **Driver Parameters**. These tables can be used to specify values for the parameters associated with the processors or peripheral drivers in the design. The driver table also displays interrupt handler parameter if the peripheral using the driver is connected to an interrupt port. The name of the interrupt handling routine can be specified for any peripheral interrupt signal. If the peripheral has no interrupt port, or if those interrupt port(s) are not connected to any signal in the MHS file, then this parameter does not show up. Please see the Microprocessor Driver Definition (MDD) chapter for more information.

Library and O/S Parameters

This tab shows a list of all configurable library and Kernel/OS parameters for all the libraries and OS in the design. Please see the Microprocessor Library Definition (MLD) and the Libraries guide for more information.

Software Application Management

MSS file specifies the software platform for the embedded system design. This includes the OS, drivers for IPs and other libraries. Multiple applications can be run on a software platform. XPS allows users to specify multiple application projects. This is specified in the **Applications** tab. Each application is associated with a processor instance that executes the application. Users must specify a unique name for each application project. An application project has a list of C source and header files associated with it. Users can also specify compiler options for each application. All the source files for a processor are compiled using the compiler specified for that processor in the SW platform settings for that processor. XPS has an integrated editor for viewing and editing C source and header files of the user program.

Adding Files

Files can be added to a active software application by clicking the right mouse button on the Sources or Headers item in the application project. The same operation can be accomplished by using the **Project** → **Add Program Sources** menu item in the Main menu. Multiple files are added by pressing the control key and using arrow keys (or the mouse) to select in the file selection dialog. XPS adds files to Sources or Headers subtree depending upon the file extension. All directories where the header files are present are automatically added to the Include Search Path compiler option.

Deleting Files from Project

Any file can be deleted from a software application by selecting the file in the Project View window then clicking the right mouse button on the item and choosing **Delete File**. Note that the file does not get physically deleted from the disk. It is just removed from the list of files to be compiled to generate the executable for that application.

Editing Files

Double clicking on the source or header file in the Project View window opens the file for editing. The editor supports basic editing functions such as cut, paste, copy and search/replace. The editor highlights basic source code syntax. It also supports file management and printing functions such as saving, printing, and print previews.

Mark Application for Downloading to BRAMs

Active Software application ELF files which reside on FPGA's BRAM memory need to marked for downloading into BRAMs. This can be done by right clicking on the software application and selecting "Mark for Download" menu item. Similarly, you can also deselect the application for downloading to BRAMs. If an application is marked for BRAMs, XPS passes these applications to the data2mem utility which initializes the bitstream with BRAM information from the ELF files. XPS also passes these ELF files to simgen to create appropriately initialized simulation models. By default, a software application is assumed to be using BRAMs. Note that by marking an application for download to BRAMs, no process gets invoked, but rather a flag is set up to indicate that the application has to be downloaded at the proper step in the flow.

Application to be Compiled Outside the XPS Environment

Sometimes, users want to compile their application outside the XPS environment (e.g. in VxWorks, Eclipse etc.), but they might want XPS to be aware of the ELF file. In such cases, they should create an application project and specify the ELF file which they will be creating outside XPS. However, users should not add any C-source files associated with it. This indicates to XPS that user has an associated ELF file, but does not want to compile it within XPS. Any changes that might require user to recompile his application (e.g. MHS/MSS file change) must be managed by the user himself.

Bootloop Software Applications

For each processor, XPS adds an special bootloop software application. These applications have a precompiled ELF associated with them. The pre-compiled ELF and the source file, linker script and the make file used to compile that ELF can be found in the EDK installation directory. These applications are displayed at the top of the Software Applications tree. Users can not modify sources and compiler options for these applications. Users can only select to either download this application into BRAMs or not.

The bootloop application ELF files is a simple single-instruction application. The instruction branches to itself thus creating an infinite loop. This is useful in cases where the processor has started execution but the actual application has not been downloaded to external memory. The bootloop prevents the processor from executing arbitrary instructions. This application resides at the start address location of the processor. For microblaze, the start address is 0x00000000, while for ppc405, it is 0xFFFFFFFC.

Xmdstub Software Applications

For every microblaze processor in design, an application called <*processor_instance*>`_xmdstub` is created by XPS. The ELF file associated with this processor is created as part of the library generation at the location of <*proc_instance*>`/code/xmdstub.elf`. Users can decide whether to download this application or not. Typically, if any of the active user applications is in XMDSTUB mode, then users would want to download xmdstub.elf for that processor onto BRAM memory.

Compiler Options

A Compiler Option Dialog Window opens up when any active software application name is double-clicked or the **Set Compiler Option** menu option is chosen for that software application in the Software Projects tree in Applications tab. This dialog has the following four tabs.

Environment

The tab displays the compiler being used for compiling this application. The compiler used can be changed in the "Software PlatForm Settings" dialog. For a microblaze application, users can specify what mode the application should be compiled into, XMDSTUB or EXECUTABLE.

This tab gives you the ability to provide **Program Start Address**, **Stack Size**, and **Heap Size** for the gcc-based compilers (mb-gcc and powerpc-eabi-gcc)**.** Please note that these options should **not be used with dcc** (they should be specified in the linker script for dcc). Heap size is only for PowerPC instance.

Optimization

This tab allows you to specify various compiler options. The degree of optimization can be specified to be 1,2, or 3. User can specify whether to perform Global pointer optimizations. Also, if they included the xilprofile library in the "Software PlatForm Settings" dialog, then can also choose whether to enable profiling for this application or not.

Users can also choose the debug options, whether the code should be generated without debug symbol, or with symbols for debugging (-g) or with symbols for assembly (-gstabs).

Directories

This tab allows you to specify various search directories for the **Compiler** (-B), for **Libraries** (-L) and for **Include** (-I) files. You can specify what user libraries, if any, should be used by the linker in the **Libs to Link** (-l) field. The libxil.a library is automatically picked up by gcc- based compilers. For dcc, XPS automatically adds libxil.a as a library to link in the makefile compiler options. You can also specify any **Linker script** (some times called map file) to be used. Again, the gcc based compilers pick up the default linker script from the EDK installation area if this option is not specified. You can also specify the name of the **Output ELF file** to be generated by the compiler. If these paths are not absolute, they must be relative to the project directory.

Advanced

The user can also specify various options which the compiler should pass to the **Preprocessor** (-Wp), the **Assembler** (-Wa), and the **Linker** (-Wl). Each option is dealt in detail in the GNU Compiler Tools documentation. You do not need to type in the specific flags as XPS introduces the correct flag for each option automatically. However, if you type the flags, then XPS does not introduce them. If there are more than one option in a field, they should be separated by space.

For compiling program sources, if you want to specify any Compiler Options in addition to those specified in other tabs, you can specify them in the **Program Sources Compiler Options** edit box.

Table 2-2 shows the options that are displayed in the compiler options dialog window under various tabs.

Table 2-2: **Processor Options**

Option	Value Type	Description
Compiler Options	Optimization Level	Choose the level of compiler optimization. Equivalent to -O option in gcc.
Global Pointer Optimization	Compiler Option	This option enables global pointer optimization in the compiler. This option is only for MicroBlaze.
Debug	Compiler Option	**-g** option to generate debug symbols.
Search Paths	Directories	Compiler, Library and Include paths. Equivalent to -B, -L and -I option to gcc.
Libraries to Link	Linker Option	The libraries to link against while building the ELF file (-l option)
Output File	File path and name	Sets the name of the executable file. Equivalent to -o option of gcc.
Program Start Address	Hex Value	Specifies the start address of the text segment of the executable for MicroBlaze and the program start address for PPC.
Stack Size	Hex Value	Specifies the stack size in bytes for the program.
Heap Size	Hex Value	Specifies the heap size in bytes for the program. Heap size can only be specified for a PPC Instance.
Pass Options	Compiler Options	Options can also be passed to the compiler, assembler and linker. The options have to be space separated.

For more information on the options, please refer to Chapter 12, “GNU Compiler Tools”.

Generating Linker Scripts

A Generate Linker Script Window opens up when **Generate Linker Script** menu option is chosen for that software application in the Software Projects tree in Applications tab. This dialog has the following configuration information.

Sections View

The sections view displays the list of sections associated with the software application. If an elf file is present for the software application, then the section view is populated from the elf file settings. Each section has the size and memory assignment information. If the elf file is not present, then the default sections for the processor that is associated with the software application is listed. Each of the section listed can be mapped to any specific memory.

Memory View

The memory view displays the list of all memories associated with the processor instance of the software application. Memory information includes instance name used in the MHS file, start address of the memory and the size of the memory. The view is read-only and hence none of the fields are editable.

Heap and Stack View

The heap and stack view displays the heap and stack information associated with the software application. If an elf file is present for the software application, then the heap and stack view is populated from the elf file settings. If the elf file is not present, then the default value for each of stack and heap are assigned. Each of stack and heap can be configured for size and memory assignment.

Elf File Information

The Elf file used in pre-populating the sections view, and the Heap and Stack View is shown here. The Elf file is the executable associated with the software application

Output Linker Script

The Output Linker script file name is the name of the file specified for linker script in the **Set Compiler Option menu.** If a valid linker script file exists, then this file is copied in as <*original_linker_script_file*`.bak`> before generating the linker script in the file <*original_linker_script_file*>.

Add/Delete Sections

Apart from the list of the sections listed in the Sections View, new user sections can be added. Selecting the **Add Section** button creates an extra row in the Sections View. The name of the section and the memory assignment can be configured. To delete a newly added section, select the corresponding row and click the **Delete Section** Button. This will remove the newly added section. Note that the default sections retrieved either from the Elf file or assigned based on the target processor instance cannot be deleted.

Generate

Once all the settings are configured, click **Generate** to generate the linker script. If there are any errors in the settings, relevant error messages are displayed in the transcript console at the bottom of XPS. Note that once a valid linker script is generated, the software application needs to be built in order for the settings to be preserved. Generated elf file is used to retrieve the settings on the next run of Linker Script generator.

Flow Tool Settings and Required Files

XPS supports tool flows as shown in Table 2-1. The Main menu has an **Options** submenu. You can set various project and tool options, as described below for each menu item.

Compiler Options

This menu opens the same dialog box as one opened by double-clicking on a software application name. If there is a single application in user's system, it will automatically open the dialog box corresponding to the application, otherwise, user will be asked which software application they want the options to be set for. User can set various compiler options in the processor dialog box which opens, as explained earlier in Processor Dialog Box section.

Project Options

Menu item **Options** → **Project Options** opens a dialog box which allows user to specify various project options. The same dialog can be brought up by clicking on the Project

Options button in the toolbar or by double-clicking on any item in the Project Options tree in the Project View window. There are three tabs in this dialog box.

Device and Repository

The target device for the project can be changed here. There are four different items: **Architecture**, **Device Size**, **Package**, and **Speed Grade**.

Users can specify the **Search Path** directories here. However, if this option is changed, users must close the project immediately. If this option is changed here, the changes will be effective only if the project is closed and loaded again.This option corresponds to the **-lp option** of various batch tools. See Chapter 7, "Library Generator" and Chapter 5, "Platform Generator" for more information.

Users can also specify their **own Makefile** to be used in XPS.Before EDK 6.2, XPS used to generate only 1 makefile, namely <*projname*>`.make`. The XPS makefile is split into two parts

- The main makefile: <*projname*>`.make`
- The include makefile: <*projname*>`_incl.make`.

The <*projname*>`_incl.make` file contains all options and settings defined in form of macros. The main makefile <*projname*>`.make` contains all the targets and commands for the complete flow. The main makefile includes the <*projname*>`_incl.make` using the following make directive:-

```
include system_incl.make
```

This makes all the macros defined in <*projname*>_incl.make visible in <*projname*>.make. XPS always writes out both the makefiles. However, users can choose not to use the <*projname*>.make file for their flow. Instead, they can specify their own makefile. Note that user makefile specified must be different from the two makefiles generated by XPS. Users are expected to include the <*projname*>_incl.make in their own makefile too. This way, any changes they make to any options and settings in XPS will be reflected in their own makefile too. Typically, a user would generate the <*projname*>.make file once and then copy it and modify it for their own purposes.

Note that you will need to update your makefile whenever you make a significant change in your design. Some of the changes which affect makefile structure are:-

- Adding, deleting, or renaming a processor
- Adding, deleting, or renaming a software application
- If you change the choice of implementation tool between ISE (ProjNav) and XPS (Xflow).
- The ACE file generation command might be changed if you change the number of processors in your design or if you add/delete opb_mdm ip for microblaze designs.
- The XILINX_EDK_DIR macro defined in system_incl.make file changes across Unix (Solaris/Linux) and Windows platforms.

Hierarchy and Flow

This tab allows user to specify the design hierarchy, whether the processor design being done in XPS is the top level module or if it is just a sub-module in the entire hierarchy. If this design is a sub-module, the Top Instance edit box allows you to specify the instance name used to instantiate this module in the top-level design. This corresponds to the **-iobuf** and **-ti** options of PlatGen tool.

From EDK 6.1 onwards, XPS only supports modular (hierarchical) design mode. The Flat mode is not supported.User can also choose whether to run the Xilinx Synthesis Tool (XST).

Users can also specify the flow to use for running the Xilinx implementation tools. The available options are XPS (Xflow) and ISE (Project Navigator) flow. Note that if the design is a sub-module, users must use the ISE flow. Please see the "ISE Project Navigator Interface" section described later for details on how to add design components and files to ProjNav project using XPS.

HDL and Simulation

This tab allows the user to specify the HDL (VHDL or Verilog) to be used by PlatGen and SimGen. Users can also specify the location of various simulation libraries. For details on simulation libraries, please refer to SimGen tool.Users can specify the simulation tool of their choice. Currently, EDK supports ModelSim and NCsim. Users can also specify the current simulation mode they want to use. These options are saved into the XMP file.

Required Files

If XPS (Xflow) is chosen to run the implementation tools, XPS expects a certain directory structure in the project directory. For each project, the user must provide User Constraints File (UCF). The file should reside in the `data` directory in the project directory and should have the name <*mhs_name*>`.ucf`. Users are also expected to provide an **iMPACT** script file. This file should reside in the `etc` directory and should be called `download.cmd`. If these files do not exist, XPS will prompt the user to provide these files and will not run XFlow.To run Xilinx Implementation tools, XPS uses two more files, `bitgen.ut` and `fast_runtime.opt` from `etc` directory. However, if the two files are not present, XPS copies the default version of these two files into that directory from the EDK installation directory. To change options for Xilinx implementation tools, the user can modify the two files. Note that when a new project is created, if the data and etc directories do not exists, XPS creates these empty directories in the project directory.

Tool Invocation

After all options for the compiler and library generator are set, the tools can be invoked from the **Run** submenu in the Main menu. The main toolbar also contains buttons to invoke these tools.

There are two different flows in the EDK platform building flow, the hardware flow and the software flow.

Software Flow

The software flow involves building up the software part of the embedded system. There are two important steps:

1. **Generate Libraries:** This button invokes the library building tool LibGen with the correct MSS file as input.
2. **Compile Program Sources:** This button invokes the compiler for each software application which needs to be compiled with in XPS. with corresponding program sources. It builds the executable files for each processor. If LibGen has not been executed, this button first invokes LibGen.

Hardware Flow

The hardware flow involves building up the hardware part of the embedded system. There are two important steps:

1. **Generate Netlist:** This button calls the platform building tool PlatGen with the correct MHS file and produces the netlist files in NGC format.
2. **Generate Bitstream:** If using XPS for implementation tools, this button calls the tool xflow with the fast_runtime.opt and bitgen.ut files residing in the etc. directory in the project directory. XFlow in turn calls the Xilinx ISE Implementation tools. If using ProjNav for the implementation flow, the button is greyed out. User must use **Tools** → **Export to ProjNav** menu to add the XPS files into ProjNav project, run the complete flow in ProjNav and then use **Tools** → **Import** from ProjNav menu to import bitstream and bmm files back into the flow.

Merging Hardware and Software Flows and Downloading

1. **Update Bitstream:** This button invokes the tool bitinit. This is the stage where the hardware and the software flows come together. This button also calls hardware and software flow tools if required. At the end of this stage, users get a `download.bit` file which contains information regarding both the software and the hardware part of the design.
2. **Generate SystemACE File:** This menu item generates a SystemACE file. This option is available only when you have single processor in your system. This option is available only on windows and linux platform in this release. Note that there is no toolbar button for this option.
3. **Download Bitstream:** This button downloads the download.bit file onto the target board using the Xilinx iMPACT tool in batch mode. XPS uses the file etc/download.cmd for downloading the bitstream.

XPS generates a makefile in the project directory and calls the corresponding target. The dependencies between various tools being run is take care of by the Makefile.

When LibGen is invoked, an MSS file is created for the software specification. When the user exits the application, a prompt to save the current project appears.

ISE Project Navigator Interface

If ISE (ProjNav) is chosen for implementation flow in the Project Options dialog box, then user must specify the ProjNav project (NPL) file. ProjNav will run implementation tools in the directory where this ProjNav project file is created. Default NPL file location is `<proj_dir>/projnav/<proj_name>.npl`. It is recommended not to use the `implementation` directory for ProjNav flow since XPS clean mechanism deletes this directory. To run the ProjNav flow, user can create a new ProjNav project file or specify an already existing ProjNav project file.

Menu option **Tools** → **Export ProjNav Project** adds the required vhdl and bmm files to the ProjNav project. It also sets the ProjNav option **Macro Search Path** to <*proj_dir*>`/implementation` so that implementation tools can locate ngc files generated by PlatGen or XST.

Menu option **Tools** → **Import ProjNav Project** gives user the option to import a bitstream and a bmm file back into the XPS Project. The bit file should be the one generated by bitgen at the end of implementation tools. The bmm file should also be the one generated by bitgen, which has BRAM placement information. XPS copies the bit and bmm files into the

implementation directory as <*mhsbasename*>`.bit` and <*mhsbasename*>`_bd.bmm` respectively.

Debug and Simulation

Users can debug the hardware and the software part of the design either by simulation or by running it on the hardware itself. XPS provides support for invoking the corresponding tools to perform the job.

- **Xilinx Microprocessor Debug (XMD)**: Invoke the XMD tool to debug the application software. The XMD-button on the XPS toolbar opens up a XMD shell in the project directory.
- **Software Debugger:** The debug button invokes the software debugger corresponding to the compiler being used for the processor. If there are more than one processor in the design, XPS prompts to choose the processor whose program sources the user wants to debug.
- **Hardware Simulation Model Generator (SimGen):** Invoke the SimGen tool to generate various simulation models for the components instantiated in MHS File. Depending on the simulation model to be used (Behavioral, Structural or Timing), XPS calls SimGen with appropriate options to generate the simulation models and initialize memory. Then XPS compiles those models for ModelTech's ModelSim simulator and starts the simulator with the compiled files.

PBD Editor

The Processor Block Diagram Editor (PBD Editor) allows you to read, create, modify and save a description of an FPGA Platform that references Hardware (HW) components. The HW components comprise, in part, microprocessors, buses and bus arbiters, and peripheral devices.

The PBD Editor block diagram supplies the hardware platform information written into the MHS file.

PBD Editor Interface

The PBD Editor interface is shown in Figure 2-4. These areas comprise the interface:

- The workspace
- The system tabs

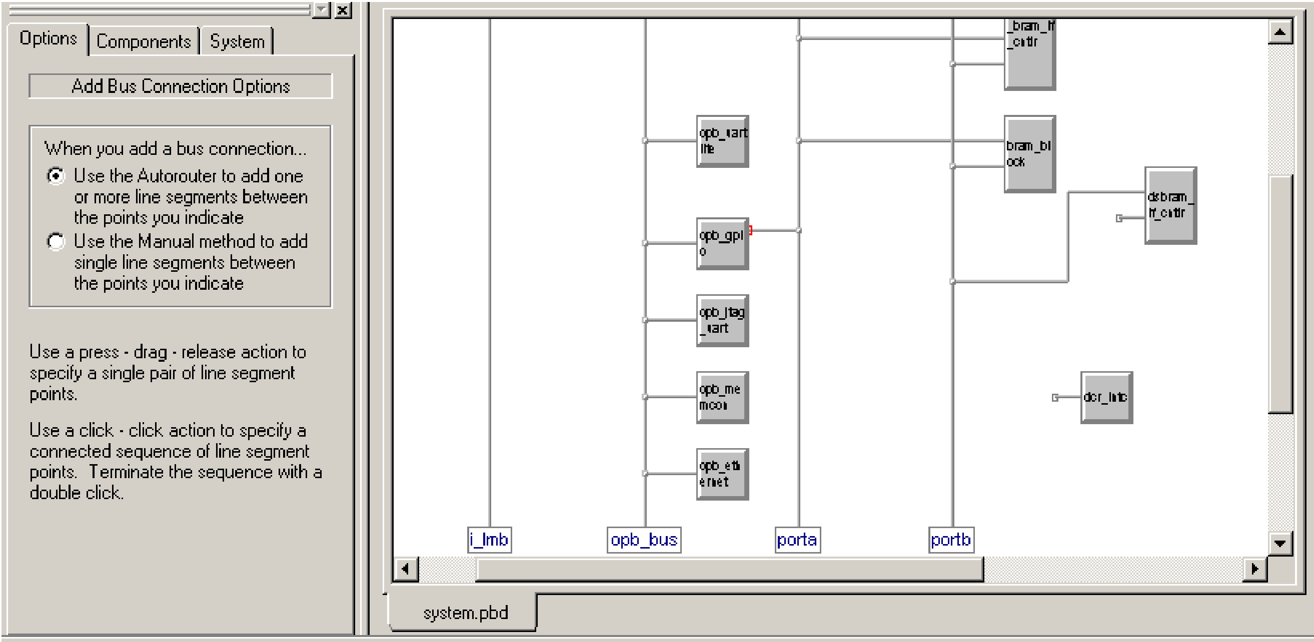


Figure 2-3: **The PBD Editor**

PBD Editor Workspace

The PBD Editor workspace is the upper right window in the XPS (see Figure 2-4). The workspace contains the block diagram describing the system hardware.

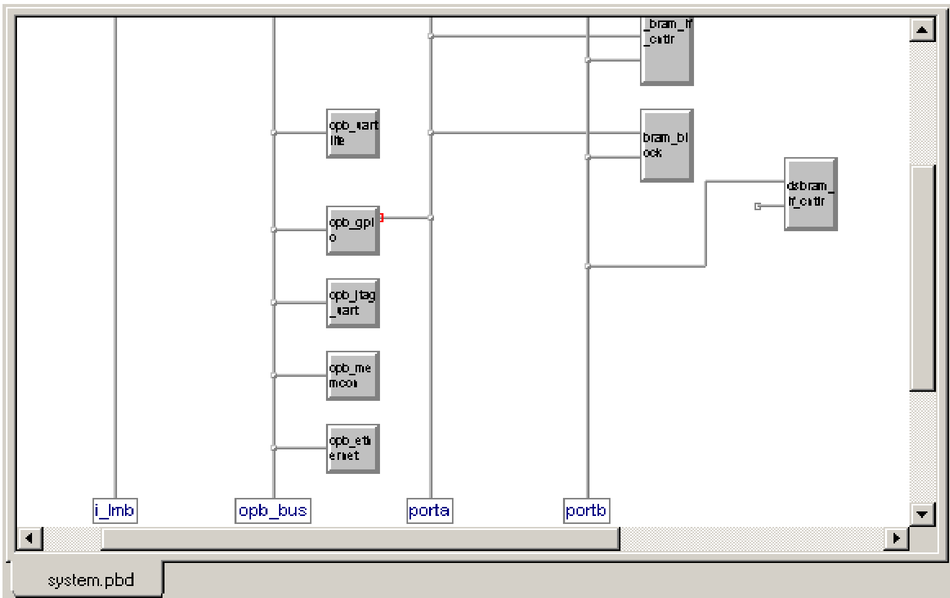


Figure 2-4: **PBD Editor Workspace**

System Tabs

The system tabs are in the upper left of the XPS window (see Figure 2-5). Two of the tabs in the window are used in the PBD Editor operation.

- The **Options** tab changes according to the tool that you are using and allows you to set options related to the tool, such as how the Add Bus Connection tool should operate.
- The **Components** tab allows you to select a component (a CPU, Bus Infrastructure component, or peripheral) to instantiate into your system. The components are Xilinx cores.

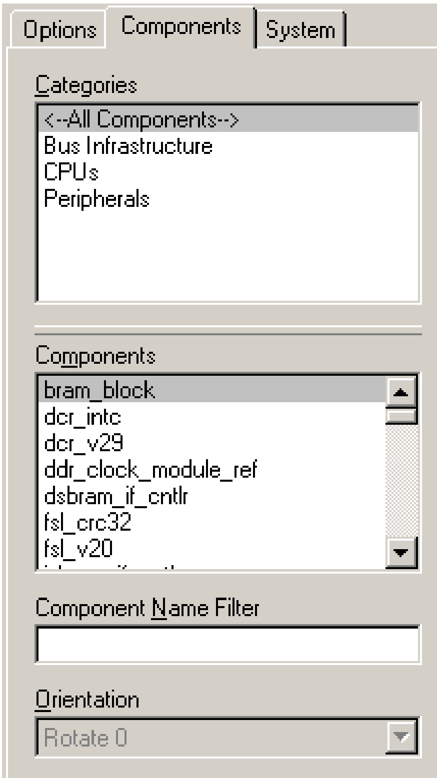


Figure 2-5: **System Tabs**

Creating the Hardware Block Diagram

The following procedures are used to create the hardware platform in the PBD Editor.

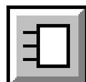
Adding a Component Instance to the System

Component instances are Xilinx cores (IP) instantiated in the hardware design. The components you add to the system may be:

- CPUs
- Bus components
- Peripherals

To add a component instance to the system:

1. Select the ***project_name*.pbd** tab in the workspace to display the system block diagram.
2. Select **Add** → **Component** or click the Add Component toolbar button.



3. In the **Components** tab, use the **Categories** and **Components** lists to specify the component you are adding.

   The component you select is attached to the mouse cursor.

   Note: To make the component selection easier, type the first letter or letters of the component in the **Component Name Filter** field. The **Components** list box shows only the components that begin with those letters. A regular expression can also be used to filter components. For example, typing ".*uart" will list all components with "uart" in the name. A "." stands for a character and "*" means "zero or more".
4. Click where you want the component instance to appear in the workspace.

Component instance notes:

- The PBD Editor assigns the new component instance the default name *corename_number*. The *number* is incremented each time another instance is added.
- To rename a component instance, see "Naming an Instance".
- If a bus pin on the component symbol touches a bus, and if the pin is compatible with the bus type, the symbol pin is connected to the bus when the component instance is placed in the block diagram.

Naming an Instance

When you add a component to the system, the PBD Editor assigns the new component instance the default name *corename_number*, and the *number* is incremented each time another instance is added. You can leave the machine-generated names as is. However, it is usually easier to debug the design using your own names.

To rename an instance.

1. Double-click the instance in the workspace.
2. In the Object Properties dialog box, change the **Instance Name**.

Setting Component Instance Parameters

You set parameters to customize the instantiated IP for your design. Parameters may be set for CPUs, bus components, or peripherals. The properties you set depend on the type of component and the IP (core) from which the component was instantiated.

IP parameters are described in the data sheets for the cores instantiated in the design. Data sheets can be accessed from the Xilinx IP Center page at http://www.xilinx.com/ipcenter.

To set parameters for a customizable component instance:

1. Double-click the component instance in the workspace.
2. In the Properties dialog box, click the **Parameters** entry in the tree view on the left side of the dialog box.
3. To override a value displayed in the **Default Parameter Values** table:
   a. Select the parameter in the **Default Parameter Values** table.

b. Clicking **Add**.

c. Change the parameter **Value** in the **Explicit Parameter Values** table.

d. Click **Apply**.

The value entered in the **Explicit Parameter Values** table overrides the value displayed in the **Default Parameter Values** table.

Setting Symbol Properties

Symbol properties determine the appearance of an instance's block in the workspace. You can modify the size of the symbol drawing or the location of the bus pins on the symbol.

Some components (the MicroBlaze processor, for example) have a large number of bus interfaces, only a few of which may be used in the block diagram. You can hide the bus interface pins that are not in use, thus reducing the size of the symbol and making the diagram easier to read.

To set symbol properties:

1. Double-click component instance in the workspace.
2. In the Properties dialog box, click the **Symbol** entry in the tree view on the left side of the dialog box.
3. To change the size of the symbol:
   a. Enter a value in the **Min Width** and/or **Min Height** fields.
   b. Click **Add**.
4. To change the orientation (top, bottom, left, or right) of a symbol pin:
   a. Select the pin in the **Available Pins** table.
   b. Click **Add**.
   c. At the top of the **Pins on Symbol** area, select the orientation you want (**Top**, **Bottom**, **Left**, or **Right**).
   d. Click **Apply**.

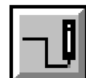
The symbol in the workspace is updated to reflect the change.

Connecting a Component Bus Pin to a Bus

When you connect a component bus pin to a compatible bus, connection lines are drawn from the pin to show the bus connection. All of the signals represented by the bus pin are connected to the bus.

To connect a component bus pin to a bus:

1. Select **Add** → **Bus Connection** or click the Add Bus Connection toolbar button.



2. Select the bus pin on the component instance you wish to connect to the bus.

   To select the pin, move the cursor near the end of the pin until four squares appear to help you locate the exact point. When the cursor is in the correct position to select the pin, a box appears with information about the component instance and the type of pin you are selecting.
3. Click anywhere on the bus to which you will connect the pin.

If the type of bus is compatible with the type of pin, connection lines are drawn to show the bus connection.

Connecting Ports

You can create nets to connect ports on component instances. To create a net, you assign the same net name to all of the ports you want to connect.

Port connections *cannot* be seen as nets drawn on the block diagram. All of the nets shown on the block diagram are bus connections.

To connect ports on two component instances:

Note: This procedure describes how to connect a port on one component instance to a port on another component instance. Using a similar procedure, you can connect ports on more than two component instances, connect multiple ports at the same time, or create system ports.

1. Double-click one of the component instances you want to connect.
2. In the Properties dialog box, click the **Ports** entry in the tree view on the left side of the dialog box.
3. In the box under **Show Ports**, choose the type of ports appearing in the ports list (**With No Default Nets**, **With Default Nets**, **All Ports**, or **New Filter**).
4. Note that ports With Default Nets need not be connected, they will be automatically connected by PlatGen. The user needs to connect these ports only when the connection is not desired.
5. In the **Show Ports** list, select the a port to which you will assign a net.
6. Click **Add**.

   The selected port is copied to the **Explicit Port Assignments** list.
7. In the **Explicit Port Assignments** list, modify the fields describing the port connection (**Polarity**, **Range**, etc.) and assign the net connected to the port a **Net Name**.
8. Perform Steps 1 through 6 for the second component instance. If you assign the same **Net Name** to a port on each component instance, the ports are connected.

Viewing and Editing System Ports

You can view and edit the all of the system ports (that is, all of the ports designated **External**) in a single dialog box. Using this dialog box, you can also add power and ground ports to the system.

To view and edit system ports:

1. Double-click an area in the workspace that does not contain any objects.
2. If you want to add power or ground system ports to the design:
   a. Click Add.
   b. In the Add External Port dialog box, enter a **Port Name** and select **GND (net_gnd)** or **VCC (net_vcc)**.
   c. In the Add External Port dialog box, Click **OK**.
3. Edit the entries in the **System Ports** table as desired.

   Some notes about the table:
   - Fields that the you can edit are displayed in white; read-only fields are displayed in grey.

- If you click the heading of a column, the entries in the column are displayed in alphabetical order. If the click the column heading again, the entries in the column are displayed in reverse alphabetical order.
- You can remove a system port by selecting it and clicking **Remove**.

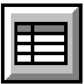
4. When you have finished your edits, click **OK**.

Viewing and Editing All of the Ports in the System

You can view and edit the all of the ports in the system (internal and external) in a single dialog box. Using this dialog box, you can also print a port list or export the ports as a CSV (Comma Separated Value) file formatted for the PBD Editor or for the Xilinx PACE (Pinout and Area Constraints Editor) tool.

To view and edit all of the ports in the system:

1. Select **Add** → **Ports** or click the Add Ports toolbar button.



2. If you want to print the **System Ports** table, click **Print**.
3. If you want to export the ports to a CSV file:
   a. If you only want to export selected ports, select the ports to export.
   b. Click Export.
   c. In the Export Ports dialog box, enter a **CSV File Name**, select an **Output Format** of **PBD Editor** or **PACE**, and specify whether you want to export **All Ports** or **Selected Ports**.
   d. In the Export Ports dialog box, Click **OK**.
4. Edit the entries in the **System and Component Ports** table as desired.

   Some notes about the table:

   - Fields that the you can edit are displayed in white; read-only fields are displayed in grey.
   - If you click the heading of a column, the entries in the column are displayed in alphabetical order. If the click the column heading again, the entries in the column are displayed in reverse alphabetical order.
5. When you have finished your edits, click **OK**.

Viewing and Editing Interrupts

You can view and edit the interrupts driving a component. Not all components have interrupt ports, and most components that use interrupts have only one interrupt port.

An interrupt may be driven by more than one net. If an interrupt is driven by multiple nets, you must specify the priority of each net driving the interrupt.

To edit the interrupts driving a component instance:

1. Double-click the component instance in the workspace.
2. In the Properties dialog box, click the **Interrupts** entry in the tree view on the left side of the dialog box.

3. In the Component Interrupts dialog box, select the Interrupt you wish to configure in the **Interrupt Port** box.
4. In the **Possible Interrupt Nets** box, select the nets that will drive the internet.

   To select multiple nets, click the first net name, then press the `Ctrl` key and click the additional net names.

   Note: If the interrupt port is a scalar port (that is, its range is blank) then only one net may be selected to drive the interrupt. An interrupt controller must be used in such a case to manage the interrupts, and the controller's output port should be used as the single input to the component with the scaler interrupt port.
5. Click **Add** to move the nets to the **Interrupt Drivers** box.
6. In the **Interrupt Drivers** box, use the **Move Up** and **Move Down** buttons to list the nets in priority order.

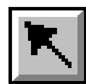
   Nets higher in the list will be serviced before nets lower in the list.
7. Click **OK**.

Editing the Block Diagram

Selecting Objects

To Select objects in the workspace:

1. Select **Edit** → **Select Object(s)**, or click the Select toolbar button.

   The **Options** tab shows the **Select Options**.
2. In the **Options** tab, set the following options:
   - Click **Select the entire bus** or **Select the line segment** to specify whether the bus or just the line is selected when you click a bus line.
   - Click **Keep the connections to other objects** or **Break the connections to other objects** to specify whether connections to other objects are retained when you move an object.
   - Click **Are enclosed by the area** or **Intersect the area** to specify which objects to select when you drag a bounding box around an area. **Are enclosed by the area** selects only those object that are completely enclosed in the bounding box.
3. Click the object to select it.

The PBD Editor also has these extended selections:

- If you hold the **Shift** key while you select an object, it is added to the current selections
- If you hold the **Ctrl** key while you select an object, its status is toggled (that is, it will be selected if it was not selected and deselected if it was selected).
- **Edit** → **Select** All selects all objects on the current sheet.
- **Edit** → **Unselect All** unselects all objects on the current sheet.

Viewing Object Information

To view information about an object in the workspace, place the cursor over the object. A box appears supplying information about the object (name, IP name, bus pin type, etc.).

Zooming in the Workspace

You can use menu commands to zoom the display in the workspace.

Zooming Behavior	Menu Command	Toolbar Icon
Zoom in	Select **View** → **Zoom** → **In**, or click the Zoom In toolbar button.	
Zoom out	Select **View** → **Zoom** → **Out**, or click the Zoom Out toolbar button.	
Zoom to display the entire schematic or symbol in the workspace	Select **View** → **Zoom** → **Full View**, or click the Zoom Full View toolbar button.	
Zoom to an area you select	Select **View** → **Zoom** → **To Box**, or click the Zoom To Box toolbar button. Zoom in or out as follows: • To zoom in, draw a bounding box around the area from the top left corner of the area to the bottom right corner. • To zoom out, draw a bounding box from the bottom right corner to the top left corner.	
Zoom to display selected objects at the highest magnification	1. Select the objects you want to center in the workspace. 2. Select **View** → **Zoom** → **To Selected**, or click the Zoom To Selected toolbar button	

Drawing Non-Electrical Objects

Non-Electrical Objects are graphic only and have no electrical meaning in the block diagram. You can draw these non-electrical objects in the PBD Editor:

- Arcs
- Circles
- Lines
- Rectangles
- Text

To draw a non-electrical object:

1. In the **Add** menu, select the object (**Arc**, **Circle**, **Line**, **Rectangle**, or **Text**) you want to draw, or select the toolbar icon for the object.

Object	Toolbar Icon
Arc	
Circle	
Line	
Rectangle	
Text	

2. If any options appear in the Options tab, select the appropriate options for the object.
3. Click to start drawing the object.
4. Drag the cursor until the object is the appropriate size.
5. If necessary, move the cursor to adjust the object.

   For example, when you draw an arc you must move the cursor until the arc appears as you want it to display.

You can draw as many objects as you want until you select another command.

XPS “No Window” Mode

XPS “no window” mode can be invoked by typing the command **xps -nw** at the command prompt. It provides functionality to generate MSS file, makefile, and run the complete XPS flow in batch mode. Users can also create an XMP project file or load an XMP project file created by the XPS GUI.

When invoking the batch mode for XPS, users can specify a tcl script along with **-scr** option. XPS sources this Tcl script and then provides a command prompt to the user. Users can also provide an existing project (XMP) file as input to xps. XPS will load the project before presenting the command prompt to the user.

In 6.3, XPS batch provides new ability to query EDK design database. New Tcl commands have been added for this purpose.

Available Commands

XPS-Batch provides you a Tcl shell interface. You can use the commands in Table 2-3.

Table 2-3: **XPS-Batch commands**

Command	Description
load [mhs | xmp | new | mss |] <*filename*>	Loads the MHS/XMP file and opens/creates XPS project. Updates project with MSS file. Input <filename> is optional when loading MSS. Users can create an empty project with suboption new
save [mss | xmp | make | proj]	Saves the corresponding file. Option proj will save all files
xset *option* <*value*>	This command sets the value of a field (corresponding to option) to the given value. Refer to Section "Setting Project Options".
xget *option*	This command displays the current value of the field (corresponding to option). Refer to Section "Setting Project Options".
run *option*	Executes makefile with appropriate target. Refer to Section "Executing Flow Commands"
xadd_swapp <*name*> <*procinst*>	Add a new Software Application with given name and associated with given processor instance
xdel_swapp <*name*>	Delete the given Software Application from the project
xadd_swapp_progfile <*name*> <*filename*>	Add given program file to the given software application
xdel_swapp_progfile <*name*> <*filename*>	Delete given program file from the given software application
xset_swapp_prop_value <*name*> *option* <*value*>	Set value of a particular property of the given software application. Refer to Section "Setting Options on a Software Application" for a list of options
xget_swapp_prop_value <*name*> *option*	Get value of a particular property of the given software application. Refer to Section "Setting Options on a Software Application" for a list of options
xget_handle [mhs | mss | merged_mhs | merged_mss]	Get handle to the MHS or MSS design, or to the merged MHS or MSS file. Refer to chapter "EDK Tcl APIs" for details on handle and merged MHS/MSS handles
exit	Closes the project and exits out the XPS

Creating a New Empty Project

For creating a new project with no components, use the command

```
load new <basename>.xmp.
```

XPS will create a project with an empty MHS file and will also create the corresponding MSS file. All the files have same basename as the xmp file. If XPS finds an existing project

in the directory with same basename, then the XMP file is overwritten. However, if MHS, or MSS file with same name is found, then they are read in as part of the new project.

Creating a New Project With Given MHS

For creating a new project, use this command:

```
load mhs <basename>.mhs
```

XPS will read in the MHS file and create the new project. The project name will be same as MHS basename. All the files generated will have the same name as MHS. After reading in the MHS file, XPS will also assign various default drivers to each of the peripheral instance, if a driver is known and available to XPS.

Opening an Existing Project

If you already have a XMP project file, you can load that file using this command:

```
load xmp <basename>.xmp
```

XPS will read in the XMP file and load the project. Project name will be same as XMP basename. Note that XPS will take the name of MSS file from the XMP file, if specified. Otherwise, it will assume these files based on the XMP file name. If XMP file does not refer to an MSS file, but the file exists in the project directory, XPS will read that MSS file. If the file does not exist, then XPS will create a new MSS file.

Reading an MSS File

You can read an MSS file using this command:

```
load mss <filename>
```

Note that if the user does not specify <filename>, it is assumed to be the file associated with this project. Loading an MSS file will override any earlier settings. For example, if you specify a new driver for a peripheral instance in the MSS file, the old driver for that peripheral will be over ridden.

Saving Files and Your Project

Users can save MSS, XMP and make files for your project using this command:

```
save [mss|xmp|make|proj]
```

Command **save proj** will save all the files.

Setting Project Options

Users can set various project options and other fields in XPS using the xset command. Users can also display the current value of those fields by using xget commands. The xget command also returns the result as a Tcl string result which can be saved into a Tcl variable. The various options taken by the two commands are shown in Table 2-4.

```
xset option [value]
xget option
```

Table 2-4: **Options for command xset and xget**

Option Name	Description
arch	Set target device architecture
dev	Set target part name
package	Set package of the target device
speedgrade	Set speedgrade of the target device
searchpath [*dirs*]	Set the Search Path as semicolon separated list of directories
hier [top | sub]	Set the design hierarchy
topinst [*instname*]	Set the name by which processor design is instantiated (if submodule)
hdl [vhdl | verilog]	Set HDL language to be used
sim_model [structural | behavioral | timing]	Set current simulation mode
simulator [mti | ncsim | none]	Set simulator for which you want simulation scripts generated
sim_x_lib sim_edk_lib	Set the simulation library paths. For details, please refer to SimGen chapter
pnproj [nplfile]	Set the ProjNav Project file where design will be exported
addtonpl	If NPL file exists, specify whether XPS should add to that file or should overwrite it
synproj [xst | none]	Set the synthesis tool to be *xst* or *none*
intstyle	Set instyle value
usercmd1	Set user command 1
usercmd2	Set user command 2
pn_import_bit_file	Set the bit file to be imported from ProjNav
pn_import_bmm_file	Set the bmm file to be imported from ProjNav
reload_pbde	Set GUI option to reload PBDE or recreate every time
main_mhs_editor	Set GUI option about main_mhs_editor

Executing Flow Commands

Users can run various flow tools by using the run command with appropriate option. XPS will create a makefile for the project and run that makefile with appropriate target. Note that XPS generates the makefile everytime the run command is executed. Valid options for the run command are shown in Table 2-5.

```
xget option
```

Table 2-5: **Options for command run**

Option Name	Description
netlist	Generate netlist
bits	Run Xilinx Implementation tools flow and generate bitstream
libs	Generate software libraries
bsp	Generate VxWorks bsp for given ppc405 system
program	Compile user program into ELF file(s)
init_bram	Update bitstream with BRAM initialization information
ace	Generate SystemACE file after .bit file is updated with BRAM info
simmodel	Generate simulation models (does not run simulator)
sim	Generate simulation models and run simulator
download	Download bitstream onto the FPGA
exporttopn	Export the processor design to ProjNav
importfrompn	Import .bit and .bmm files from ProjNav
netlistclean	Delete ngc/edn netlist
bitsclean	Delete .bit, .ncd, and .bmm files in implementation directory
hwclean	Delete implementation directory
libsclean	Delete software libraries
programclean	Delete ELF file(s)
swclean	Calls libsclean and programclean
simclean	Delete simulation directory
clean	Delete all tool generated files and directories
resync	Updates any MHS file changes into the memory
assign_default_ drivers	Assigns Default drivers to all peripherals in the MHS file and saves to MSS file.

Reloading an MHS File

All EDK design files refer to MHS. Any changes in MHS have impact on other design files too. If there are any changes in the MHS file after you loaded the design, use the following command:

```
run resync
```

This will cause XPS to re-read MHS, MSS and XMP file again.

Adding a Software Application

Users can add new software application projects in XPS batch using the xadd_swapp command. When adding a new sw application, users must specify a name for that

application and a processor instance on which that application will be run on. By default, XPS assumes that the ELF file related to a new software application will be created at <*swapp_name*>`/bin/`<*swapp_name*>`.elf`. This can be changed once the application has been created.

```
xadd_swapp <swapp_name> <proc_inst>
```

Deleting a Software Application

An already existing software application can be deleted from project in XPS batch using the xdel_swapp command. Users must specify the name of the software application they want to delete.

```
xdel_swapp <swapp_name>
```

Adding a Program File to a Software Application

Users can add any program file (C source or header files) to an existing software application using the xadd_swapp_progfile command. The name of the swapp to which the file needs to be added and the location of the program file needs to be specified. Based on the extension of the file, XPS automatically adds it as a source or header.

```
xadd_swapp_progfile <swapp_name> <filename>
```

Deleting a Program File from a Software Application

Users can delete any program file (C source or header file) associated with an existing software application using the xdel_swapp_progfile command. The name of the swapp and the program file location needs to be specified.

```
xdel_swapp_progfile <swapp_name> <filename>
```

Setting Options on a Software Application

Users can set various software application options and other fields in XPS using the xset_swapp_prop_value command. Users can also display the current value of those fields by using xget_swapp_prop_value command. The xget_swapp_prop_value command also returns the result as Tcl string result. The various options taken by the two commands are shown in Table 2-6.

```
xset_swapp_prop_value <swapp_name> <option_name> [value]
xget_swapp_prop_value <swapp_name> <option_name>
```

Table 2-6: **Options for commands xset_swapp_prop_value and xget_swapp_prop_value**

Option Name	Description
sources	Can be used only for displaying a list of sources. For adding sources, use xadd_swapp_progfile command.
headers	Can be used only for displaying a list of headers. For adding header files, use xadd_swapp_progfile command.
executable	Path to the executable (ELF) file.

Table 2-6: **Options for commands xset_swapp_prop_value and xget_swapp_prop_value**

Option Name	Description
download	Option to specify whether the ELF for this SwProj should be used for initializing BRAMs or not. Values are true or false.
procinst	The processor instance associated with this sw application.
compile_sources	Option to specify whether this software application ELF should be compiled within XPS, or whether it is compiled outside XPS (in this case, XPS expects precompiled ELF to be present. Value can be true or false.
compileroptlevel	Specify compiler optimization level. Values can be from 0 to 3.
globptropt	Specify whether to perform Global Pointer Optimization. Value can be true or false.
debugsym	Debug Symbol Setting. Value can be from 0 to 2 corresponding none, -g and -gstabs options.
searchcomp	Compiler Search Path Option (-B)
searchlibs	Library Search Path Option (-L)
searchincl	Include Search Path Option (-I)
lflags	Libraries to Link (-l)
propopt	Options passed down to the preprocessor (-Wp)
asmopt	Options passed down to the assembler (-Wa)
linkopt	Options passed down to the linker (-Wl)
progstart	Program Start Address
stacksize	Stack Size
heapsize	Heap Size
linkerscript	Linker Script (-Wl,-T -Wl,<linker_script_file>)
progccflags	Other compiler Options which can not be set using the above options

Settings on Special Software Applications

For every processor instance, there is a **Bootloop** application provided by default in XPS. For microblaze instances, there is also a Xmdstub application provided by XPS. The only setting available on these special software applications is to “Mark for BRAM Initialization”. The **xset_swapp_prop_value** can be used. XPS no window mode will recognize <*procinst*>`_bootloop` and <*procinst*>`_xmdstub` as special software application names. For example, if the processor instance is mymblaze, then XPS will recognized mblaze_bootloop and mblaze_xmdstub as software applications. Users can set the init_bram option on this application.

```
XPS% xset mblaze_bootloop init_bram true
XPS% xset mblaze_xmdstub init_bram false
```

Note however, that this assumes that there is no user software application by the same name. If there exists a user application with same name, then you will not be able to change the settings using the XPS Tcl interface. Thus, in XPS no window mode, you should not create an application with name <***procinst***>`_bootloop` or <***procinst***>`_xmdstub`. This limitation is valid only for XPS no window mode and does not apply if you are using the GUI interface.

Closing a Project and Exiting

For closing the project, you can use this command:

exit

This will also save the project and close XPS. Thus, you can only work on a single project during a single execution of the batch mode version of XPS.

Limitations and Workarounds

MSS Changes

XPS-batch supports limited MSS editing. So, if user wants to make any changes in the MSS file, he/she will have to hand-edit the file, make the changes and then run the “load mss” command to load the changes into XPS. Note that user does not have to close the project. S/he can save the MSS file, edit it and then just re-load it into the project by using load mss command.

XMP Changes

It is not recommended to change the XMP file by hand. XPS-batch supports changing of project options through commands. It also supports adding of source and header files to a processor, and setting any compiler options. Any other changes must be done from the XPS GUI.

Chapter 3

Base System Builder

The Base System Builder (BSB) wizard is a software tool that help users quickly build a working system targeted at a specific development board.

Based on the user's board selection, BSB will offer the user a number of options for creating a basic system on that board. These options include processor type, debug interface, cache configuration, memory type and size, and peripheral selection. For each option, functional default values will be preselected in the GUI.

If the user's target development is not available or not currently supported by the BSB, the user may select the Custom Board option rather than select a target board. Using this option, the user must specify the individual hardware devices that they expect to have on their custom board. In order to run the generated system on a custom board, the user must be sure to enter the FPGA pin location constraints into the UCF file. BSB will automatically insert these constraints into the UCF file in the case where a supported target board is selected.

Upon exit of BSB, a hardware specification (MHS) file and software specification (MSS) file will be created and loaded into the user's XPS project. The user may then optionally further enhance the design in the Xilinx Platform Studio (XPS) GUI.

The Base System Builder will also optionally generate a software project called "TestApp" which contains a sample application and linker script and can be compiled and run on the hardware on the target development board. Note that XPS supports multiple software projects for every hardware system, each of which may contain its own set of source files and linker script.

This chapter contains the following sections.

- "BSB Flow"
- "Limitations"

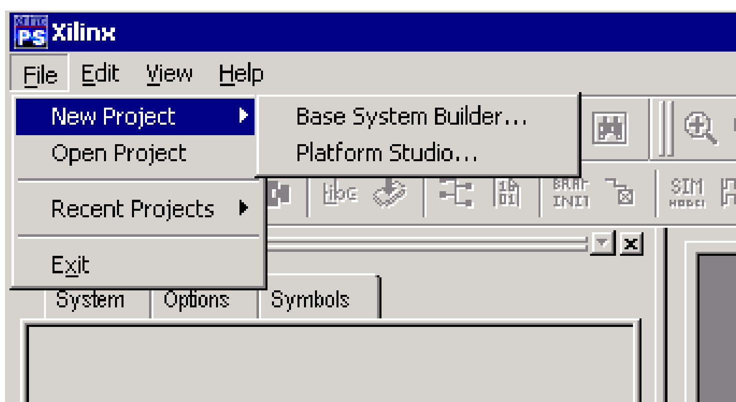
BSB Flow

This section describes the steps the user will go through in the BSB wizard. Note that each page of the wizard contains a **More Info** button at the bottom which will provide a detailed explanation of the functions of that page.

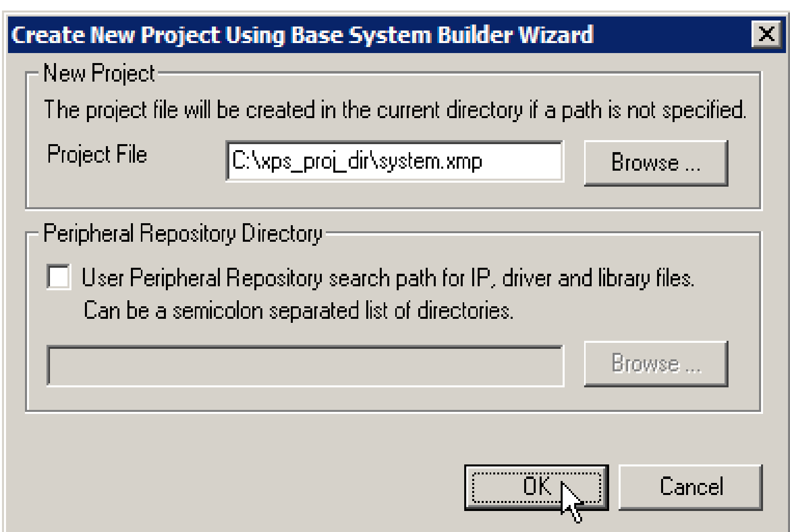
Invoking BSB

The Base System Builder can only be invoked when creating a new XPS project.

Invoke BSB by selecting **File** → **New Project** → **Base System Builder**. .



In the Create New Project dialog box, enter or browse to the directory where you would like to create a new XPS project. It is recommended that you start with a clean directory because any existing project files, including the XMP, MHS, and MSS files, may be overwritten when your new XPS project is being created.



Selecting a Starting Point

There are two starting options in BSB:

1. Create a new design

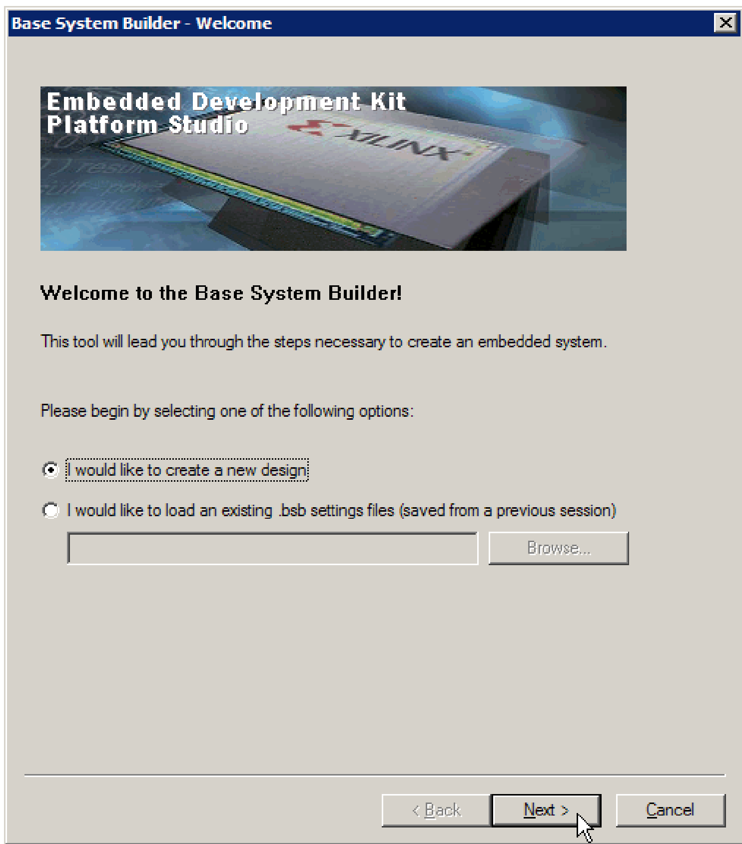
   This option should be selected if you are using BSB for the first time or if you are creating a brand new design in BSB.

2. Load an existing BSB file

   This option should be used if you have used BSB previously to generate a BSB settings file. A BSB settings file is created by BSB upon exit and stores all the GUI selections made by the user in that wizard session. When the file is loaded in a subsequent session, all the saved selections will be preloaded into the GUI. The user may generate

an identical system but just clicking "Next" through all the wizard pages, or they may make changes to the GUI to generate a different system. A new BSB settings file is always created upon exit of the BSB wizard, reflecting the final GUI selections of the current session. This feature may be useful to users who want to create several projects with similar designs. The BSB file is not a text file and is not intended to be modified by the user.

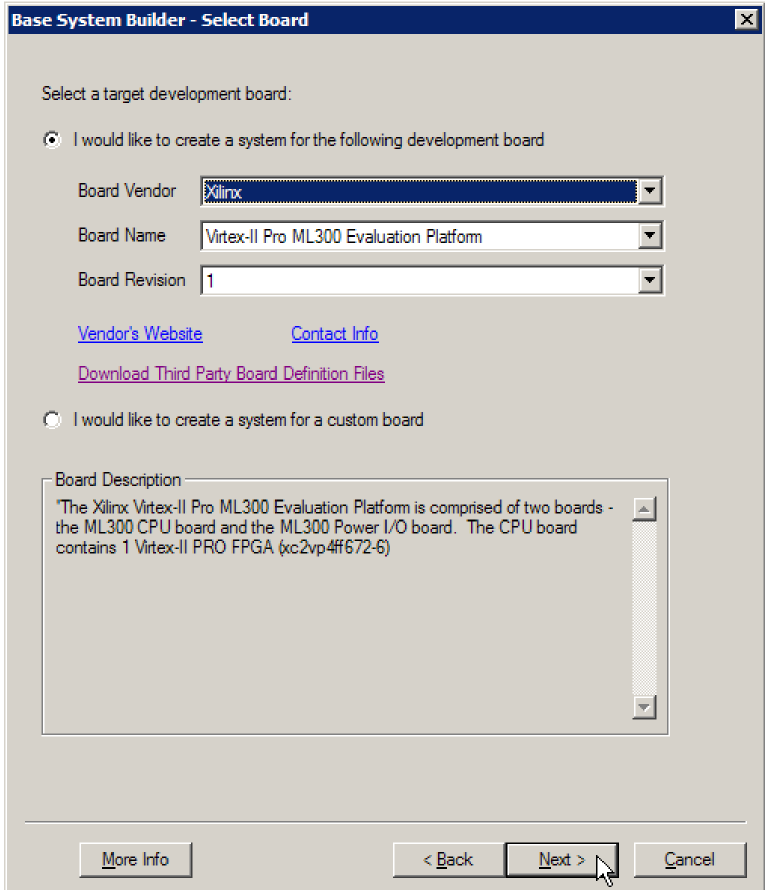
It is important to note the BSB setting file stores only BSB GUI selections and does not reflect any changes made to the system outside of BSB. For example, if a user adds or edits a core in the XPS GUI or manually edits the MHS file.



Selecting a Target Development Board

Users must begin by selecting a target development board. Board selection is indicated by the vendor name, board name, and revision number. A brief description of the currently selected board is displayed on this page, showing the Xilinx FPGA device, memories, and IO devices available on that board.

If the target board is not available or not supported in the drop-down list on this page, the user may select the Custom Board option. Selecting this option may require user to input more information into the GUI on subsequent wizard pages.

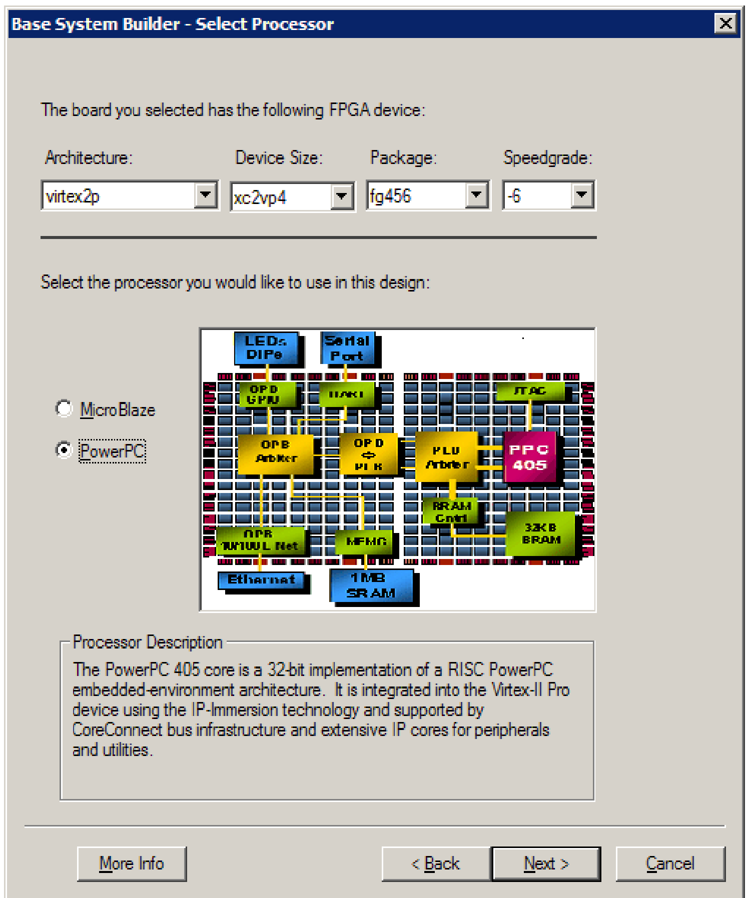


Selecting a Processor

Currently, the Base System Builder supports two processors: MicroBlaze, a configurable "soft" processor implemented in FPGA logic, and the PowerPC 405 processor, a hardware device available only in some Xilinx FPGA architectures. If the PowerPC is unavailable in the FPGA device on your development board, this selection will be disabled in the GUI.

A brief description of the currently selected processor is displayed on this page, along with an illustration of what a typical system using this processor might look like.

If the Custom Board option was selected, the user must specify the actual FPGA device that they will use. If a specific target board was selected on the previous page, the device information for the FPGA on that board will be displayed but can not be changed by the user.

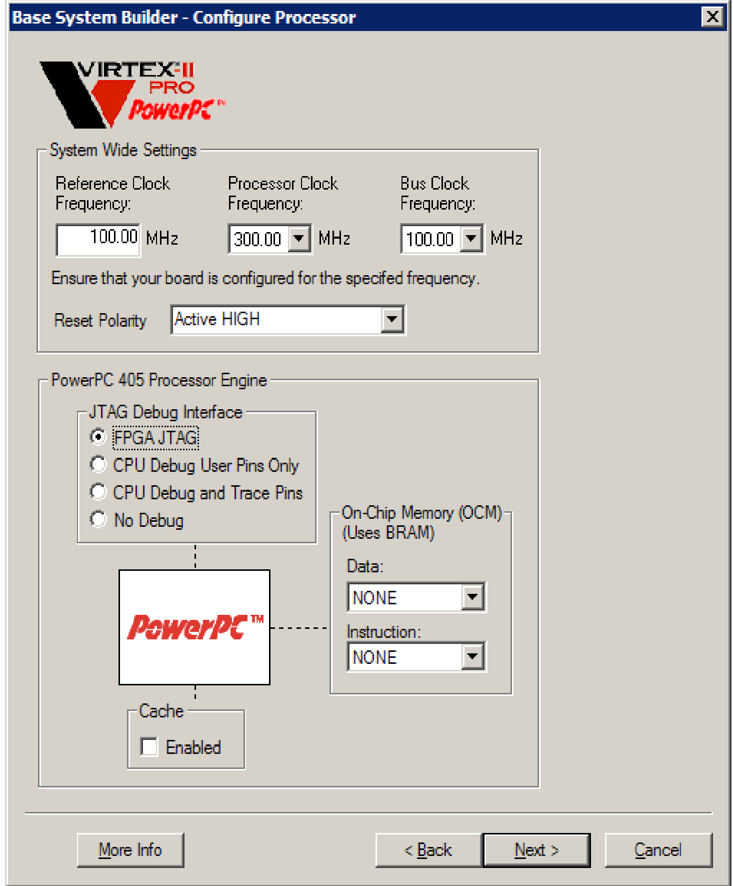


Configuring Processor and System Settings

Based the processor selected in the previous page, the user can configure certain system and processor specific settings.

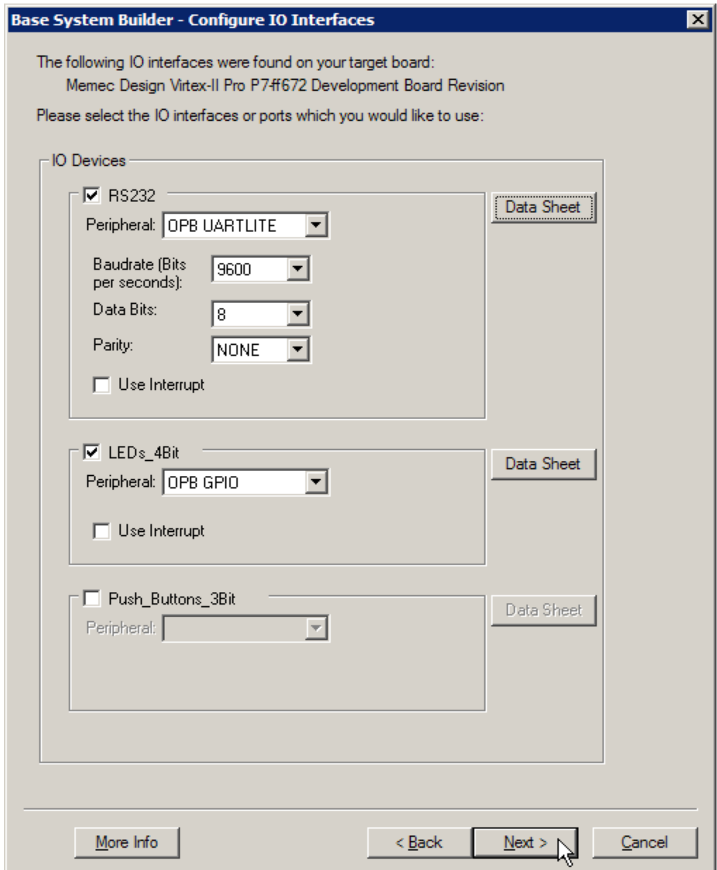
System settings include processor and bus clock frequencies. Allowable values may be restricted by the clock resources available on the target development board or the on-chip resources available in the FPGA device. If the Custom Board option was selected, the user may specify the reference clock frequency available on the custom board as well as the polarity of the external reset switch.

Processor specific settings include debug interfaces, cache options, and configuration of any on-chip memory which communicate over a processor-specific bus.

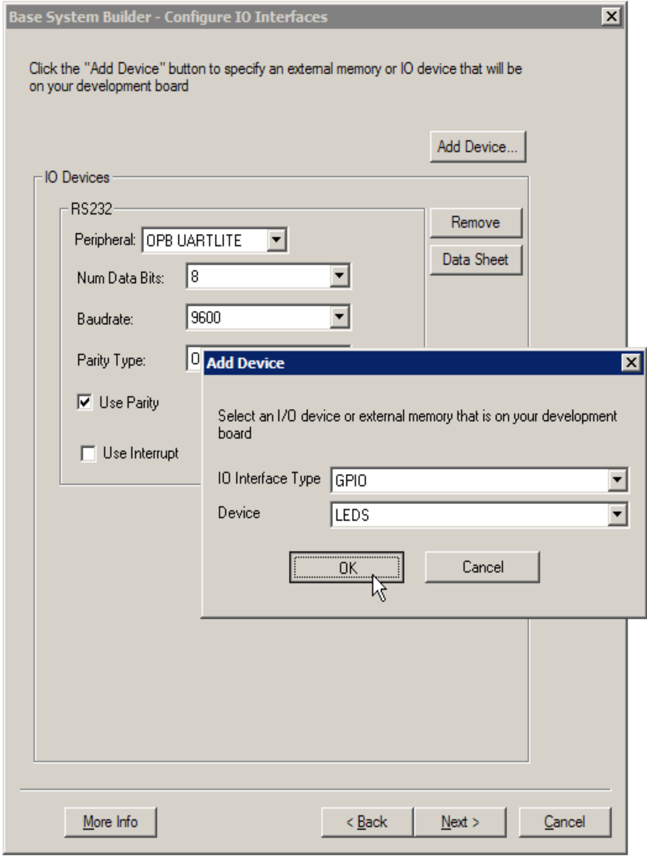


Selecting External Memories and I/O Devices:

If a specific target board was selected, BSB will determine what external memory and I/O devices are available on that board. Each device found will be displayed in the GUI with a checkbox next to it which will enable or disable that device interface.



If the Custom Board option was selected, the user must specify all external memory and I/O devices that will be on the custom board. The GUI will provide an **Add Device** button that will open a dialog box that the user can use to add a device. Any device added this will be enabled. A **Remove** button is available to remove devices.



For all enabled external memories and I/O device interfaces, either determined by the selected board or added by the user in the Custom Board option, the user will select from a list of IP cores which can be used to control that memory or interface. BSB will instantiate the selected core in the system, connect it to the appropriate bus, and automatically set any parameters which are dictated by the on-board device that core is controlling. For ease of use, most core parameter values can not be explicitly set by the user in the BSB GUI. The BSB wizard is designed to select default parameter values which will create a functional base system on a specific development board. If needed, users may manually change the parameter values in the generated MHS file. It is recommended that users using the Custom Board option manually check the MHS file to ensure that the IP is parameterized correctly for the hardware devices on the custom board.

For each device interface enabled, BSB will create the necessary top-level system ports. If a specific board was selected, these ports will be assigned to the correct FPGA pin locations in a generated UCF file. In a custom board was selected, the user must enter these pin locations manually into the UCF file before they are able to run this design on actual hardware.

Depending on the number of devices on the board, the IO Devices Selection panel may span across several wizard pages. The Back button can be used to view or edit previous selections at any time while the wizard is active.

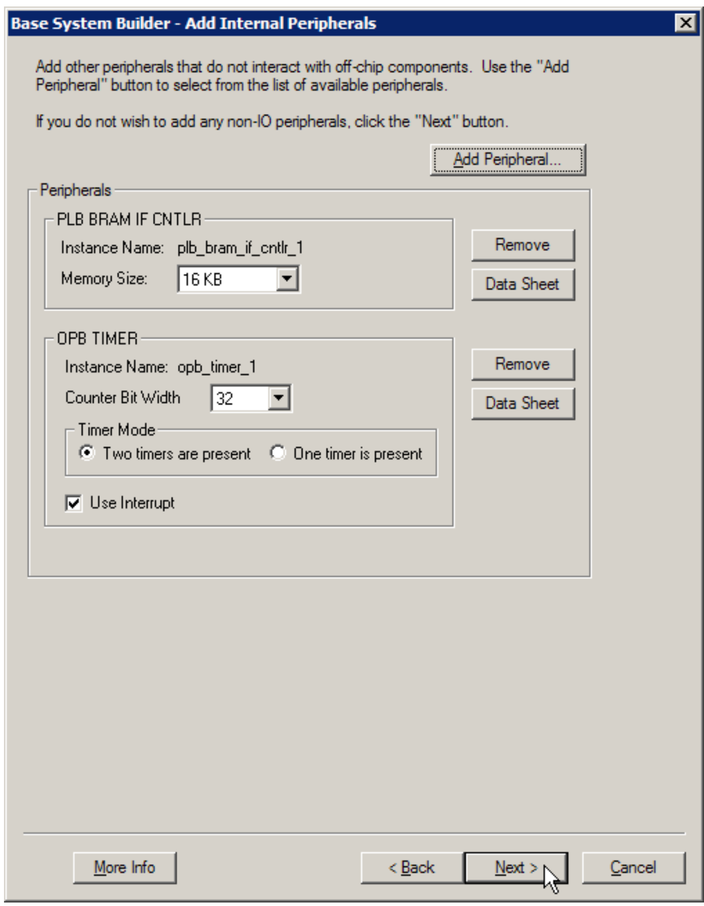
If you are unsure about what IP core to use, you may click the **Data Sheet** button on the right to view the data sheet of the currently selected core.

Adding Internal Peripherals

Internal peripherals are IP cores which do not communicate directly with any devices outside of the FPGA. Examples of such peripherals are on-chip memory (BRAM) controllers and timers. The user may add internal peripherals by clicking the **Add Peripheral** button at the top of this page and selecting from a list of internal peripherals. Any selections added by default or by the user can be removed by clicking the **Remove** button next to that device.

Depending on the number of internal peripheral devices added by the user, additional wizard pages may be created to display the current list. The **Back** button may be used to remove or edit previous selections.

The Base System Builder will instantiate all internal peripherals which are added to the system and connect them to the appropriate bus. It will NOT generate any top-level system ports for internal peripherals.



Configuring Software Settings

The Base System Builder will generate a software project for this hardware system containing a sample C application and linker script for the hardware system. This application is intended to verify system "aliveness" and also to provide an illustration of how to create a simple application. The contents of this program will depend on the

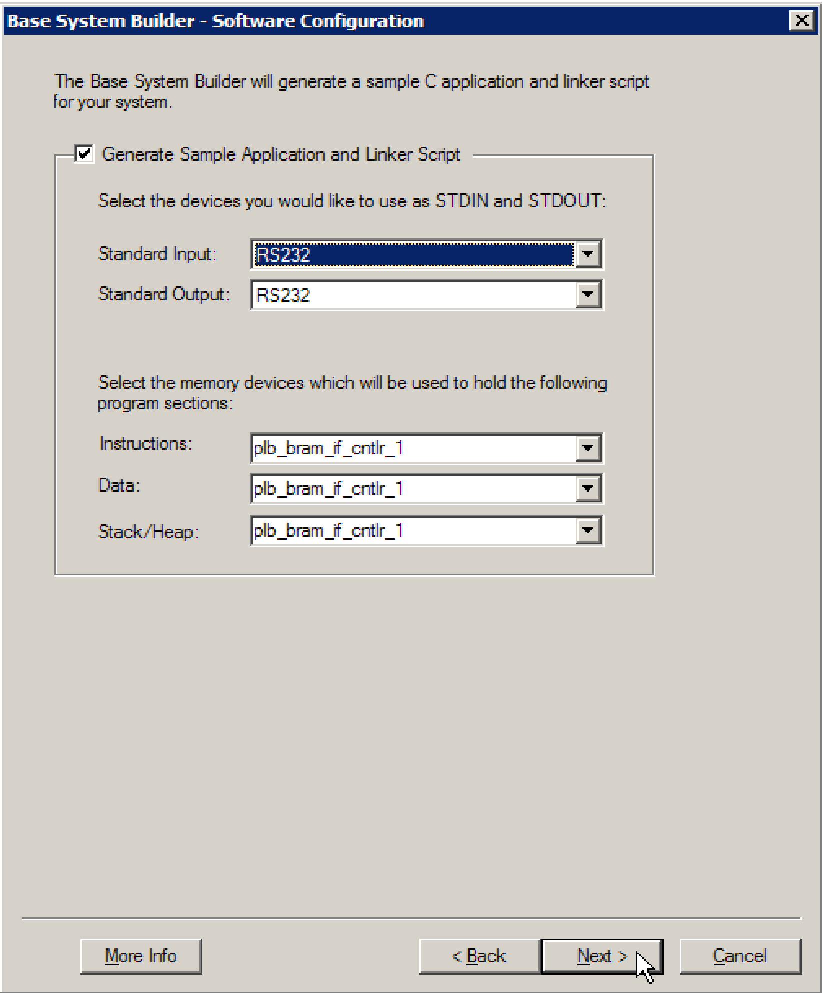
hardware components which are included in the system as well as the options selected in this page.

If a standard output peripheral is selected, the generated application will include a print function call to the device selected.

The user may select the memory devices where different sections of the program should be placed in. It should be noted that if any part of the program is placed in external memory, the user will need to have access to a debugger tool (such as XMD) which can download the program onto that external memory device. By default, BSB will place the entire application in internal BRAM memory (unless there are no BRAMs added in the system). This configuration allows the user to include the application in the FPGA configuration bitstream, and thus, the software application can run upon power-up or reset.

The generated application will include a simple memory read/write test to all memories in the system which are writeable (not a ROM), do not hold *any* parts of the application itself, and do not reside on the reset vector address for the processor.

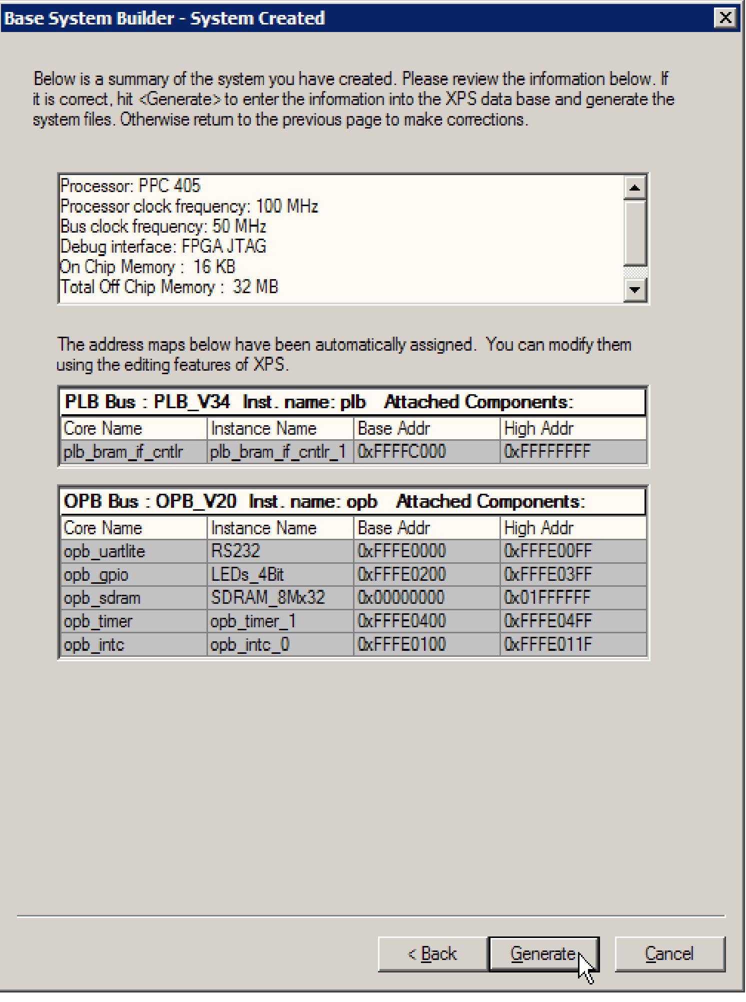
The user may choose to not generate the sample application and linker script by deselecting the checkbox at the top of this page.



Generating the System and Address Map

Before generating the output files, the Base System Builder will display a summary of the system you have created. This page contains a table of IP cores which are instantiated in the system as well as the address map for these devices. The device addresses generated by BSB conform to addressing requirements of each IP core and cannot be modified in the BSB GUI. Users can manually change the address values in the generated MHS file, but are encouraged to consult the data sheets for individual IP cores to avoid entering illegal address values.

At this point, the user may use the **Back** button to make changes to previous selections, or click the Generate button to complete the wizard and generate all output files.



Output Files

The list of generated files are displayed on the final page of the Base System Builder Wizard. These files include

- `system.mhs`

  Microprocessor Hardware Specification file consisting of component instantiations, parameterization, and connections.

- `system.mss`

  Microprocessor Software Specification file consisting of default driver names for each hardware component, including processor and OS, and parameterization of drivers if needed.

- `data/system.ucf`

  Xilinx User Constraints File containing constraints such as timing, FPGA pin locations, FPGA resource specification, and IO standards. If the Custom Board option is used, this file will not be complete! User will have to manually enter FPGA pin locations and possibly other constraints determined by the hardware on the custom board.

- `etc/fast_runtime.opt`

  Options file containing default options which will be used by the Xilinx implementation tools if run from XPS.

- `etc/download.cmd`

  Xilinx download command file which can be used to run iMPACT (the download tool) in batch mode. This file uses the iMPACT *identify* command, which assumes that the user has installed the necessary data files for all devices on the JTAG chain on the development board. This file may be modified by the user, if necessary. Please consult the iMPACT documentation for more information.

Optional files:

- `TestApp/src/TestApp.c`

  Sample application source file

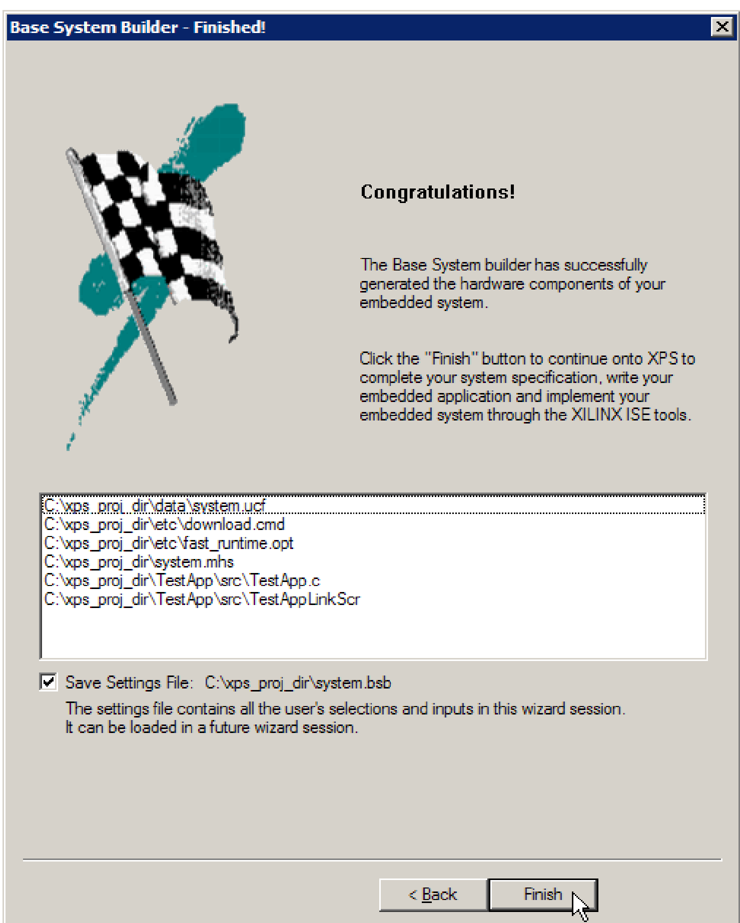
- `TestApp/src/TestAppLnkScr`

  Linker script defining what memory locations to place each section of the application program in.

- `system.bsb`

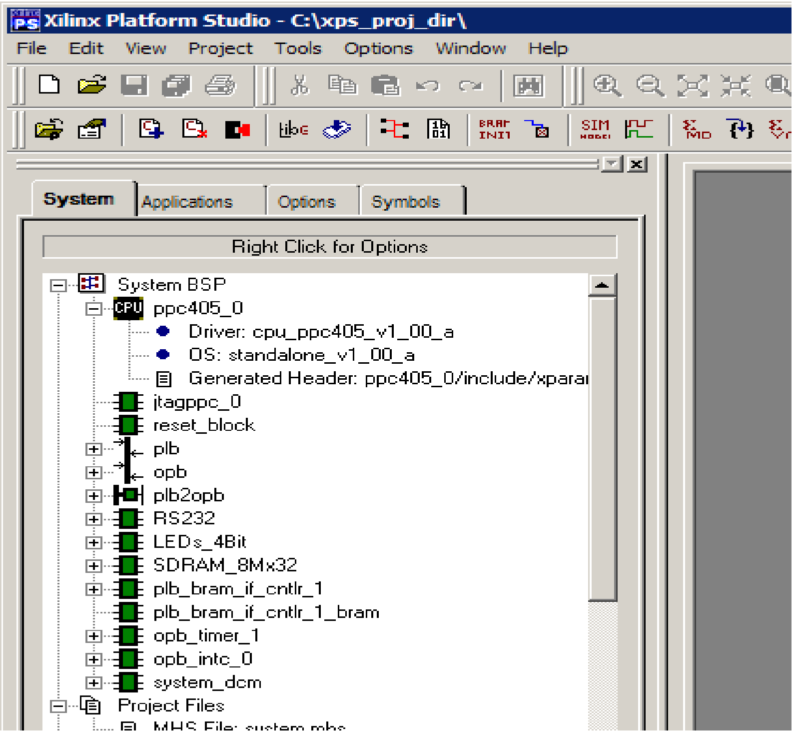
  BSB specific settings file which can be loaded into a subsequent BSB session to automatically load the same GUI selections that were made in this session

.



Exiting BSB

Upon exit of the Base System Builder, the user will find the XPS GUI opened to the newly created project. In addition to generating the output files described above, BSB will also set some project (XMP) and software (MSS) parameters which may be necessary for the system that was built. These parameters will be saved when you save the XPS project.



Limitations

The Base System Builder was designed for users who want to create a basic functional system quickly. As such, it does not allow users to create advanced systems or specify very specific configurations.

The following are known limitations of the Base System Builder wizard:

- BSB does not support multi-processor systems
- BSB does not allow users to specify or modify the address map
- BSB does not check for specific hardware resources on the target FPGA device. The user must consult the data sheet for the FPGA they are using to ensure that it contains enough logic elements and other resources required by the system they are creating.
- Systems generated by BSB are not guaranteed to meet timing.

Any system that is created by the Base System Builder can be further enhanced either in the XPS GUI or by manually modifying the design files generated by BSB. Therefore, advanced users can also use the Base System Builder as a starting point for building a complex design.

Chapter 4

Create/Import Peripheral Wizard

The Xilinx Embedded Design Kit (EDK) comes with a large number of commonly used peripherals. Many different kinds of systems can be created with these peripherals, but it is likely that you may have to create your own custom peripheral to implement functionality not available in the EDK peripherals library.

The Create/Import Peripheral Wizard helps you create your own peripherals and import them into EDK compliant repositories or Xilinx Platform Studio (XPS) projects.

In the *Create* mode, this tool creates a number of files. Some of these files are templates which will help you implement your peripheral without needing to have a detailed understanding of the bus protocols, naming conventions or the formats of special interface files required by the EDK. By referring to the examples in user logic module and using various auxiliary design support files that output by the wizard, you can quickly get started on designing your custom logic.

In the *Import* mode, this tool will help you create the interface files and directory structures that are necessary to make your peripheral visible to the various tools in the EDK. For this mode of operation, it is assumed that you have followed the naming conventions required by the EDK. Once imported, your peripheral will be like any other module available in the EDK peripherals library.

These modes are described in the following sections:

- "Invoking the Wizard"
- "Creating New Peripherals"
- "Importing an Existing Peripheral"
- "Organization of Generated Files"
- "Limitations"

Invoking the Wizard

The Create/Import Peripheral Wizard can be invoked from XPS before you create or open an XPS project, or directly from Windows **Start** menu outside of XPS.

Invoke Create/Import Peripheral Wizard from XPS by selecting **File →Create/Import Peripheral**

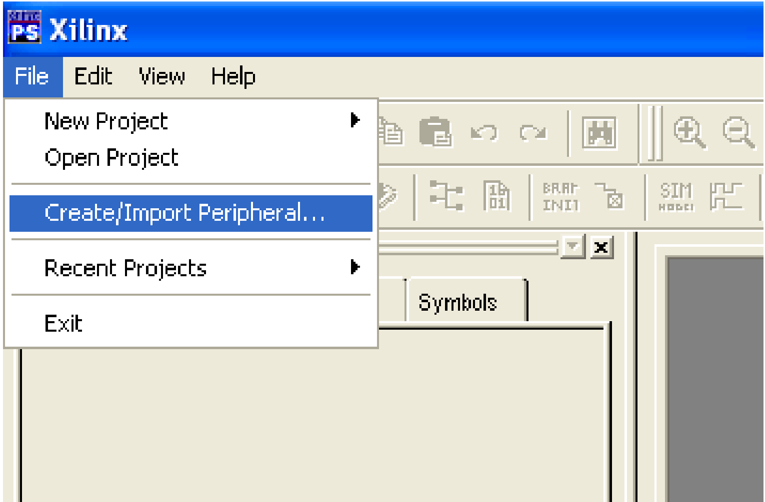


Figure 4-1: **Invoke Create/Import Peripheral Wizard from the XPS Menu**

User can view various CoreConnect and IPIF documentations through the hyperlinks listed on the welcome screen.

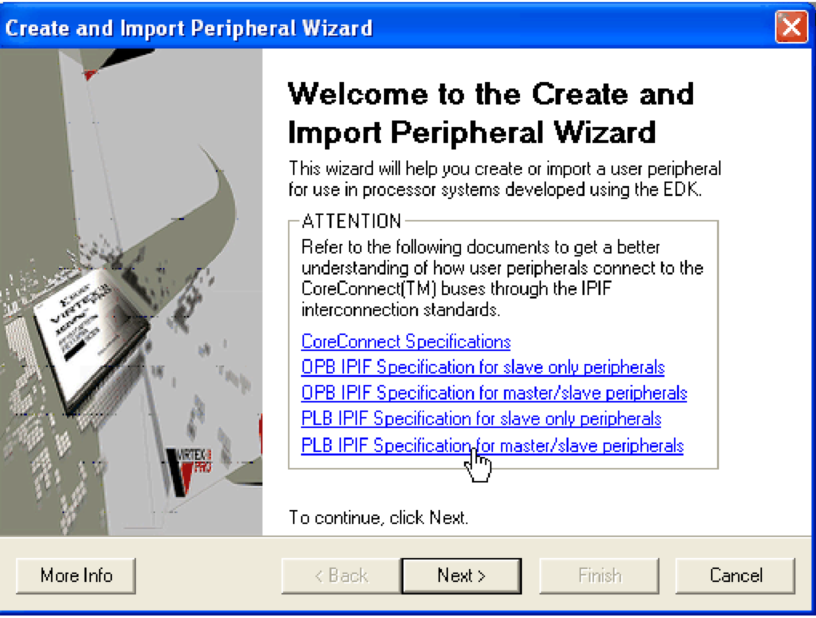


Figure 4-2: **Welcome to the Create/Import Peripheral Wizard**

User choose to open up the *Create* mode, see next “Creating New Peripherals” section for this flow.

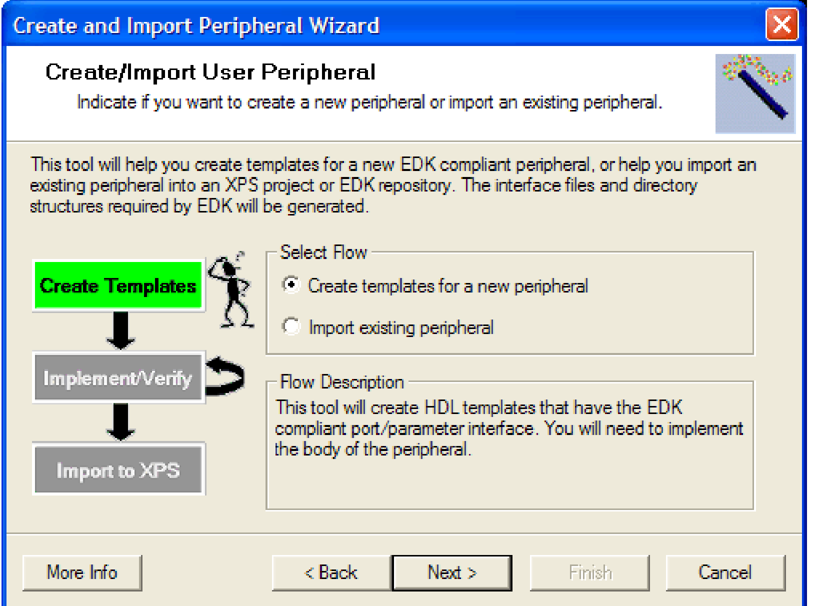


Figure 4-3: **Choose Create Mode**

Or open up the *Import* mode, see the “Importing an Existing Peripheral” section for this flow.

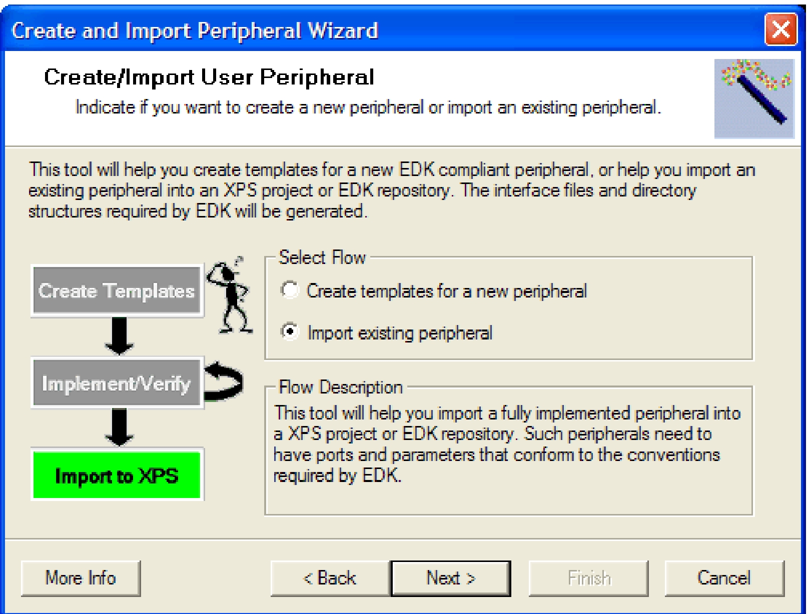


Figure 4-4: **Choose Import Mode**

Creating New Peripherals

In this mode, this tool helps you create a peripheral suitable for instantiation into systems designed using the EDK. You will have to answer a few simple questions and this tool will output a number of HDL files that conform to the conventions and rules required by the EDK. You will have to implement the body of one of the outputted HDL blocks. The interface to this block is very generic: you will not have to fully understand the intricacies of the CoreConnect bus protocol to implement your peripheral.

The current limitations of the tool include the following:

- Supports VHDL only

  This is because the underlying library elements are implemented in VHDL. Future releases are likely to support a mixed-language development mode where the user-logic module is written in Verilog.

EDK compliant peripherals have the following components:

- A Bus Interface

  This is just a set of ports that the peripheral must have to connect to the targeted bus.

- A component called the IP Interface (IPIF)

  The bus interface connects to this component. Additionally, it provides a lot of functionality that most EDK compliant peripherals need. These include address decoding, addressable registers, interrupt handling, read/write FIFOs, DMA, etc. This component is structurally parameterizable, and therefore only the required logic is implemented.

- A component that implements the application specific logic that cannot be implemented in the IPIF

  This has been called *user-logic* in the subsequent discussion.

The user-logic interfaces to the IPIF through a set of ports called the IP Interconnect (IPIC). These ports are designed to simplify the implementation of the user-logic.

This tool will guide you through a set of panels that help you customize each of the above elements.

Peripheral creation involves the following:

- Indicate module name and destination, i.e. the XPS project or EDK repository in which the peripheral must be stored
- Select the bus type to which the peripheral is targeted
- Select and configure IPIF (Intellectual-Property interface) services. These are common functionality required by most peripherals. If selected, the amount of HDL code the user has to write is minimized
- Implement user-logic in generated files. This part require the use of common HDL based design flows

It should be noted that the VHDL template files output by this tool is already a complete and working design, various sample code snippets have been put in the user-logic module to demonstrate the features that you have selected. You'll be able to use the output template in the way same as any other EDK peripherals even without touching any piece of the VHDL code, and by referring to these example codes, it should help you to implement your custom functionality.

Identifying the Physical Location of Your Peripheral

The EDK requires that all HDL and interface files representing your peripheral be stored in a predefined directory structure under a XPS project or EDK peripherals repository. The EDK repository is the more versatile storage mechanism because many XPS projects can access one EDK repository. This tool takes care of creating the right directory structures and interface files.

In this panel, you indicate whether you want a XPS project or EDK repository, and what the physical location of the XPS project or EDK repository is. A XPS project is a directory with a `.xmp` file. A EDK repository is a directory.

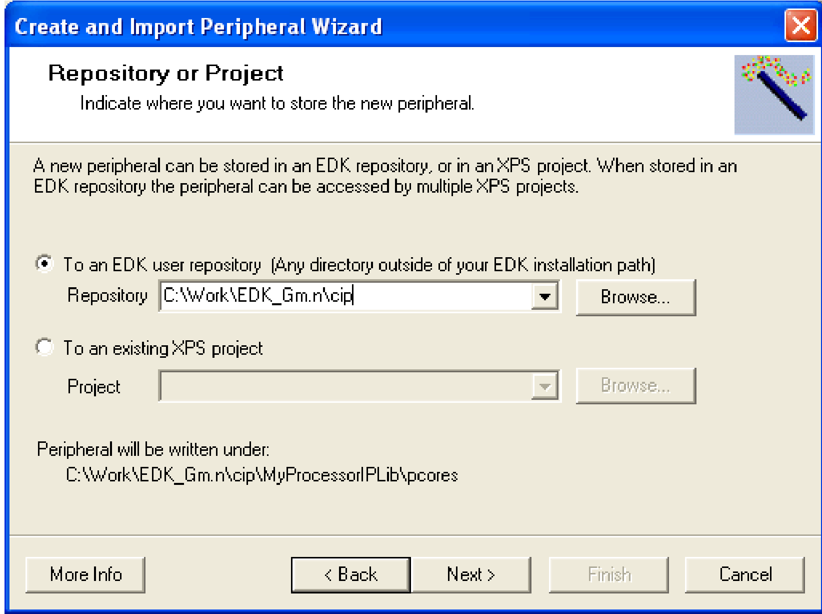


Figure 4-5: **Identifying the Physical Location of a Peripheral**

The actual core directory gets created in one of the following based on whether you choose EDK repository or XPS project:

<*EDK-Repository-Dir*>`/MyProcessorIPLib/pcores`

or

<*Directory-containing-XPS-Project-File*>`/pcores`

Identifying Module and Version

In this panel you do the following:

- Indicate the name of your top module Typically this is the name of the top module in the design hierarchy that makes up the peripheral.
- Indicate the version identifier for your module The version identifier for the core has three components: a major revision number, a minor revision number and a hardware/software compatibility identifier.

The EDK requires that the top module and (possibly) other sub-modules for your core be compiled into a logical library named after the top module and the version number. The rules are best described through the following examples:

Table 4-1: **Naming conventions for peripherals using version identifiers**

Peripheral Name	spi46
Major version	9
Minor Version	12
Software/hardware compatibility identifier	g
Logical library name	spi46_v9_12_g

Table 4-2: **Naming conventions for peripherals not using version identifiers**

Peripheral Name	spi46
Logical library name	spi46

It is very important that all the elements of this peripheral are compiled into the indicated logical library or into some other logical library already available in the XPS project or in any of the currently accessible EDK repositories. This tool will actually process only the files that are compiled into the logical library indicated by the above examples. Other files are assumed to be available in the XPS project, or in any of the currently accessible repositories. Naturally, this means that the library and use lines in your VHDL need to use this logical library name.

Additionally, the chosen library name cannot be **work**.

The subsequent sections of this document deals with details of peripheral creation or peripheral import using this tool.

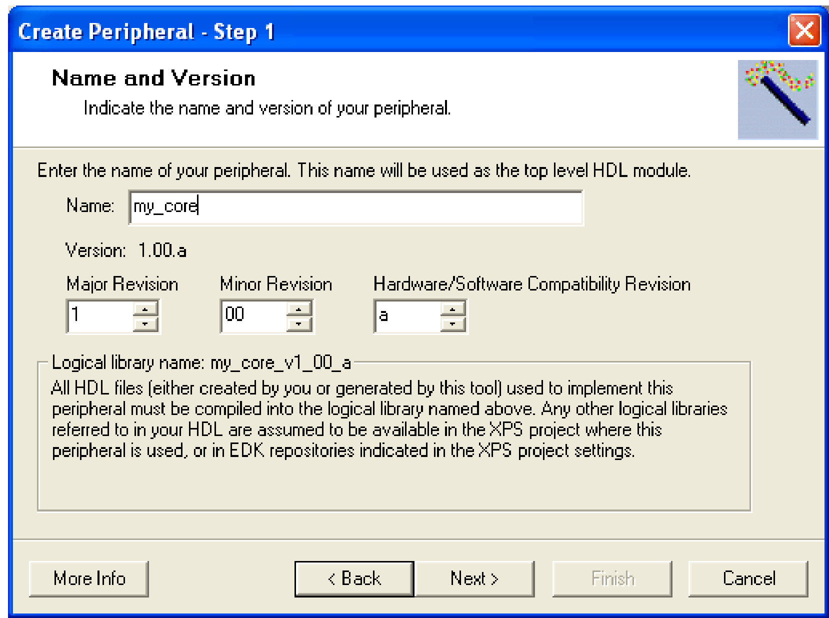


Figure 4-6: **Module Name and Version**

Select Bus Interface

In this panel you indicate the CoreConnect bus-interface, *i.e.* if your peripheral is a fast (but more complicated) PLB (Processor Local Bus) or a comparatively simpler and slower OPB (on-chip peripheral bus) peripheral.

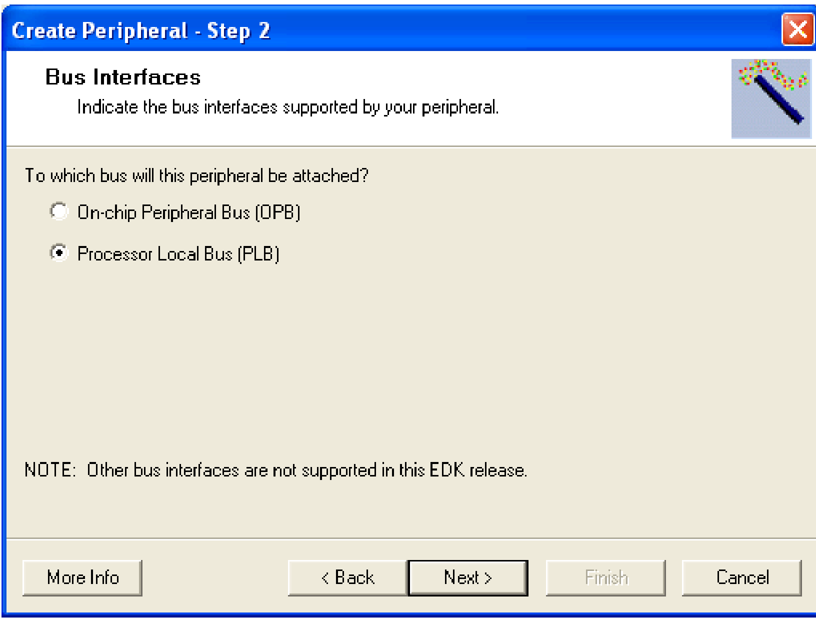


Figure 4-7: **Select Bus Interface**

Select IPIF Services

All user peripheral templates created with this tool incorporate a module called the IPIF (Intellectual Property Interface.) There are two kinds of IPIFs: PLB and OPB. One side of this interface implements the PLB or OPB interface, and the other side implements the IPIC (intellectual-property interconnect) interface. The user peripheral implements the IPIC. The IPIC is bus agnostic, hence it is possible to create user modules with a IPIC interface that can operate on both a PLB or OPB. Additionally, the IPIC is 'hardware friendly' and thus easier to work with.Table 4-3.

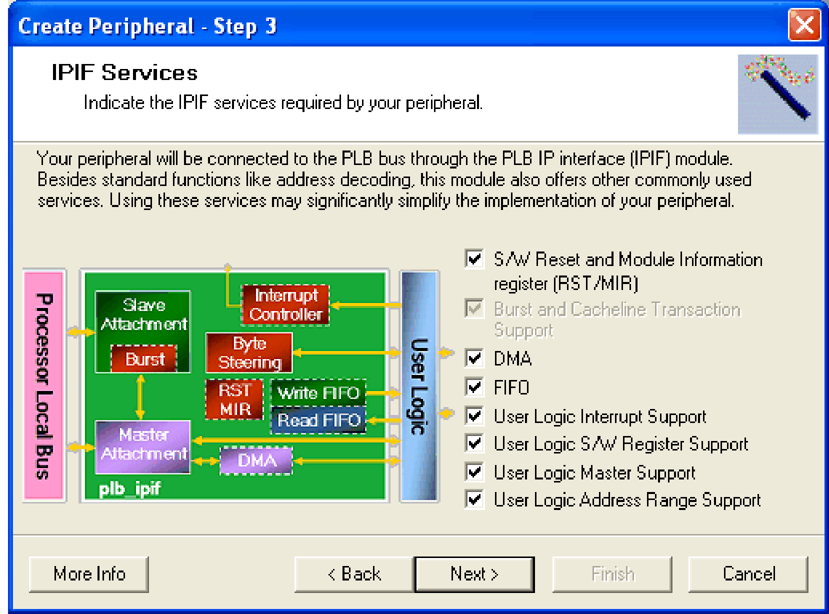


Figure 4-8: **Select IPIF Services**

The IPIF provides some very basic services like slave attachment, address decoding, byte steering, and some optional services that may greatly simplify the task of creating your peripheral. Based on the services you selected, the wizard will create corresponding PLB or OPB templates with slave-only operation or master-slave combined operation for you. Note choose either the DMA service or user-logic master support service will trigger the wizard to generate a master-slave combined template instead of slave-only template.

These features are described below.

Table 4-3: **IPIF Services**

IPIF Feature	Description
Include Software Reset and Module Information registers	The peripheral will have a special write only address. When a specific word is written to this address, the IPIF will generate a reset signal for the peripheral. The peripheral should reset itself using this signal. This allows individual peripherals to be reset from the software application. The peripheral will also have a read-only register that will identify the revision level of the peripheral.
Include Burst Cache line Transaction Support	Burst and cache line transactions allow the bus master to issue a single request that results in multiple data values being transferred. Support of these transactions requires significant hardware resources. Presently, the 'fast' burst mode is used. Cache line is available for the PLB peripherals only.
Include DMA	The IPIF part of the peripheral will have a build in DMA service. Using the DMA service will automatically enable the burst support to optimize data transactions.
Include FIFO	The IPIF part of the peripheral will have a built in FIFO service.
User-logic interrupt support	The peripheral will have a interrupt collection mechanism that manages the interrupts generated by the user-logic and the IPIF services and generate a single interrupt output line out of the peripheral.
Include Software Addressable Registers support in user-logic	The user-logic part of the peripheral will have registers addressable through software.
Include Master support in user-logic	This will include the IPIC master interface signals for user logic master operations. It will also include example HDL of a simple master operation model.
Include Address Range support in user-logic	This will generate enable signals for each address range. This feature is useful for peripherals that need to support multiple address ranges, e.g. multiple memory banks. The distinction between this and other cases is that the enable signals are generated for each address range of the address space supported by the peripheral, rather than for each addressable register in the user-logic module.

Configure DMA

DMA is available in the IPIF for peripherals that may need Direct Memory Access support, such as ethernet interface. The DMA component will setup two channels, which can be used as either transmit or receive channel, operating in Simple DMA mode. (Packet mode Scatter Gather DMA mode would be supported in a future release.)

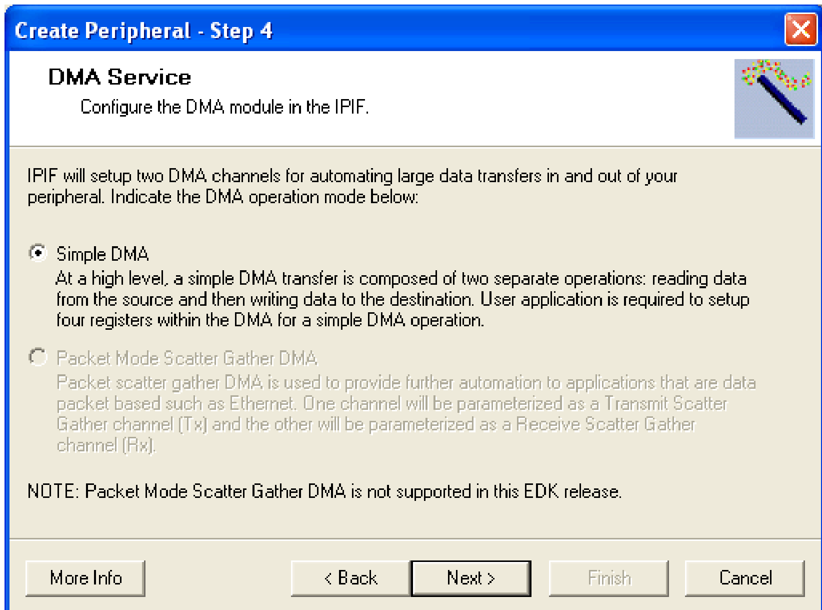


Figure 4-9: **Configure DMA**

Configure FIFOs

FIFOs are available in the IPIF for peripherals that may need data buffering support. Two types of FIFOs are provided: Read Packet FIFO and Write Packet FIFO

In this panel, you choose to include a Read and/or Write FIFO. You also configure the FIFO by indicating the number of entries it can store (i.e. its depth) and the size of each word (byte, half-word, word or double.) Other features such as packet mode access and signals that indicate FIFO vacancy, etc. can also be requested.

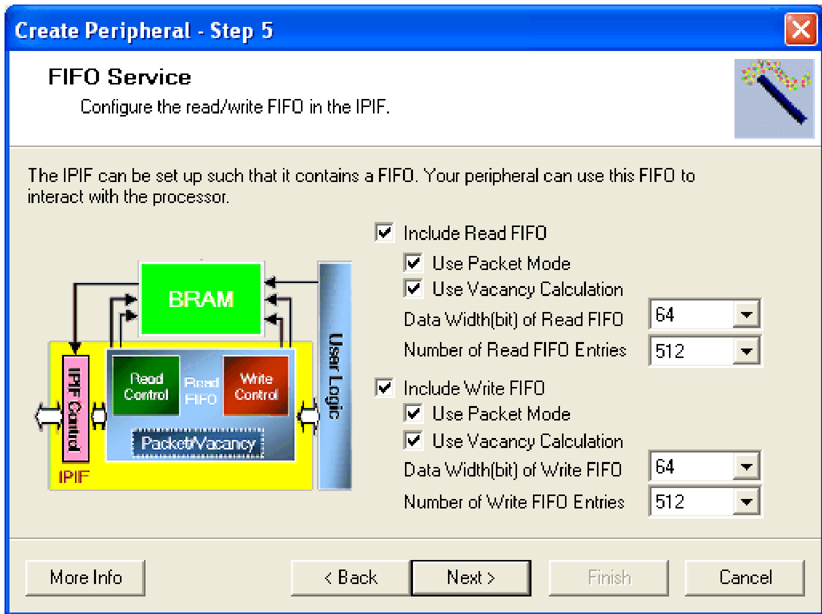


Figure 4-10: **Configure Read/Write FIFOs**

Configure Interrupt Handling

The peripheral will have an interrupt collection mechanism that manages the interrupts generated by the user-logic and the IPIF services and generates a single interrupt line out of the peripheral.

An addressable register based mechanism for enabling/disabling the interrupts generated by the peripheral is provided, as are registers to determine the status and source of the interrupts.

The interrupts generated by the user-logic part of the peripheral are first processed by a 'IP Interrupt Source Controller' or 'IP ISC'. The interrupt signal out of this controller is then fed into the a 'device interrupt source controller' or 'device ISC' in the IPIF where they are processed in conjunction with the interrupts generated out of the other IPIF services. The IP ISC has a software addressable interrupt enable register (IP IER) that may be used to enable/disable interrupts from the software application. Both the IP ISC and 'device' ISC are implemented in the IPIF component of the core.

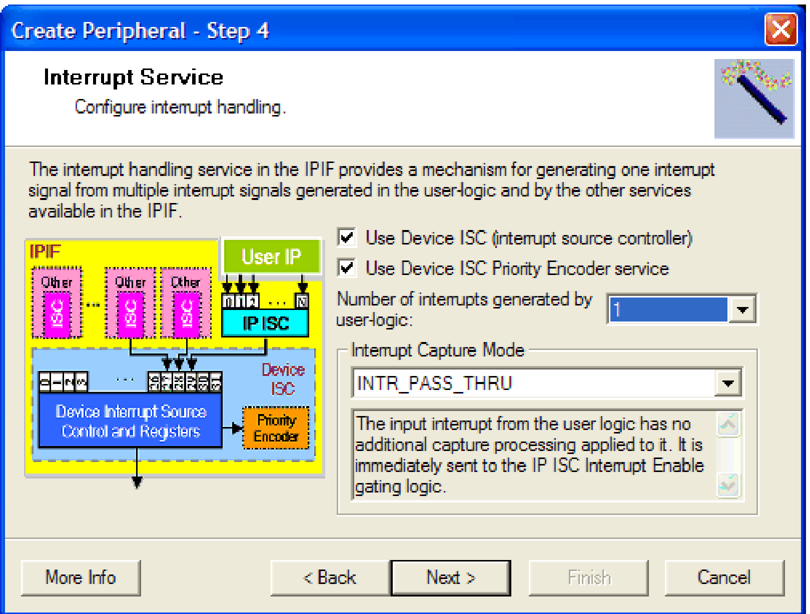


Figure 4-11: **Configure Interrupt Handling**

In this panel, you will have to indicate the number of interrupts generated by the user-logic, and the capture mode of these interrupts.

The following interrupt capture modes are supported:

- INTR_PASS_THRU

  The interrupt from the user logic has no additional capture processing applied to it. It is immediately sent to the IP ISC interrupt enable logic (IP IER) and thence to the 'device' ISC.

- INTR_PASS_THRU_INV

  The input interrupt from the user logic is logically inverted but has no additional capture processing applied to it. The inverted interrupt level is passed through the IP IER and sent to the 'device' ISC interrupt enable logic. This mode is mainly used to capture active- low interrupts.

- INTR_REG_EVENT

  The IP ISC Status Register will sample the IP Interrupt input at the rising edge of each bus clock pulse. If a logic high is sampled, the bit of the IP Interrupt Status Register corresponding to the input interrupt position will stay high until the User Application (ISR) clears the interrupt.

- INTR_REG_EVENT_INV

  This capture mode is the same as the INTR_REG_EVENT mode except that the IP Interrupt is logically inverted before it enters the sample and hold logic of the IP interrupt status register.

- INTR_POS_EDGE_DETECT

  The IP ISC Status Register will sample the interrupt input at the rising edge of each bus clock pulse. A one bus clock delayed sample will also be maintained. The new sample and the delayed sample will be compared. If the new sample is logic high and the old

sample is logic low (a rising edge event), the IP Interrupt Status Register will latch and hold a logic ‘1’ for the interrupt bit position. Once latched, the bit of the IP Interrupt Status Register corresponding to the input interrupt position will stay high until the user application (interrupt service routine) clears the interrupt.

- INTR_NEG_EDGE_DETECT

  The IP ISC Status Register will sample the interrupt input at the rising edge of each bus clock pulse. A one bus clock delayed sample will also be maintained. The new sample and the delayed sample will be compared. If the new sample is logic low and the old sample is logic high (a falling edge event), the IP Interrupt Status Register will latch and hold a logic ‘1’ for the interrupt bit position. Once latched, the bit of the IP Interrupt Status Register corresponding to the input interrupt position will stay high until the user application (interrupt service routine) clears the interrupt.

You will also have to indicate if you want to include the interrupts generated outside of the user-logic block (in the other IPIF services) by checking the ‘Use Device ISC (Interrupt Source Controller)’ check box. You can also choose to use the priority encoder service offered by the IPIF. If the device interrupt service controller is not chosen, then only the interrupts generated by the user-logic are recognized and processed through a user-logic specific interrupt service controller. Figure 4-12 gives a general indication of the implementation of the interrupt services in the IPIF. Note that including DMA service will automatically enable the Device ISC implicitly even if user has no user-logic interrupts, this will allow software application to detect completion of DMA transactions via interrupt mode instead of polling mode.

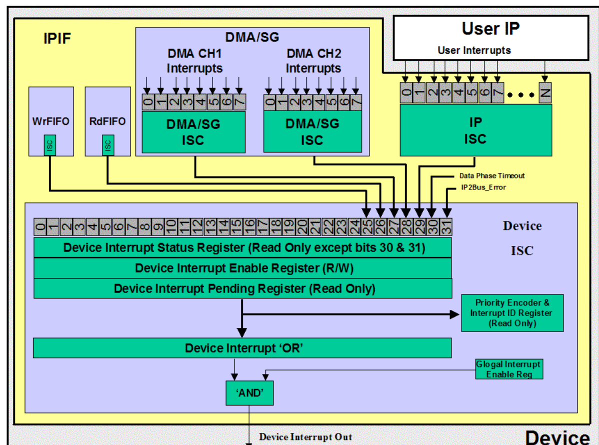


Figure 4-12: **The Interrupt Service in the IPIF**

The Device ISC Priority Encoder service of the IPIF is basically a function that loops on the device ISC pending register keeping track of the ordinal position the highest priority interrupt source. The priority is from LSB to MSB, meaning bit 31 of pending register has the highest priority while bit 0 has the lowest priority. For example, if bit 29, 26 and 25 of pending register are ‘1’, then the interrupt ID register will have value 2 since bit 29 has the higher priority. (The order of the bits is from LSB to MSB).

This service is meant to be used by the software application. When this service is enabled, the software application can set up a vector table to map different interrupt service routine for each interrupt bit of the pending register, and use the Device ISC Interrupt ID register to map the identifier of the actual interrupt. This is considered to be more efficient than using code (`if-elsif-else`) to implement priority interrupt handling.

Note that the peripheral is sometimes referred to as a 'device' in this tool and associated documentation. 'Device' just refers to the peripheral in question, not the FPGA!

Additionally, it is important to understand that the interrupts discussed here are processed by the IPIF, not directly by the interrupt controller processing the interrupts sent to the processor. The types of interrupts that can be processed by the interrupt controller in the processor system are of the form described under "Interrupt Signals" in the "Importing an Existing Peripheral" section of this chapter.

Configure Software Accessible Registers

If this option is selected, this tool will add software accessible registers in the generated user-logic template. It will also include example HDL to read and write these registers by byte, half-word, word or double-word (for PLB). This HDL indicates how these registers are read and written.

This is among the most useful features of this tool. You can easily use these registers to feed data into and from other hardware.

In this panel, you indicate the number and size (byte, half-word, word, or double) of these registers. We recommend the size of these registers be the same as the data-width of the bus to which it is connected, 32 bits for OPB peripherals and 64 bits for PLB peripherals. This will allow for a smaller implementation of the IPIF by optimizing out the implementation of the byte-steering logic.

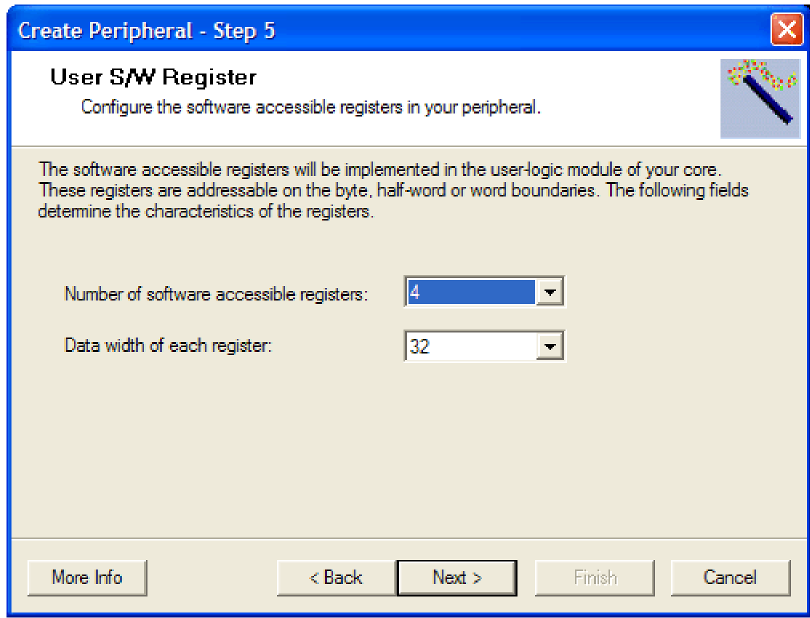


Figure 4-13: **Configure Software Accessible Registers**

Configure Address Ranges

Certain peripherals like memory controllers support multiple address ranges. This IPIF service provides you IPIC ports that help you work with multiple address ranges. Enable signals for each range is provided.

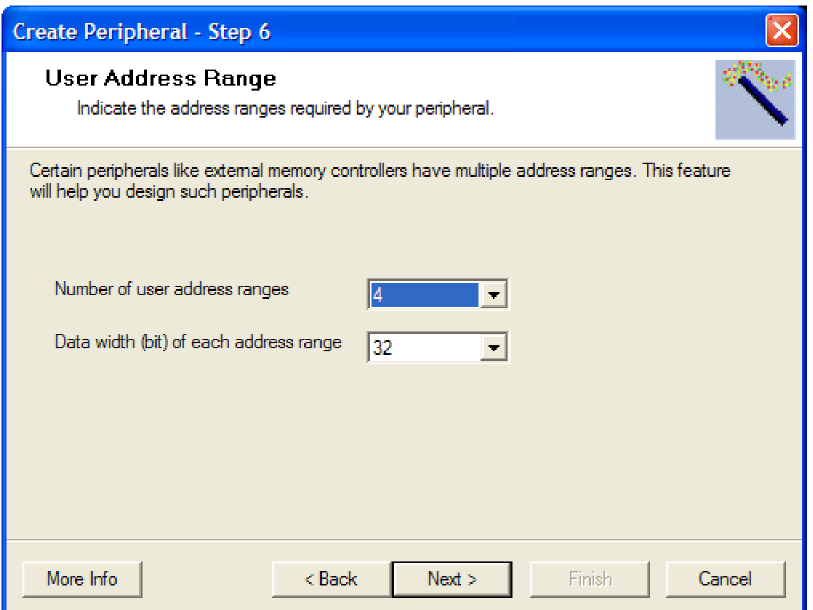


Figure 4-14: **Configure Address Ranges**

You will need to indicate the number of address ranges, and the size (byte, half-word, word and double-word) of the data being accessed. We recommend the size of these registers be the same as the data-width of the bus to which it is connected, 32 bits for OPB peripherals and 64 bits for PLB peripherals. This will allow for a smaller implementation of the IPIF by optimizing out the implementation of the byte-steering logic.

An space select (enable) signal is generated for each range, rather than each word in the address space supported by the peripheral. (Note that this is different from the case of software addressable registers where an enable signal is generated for each register.)

Configure the IPIC

Typically the IPIC ports generated by this tool is dependent on the selections you make in the Select IPIF Services panel. However, some expert users may want access to other IPIC ports. You can check off these special ports in this panel.

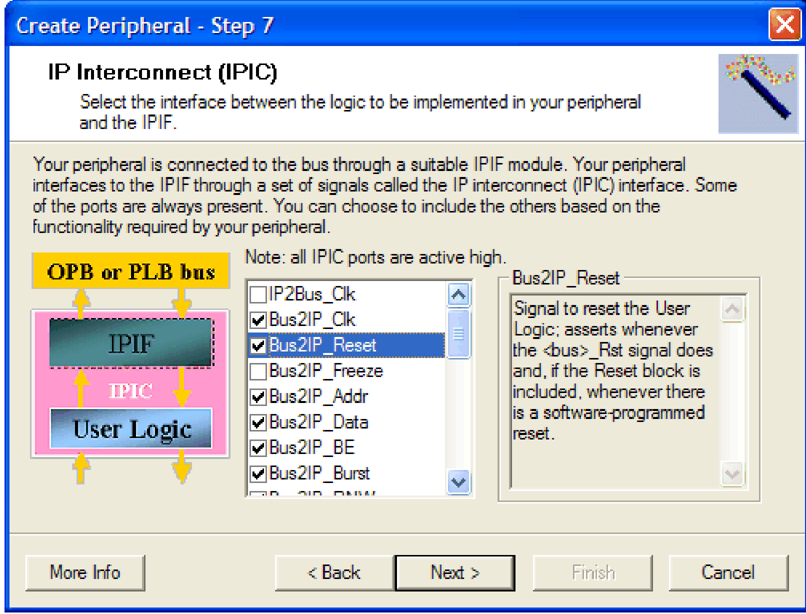


Figure 4-15: **Configure the IPIC**

Some of the IPIC ports in this panel are already selected and cannot be deselected. These ports are required to implement the functionality indicated in the Select IPIF Services panel.

Generate Optional Files

Besides the HDL template files, this tool will also generate some design support files to help your simulate and implement your custom logic and functionality. These are optional, but we highly recommend that you let the tool to generate these files for and it will significantly save your design effort.

The following optional files may be output by the tool:

- BFM simulation files

  This tool will create a BFM simulation platform for you to help you start sub-system simulation using the IBM Bus Functional Model incorporated in the EDK. It also comes with some sample bus transactions described in a BFL script file to verify various features of the template.

- ISE/XST project files

  An ISE project file may be created if you will implement your design in Project Navigator, or an XST project and synthesis script file for you to synthesis your design using XST directly.

- Software driver template file

Software driver template files and driver directory structure will be created to help you implement driver for your peripheral, some simple read/write macros and a default interrupt handler may be provided for your reference.

Peripheral Simulation Support

We recommend you to let the wizard generate BFM simulation files for you, as this will significantly save your time if you're using BFM simulation to verify your design.

The Bus Functional Model provides unit and system level simulation and verification of logic designs which comply with CoreConnect (OPB or PLB) architecture specifications. It enables you to accelerate the design cycle time by identifying and addressing possible problems at an earlier stage of the design cycle. To take the advantage, you must install the BFM toolkit in EDK and compile the ISE and EDK simulation libraries.

The wizard provides you with the link for instructions and downloading the BFM package for your EDK installation.

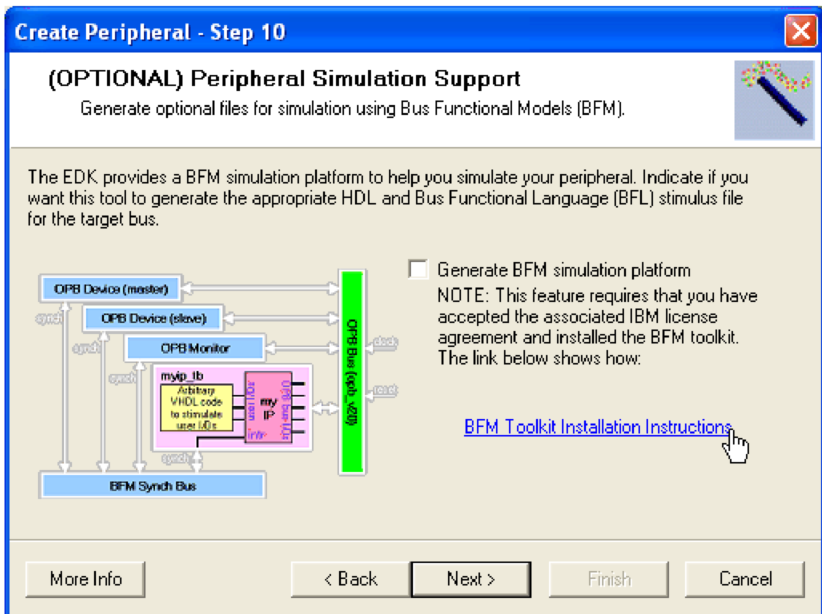


Figure 4-16: **Peripheral simulation support - not selected**

The wizard will build up a BFM simulation platform for you, including a system testbench, an IP testbench, sample bus transactions to access various IPIF features and makefiles. All setup working together to help you quickly launch your simulator and verify the waveform. A readme file will be available for instructions as how to start the BFM simulation after completing the wizard.

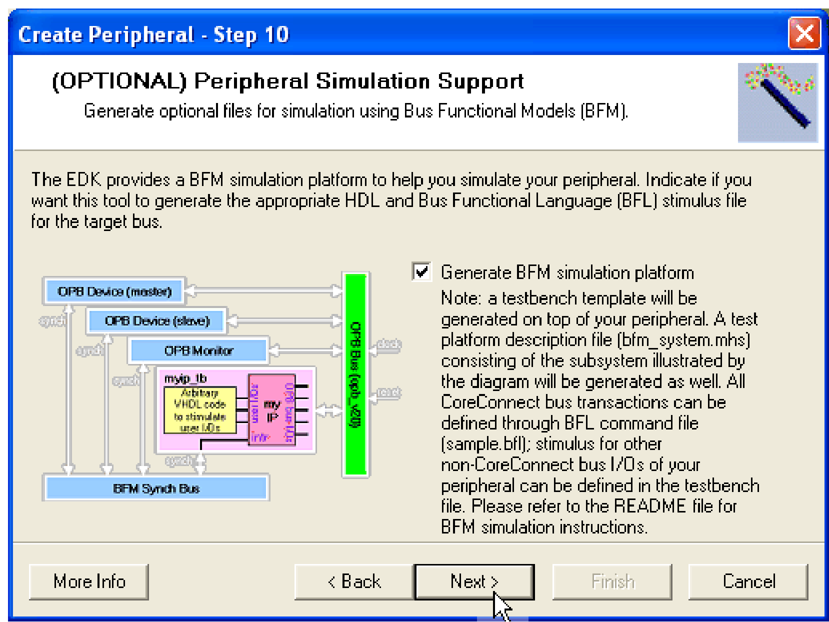


Figure 4-17: **Peripheral simulation support - selected**

Peripheral Implementation Support

We recommend you to let the wizard generate ISE/XST project files for you, as this will significantly save your time if you're using Xilinx flow to implement your design; the software driver template files are recommended if you're implementing driver for your peripheral as well.

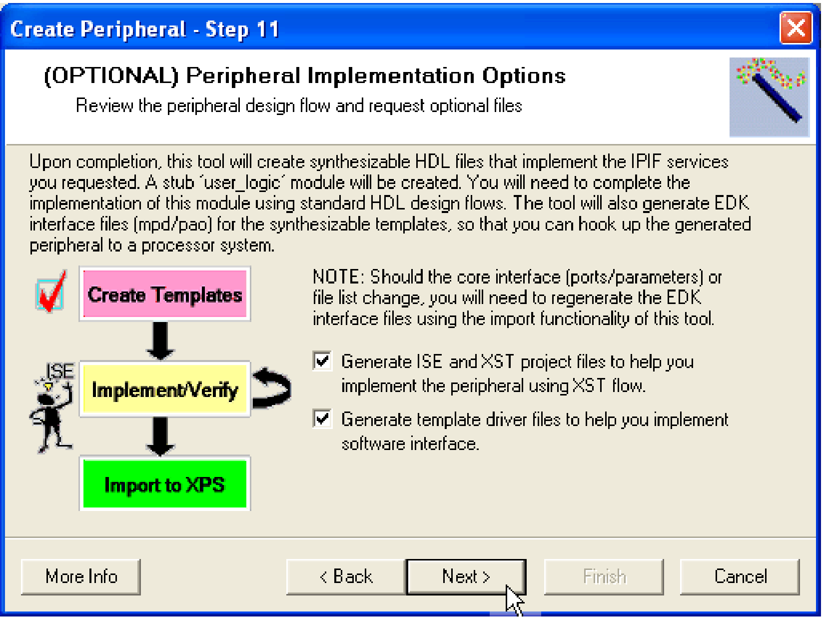


Figure 4-18: **Peripheral implementation support**

Generating the Files Representing Your Design

You are welcomed to review the summary information for the peripheral templates you requested, as well as a list of all the files that may be created by the tool.

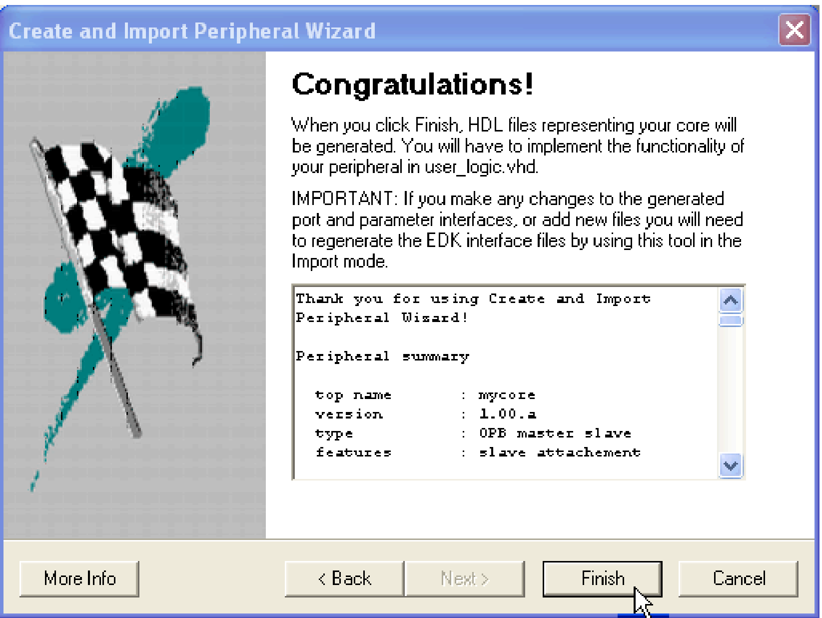


Figure 4-19: **Summary information**

Once all the required data has been collected from the user, this tool does the following:

- Create necessary directory structures.
- Creates HDL template files.
- Creates other files that help you complete the implementation of `user_logic.vhd`. These files include elements that help you design the peripheral using ISE, verify the peripheral using BFM simulation, implement the drivers, and other documentation files that help you write applications using this core.

If you already have any files in the target area, they will be overwritten.

Note that this tool is highly dependent on the port/parameter interface and the set of HDL files that comprise your peripherals. If these change during implementation, you will have to re-run this tool in the Import mode to regenerate the EDK interface files.

Review EDK Peripheral Design Flow

After completing this wizard, the following HDL files are created:

- *core_name*`.vhd`
- `user_logic.vhd`

Here *core_name*`.vhd` implements the 'top' module *core_name* of your peripheral. It instantiates the IPIF module from the built-in EDK cores library, and the `user_logic` module. The bus-interface ports appear on this module. Internally, these ports are wired to the IPIF module. The IPIF and `user_logic` modules are interconnected by the IPIC.

The `user_logic` module will usually have an empty implementation. In some cases a simple implementation may be included, *e.g.* if software addressable register support is

requested, the `user_logic.vhd` implements simple read/write to software addressable registers.

Generally, you will need to implement the `user_logic` module only. However, if your `user_logic` module is not self-contained, and needs more interface ports, you will have to add those to the *core_name* module in *core_name*`.vhd`. In such cases, just add the ports to the *core_name* module and pass them through to the `user_logic` module. Do not make any other changes to the *core_name*`.vhd` file.

Typically, you may need to run BFM simulation to verify your design that it contains the correct functionality and achieves the expected results.

Once you have completed the implementation of your peripheral, you need to import it into XPS using this tool in the import mode. This will generate the XPS interface files and run the HDL file set through a HDL parser to check for errors, *etc.*

It is very likely that you will implement `user_logic.vhd` using your favorite HDL based design flow. This will require you to understand the IPIC protocol. Please refer to the OPF_IPIF or PLB_IPIF chapter in the Processor IP document.

Once your `user_logic.vhd` is complete, you will want to put together a simple processor system to ensure that the software and hardware component of your system are interacting as expected. The software component of your system should implement the register reads and writes required to test out the interface. To do this, you will need to understand how to address the registers and interpret the data available there. These are documented in the IPIF section of the Processor IP Document. You should create a simple test system and implement and simulate that using the various flows available in the EDK.

Importing an Existing Peripheral

This tool can import an existing peripheral.Your peripheral must be written in Verilog or VHDL. It should also implement the Xilinx implementation of the CoreConnect bus conventions. This tool is easiest to use if you have followed the naming conventions for the ports and parameters. If not, it gives you the opportunity to establish the mapping of your ports and peripherals to the ports and peripherals in the Xilinx implementation of the CoreConnect bus conventions.

Generally, it is best to use this functionality in conjunction with the peripheral creation functionality described in the "Invoking the Wizard" section.

In this mode, this tool does the following:

- Query the user about the characteristics of the peripheral and the location of the HDL files that make up the peripheral. These include information about the CoreConnect Bus that the peripheral is expected to be connected to, whether it is a master and/or slave, the characteristics of the interrupts generated by the peripheral, etc.
- Copy out the HDL files into the XPS project or EDK repository using the rules for creating XPS and EDK repositories.
- Generate interface files like the Microprocessor Peripheral Data (MPD), Peripheral Analyze Order (PAO) and Black-box Data (BBD). These allow the tools in the EDK instantiate your peripheral in a system being designed using XPS.

It is very important that you follow certain conventions when you design your peripheral. The most important is the conventions used to name the top module and the logical library it is compiled into.

The subsequent sections explain the functionality offered by this tool, and what you can do with the files it generates.

Identifying the Physical Location of Your Peripheral

This functionality is identical to what is described under “Identifying the Physical Location of Your Peripheral” in the “Creating New Peripherals” section.

Identifying Module and Version

This is similar to the functionality described under “Identifying Module and Version” in the “Creating New Peripherals” section. The difference is that version control is optional in import mode, but it’s recommended to use version control.

You can either type your peripheral name or select the name through the drop down list from your previous run of this tool in Create Mode.

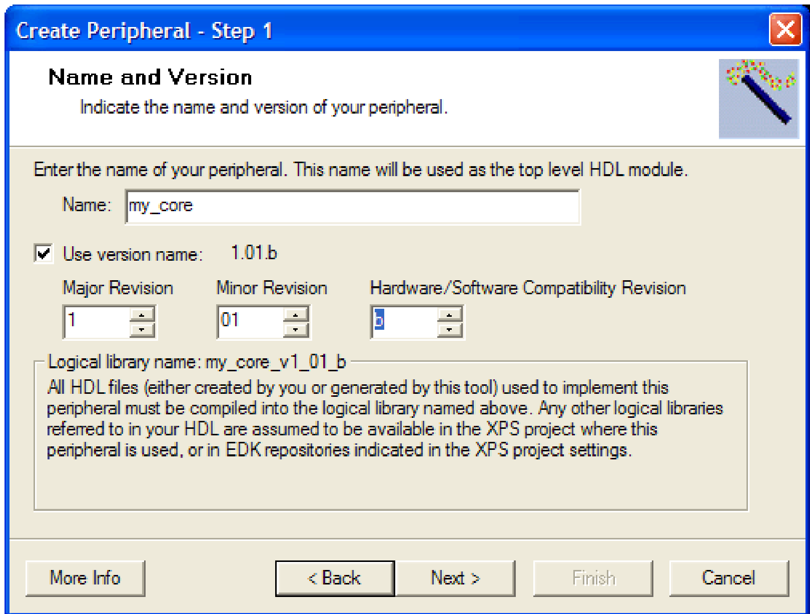


Figure 4-20: **Core Name and Version**

Select Source File Types

In this panel you indicate the kinds of files that make up your peripheral.

Presently, the system requires you to have at least one HDL file in VHDL or Verilog with the `.vhd` or `.v` extensions respectively.

Your peripheral may also instantiate black box netlists. These netlists may be EDIF, NGO, NGC or any of the netlist formats supported by the XILINX implementation tools. Typically, these have `.edn`, `.ngo`, or .ngc extensions.

If your core is a single fixed netlist, then you need to create a HDL wrapper that instantiates your netlist as a black-box.

Your core can also have documentation files in many of the common document formats PDF, TXT, etc., with `.pdf` or `.txt` extensions.

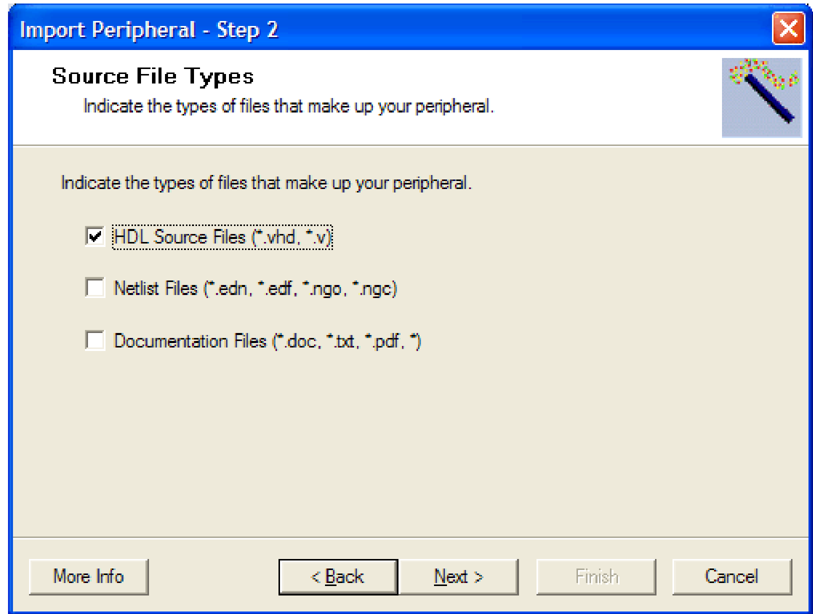


Figure 4-21: **Select Source File Types**

HDL Source Files

In this panel you help this tool locate your HDL source files. You also have to indicate whether your peripheral is in VHDL or Verilog.

You can choose to locate your HDL files by browsing to each file.But the preferred method is to browse to an XST project (`.prj`) file describing your core. This tool will try to determine the file list from the project file. This feature works well in most cases, but certain more complicated XST project files cannot be parsed accurately. So please verify the file list generated by this tool and modify as needed. Additionally, refer to the *XST User Guide* for XST project file syntax.

If the peripheral is already available in the directory structure required by the EDK, you can just browse to the `.pao` file. This tool will intuit the location of the source HDL files from the given `.pao` file.

Please ensure that filename does not have any spaces. Such path names are not supported at the present time.

The top-level HDL source file is expected to conform to the Xilinx implementation of the CoreConnect Bus Conventions. Please review OPB/PLB usage in Chapter 1 and 2 of the *Processor IP User Guide* found in the `doc` directory in the install.

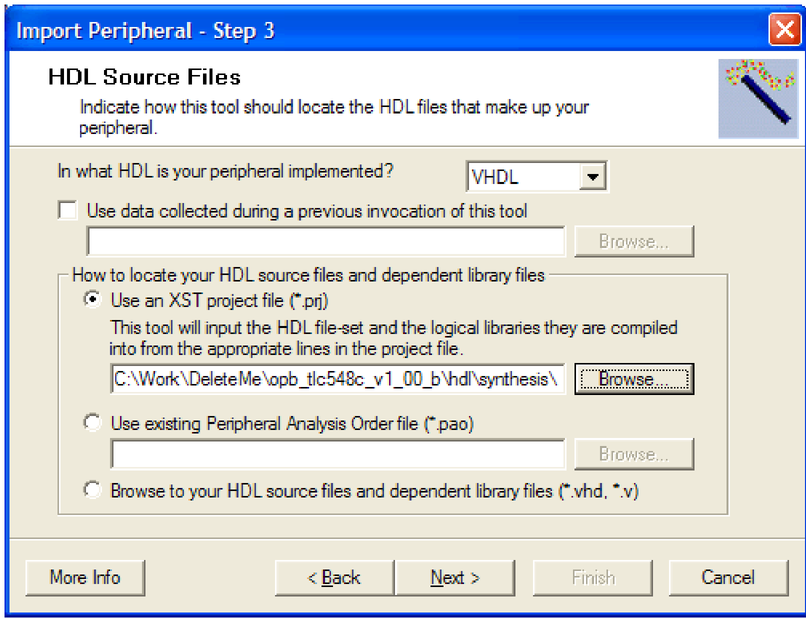


Figure 4-22: **Choose HDL Source Files**

HDL Analysis Information

In this panel you indicate compile order of your HDL files and the logical libraries they are compiled into.

If you had chosen to select your HDL source files by parsing the XST project file, then this panel would contain the list of files and the logical libraries they are compiled into. You are not allowed to modify the file-names and ordering if the given XST project file contains the '`nosort`' keyword.

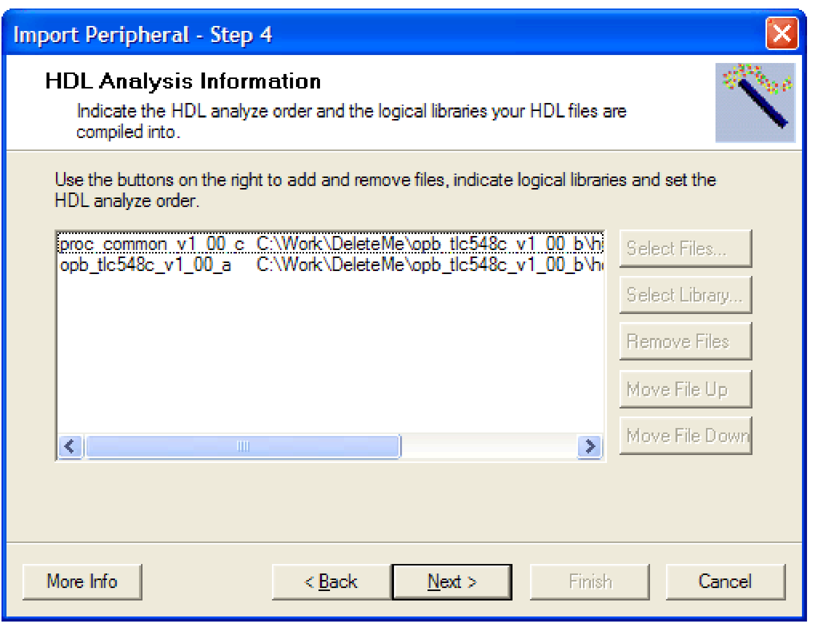


Figure 4-23: **Intuiting HDL Analysis Information from XST Project Files**

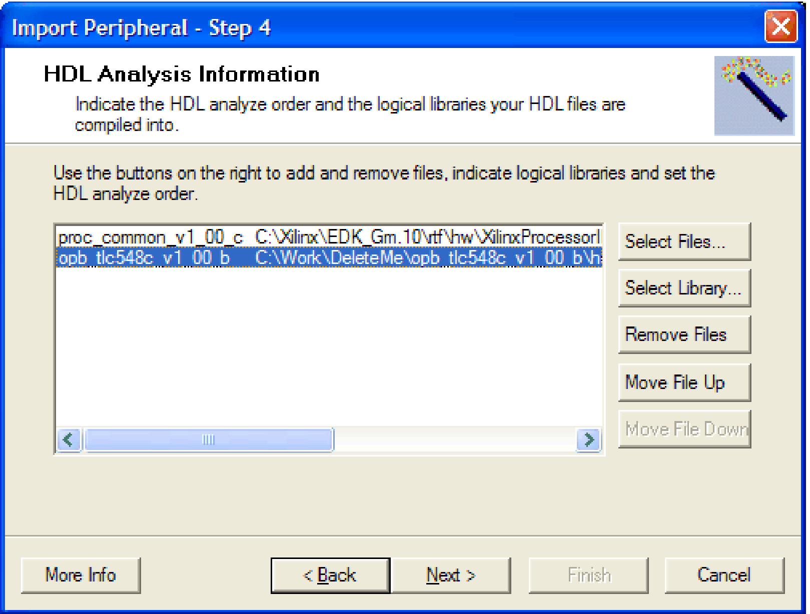


Figure 4-24: **Indicating HDL Analysis Information by Browsing to Files**

If you had chosen to select files by using the file browser, you can use the **Move File Up** and **Move File Down** buttons to change the compile order of the files.

Typically a selected file is assumed to be compiled into the logical library containing the current peripheral. This was explained in the "Identifying Module and Version" section.

If you need to include files from some other peripheral, then that peripheral must be available in the repositories known by XPS, or has been previously added to the current project.

When you click on the **Select Library** button, the libraries available in the repositories known to the current XPS project are displayed in a Library Selection Panel. When you select a library, the files available in the library are displayed. All files in the selected library are selected by default, but you can deselect the files that you don't care about by unckecking the check box next to the file.

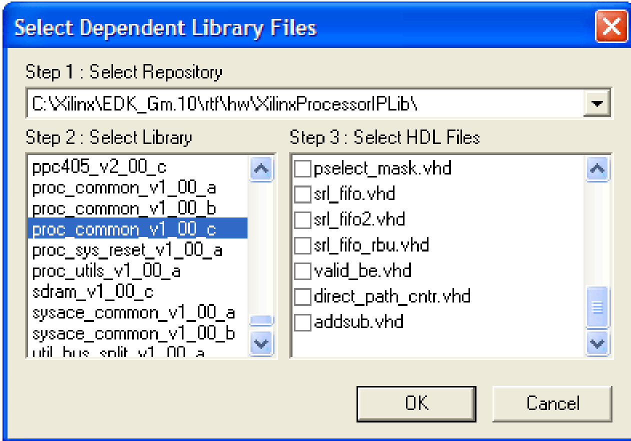


Figure 4-25: **Selecting Files from Other Libraries**

After you exit the **Select Library** panel, you are returned back to the **HDL Analysis Information** panel where the newly selected files are displayed.

Bus Interfaces

In this panel you indicate the types of bus interfaces that your peripheral supports.

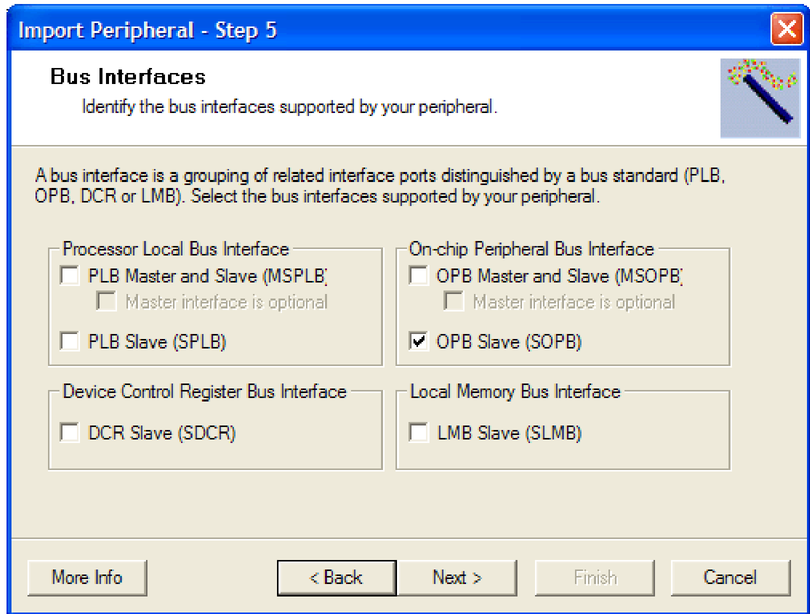


Figure 4-26: **Selecting Bus Interfaces**

The choices are as follows:

Table 4-4: **Supported bus interfaces**

Bus Interface		**Description**
MSPLB	Master-slave Processor Local Bus	This is a fast/wide bus that interacts directly with the processor. Most user peripherals are unlikely to support this bus interface.
SPLB	Slave Processor Local Bus	Select this interface if your peripheral operates as a slave on the processor local bus.
MSOPB	Master-Slave On-chip Peripheral Bus	Most user peripherals connect to the On-Chip Peripheral Bus (OPB.) Select this interface if your peripheral is a master and a slave.
SOPB	Slave On-chip Peripheral Bus	Select this interface if your peripheral operates as a slave on the OPB.
SDCR	Slave Direct Connect Register Bus	Select this if your peripheral operates as a slave on the Direct Control register bus (DCR.)
SLMB	Slave Local Memory Bus	Select this if your peripheral operates as a slave off the local memory bus (LMB.) Typically, this is applicable to systems that use the MicroBlaze 'soft-core' processor.

Note that master-only interfaces are not supported. Such interfaces are uncommon.

Identifying Bus Interface Ports and Parameters

A peripheral that implements a particular bus interface needs to the have the ports required by that interface. The ports do not have to have specific names, but it is best if the port are named exactly as specified in the specification of that interface. When the ports are named as the convention requires, this tool will correctly identify the bus interface ports.

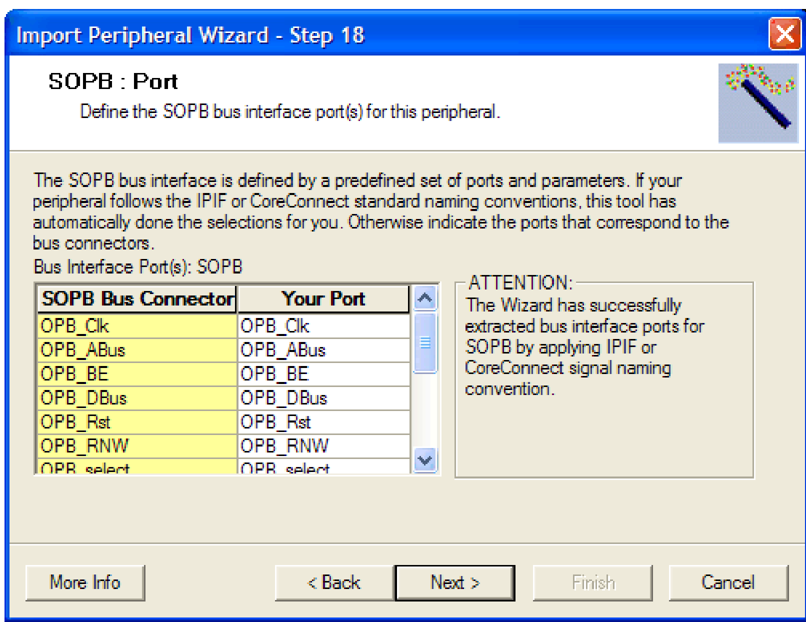


Figure 4-27: **Identifying Bus Interface Ports**

If this tool is unable to identify all the ports, the user will have to manually identify the bus interface ports. All bus interface ports must be identified before this tool will actually import any peripheral.

Similarly, some of the bus interfaces require associated parameters. Again, these are automatically identified if the parameters are named according to the interface convention. Otherwise the user will have to identify the required parameters.

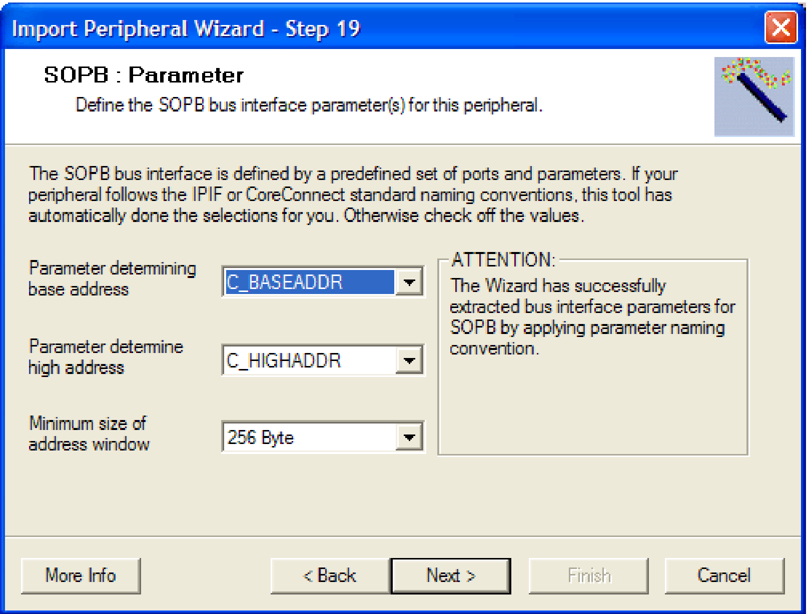


Figure 4-28: **Identifying Bus Interface Parameters**

For identifying the bus interface ports, the user is presented with a two column table. The left column lists the required bus interface ports. The cells to the right of each bus interface port have drop-down lists that list the ports on the peripheral being imported. The user needs to select the peripheral port which corresponds to each bus-interface port.

Interrupt Signals

Each peripheral needs to identify its interrupt signals and certain special attributes associated with the interrupt. These interrupts are processed by the interrupt controller in the processor system.

This panel presents a one column table that lists the non-bus interface ports on the peripheral. You check off the interrupt ports.

You also need to describe the characteristics of the selected interrupt signal. You do this by clicking on the radio buttons to the right. The various characteristics are as follows:

- Interrupt sensitivity

  The interrupt signal may be falling/rising edge sensitive, or low/high level sensitive.

- Relative priority

You can choose between Low, Medium or High. This information is used by some of the EDK tools to automatically prioritize the many interrupt generators in the system a peripheral is instantiated in.

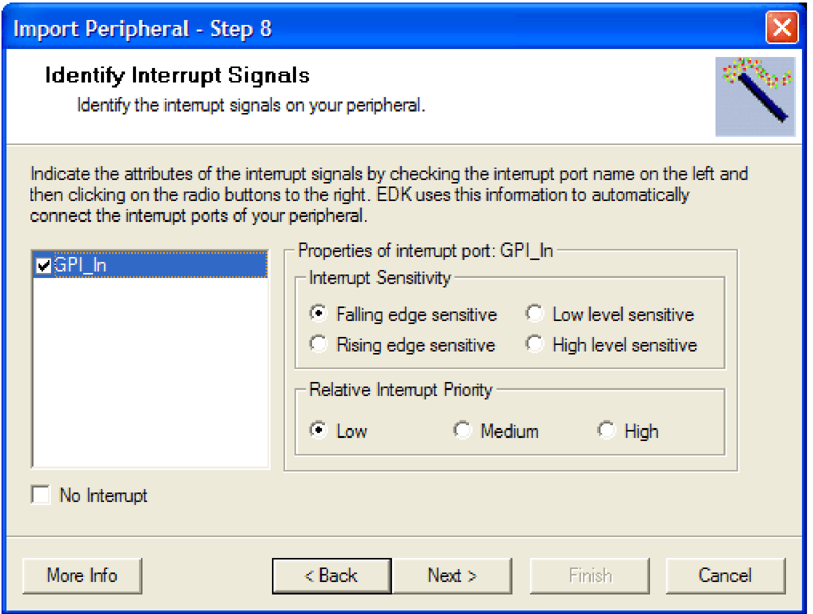


Figure 4-29: **Identifying Interrupt Signals**

If you do not have interrupts, check the **No Interrupts** check-box. Otherwise you cannot move to the next panel.

Advanced Attributes on Ports and Parameters

The Platform Specification Format (PSF) in the EDK supports a large number of attributes on ports and parameters. These attributes help the tools in the EDK automatically wire up the peripheral to the bus, connect the interrupt lines, display more readable names, provide short descriptions of port and parameter functionality, *etc.*

This tool will present screens that allow you to input the values of the attributes through a table based interface. You will see two tables:

- The one-column table on the left lists the ports identified by this tool. A drop-down list on the top of the table allows you to list bus interface ports only, or user (non-bus interface) ports only, or list all ports. The structure is very similar for the parameters.
- The table to the right has two columns. The column on the left lists the attributes and the one on the right displays the values of the corresponding attributes. We will refer to this as the *Attributes Table*. The attribute names displayed are descriptive names for the corresponding MPD keywords.

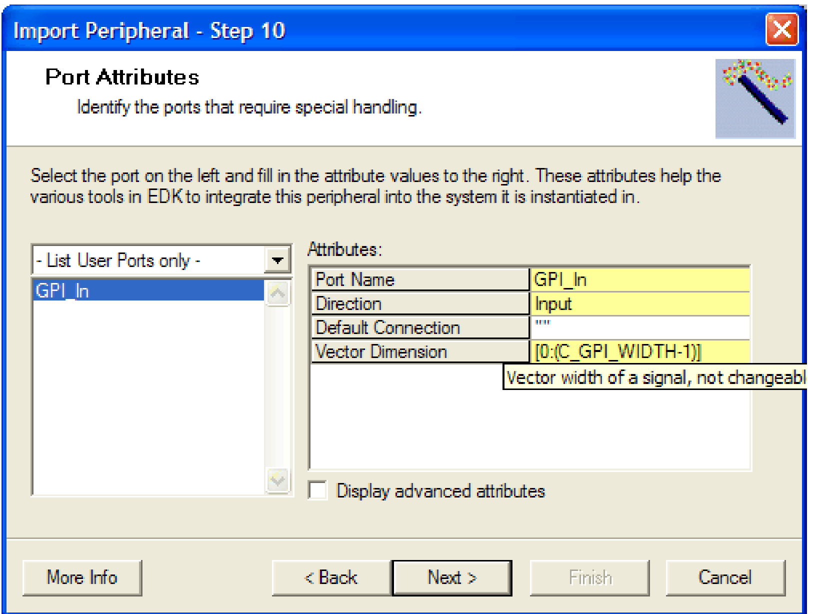


Figure 4-30: **Setting Attributes on Ports**

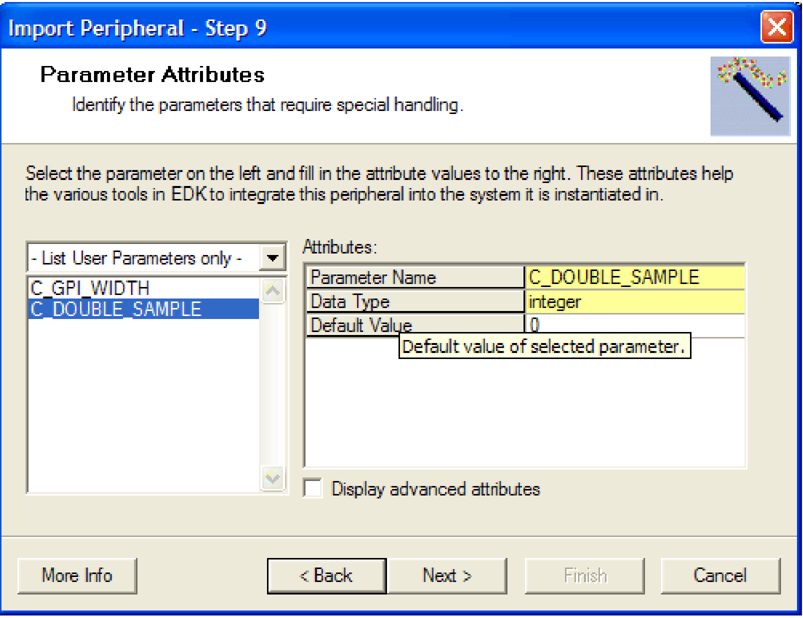


Figure 4-31: **Setting Parameters**

When you select one of the parameters or ports on the table to the left, the Attributes Table to the right gets filled in with the attribute names and values.

A **Display Advanced Attributes** check box controls the display of non-essential attributes. The advanced attributes are not displayed by default.

The value cells in the *Attributes Table* are color coded. A yellow cell contains data intuited from the `ENTITY` or `module` representing your peripheral. A green cell represents data intuited from inputs from some of the preceding screens. All other cells are editable.

If you position the cursor on one of the attributes in the left column of the *Attributes Table*, a short description of the attribute will appear. This description will usually contain the MPD keyword for this parameter.

Netlist Files

Your peripheral can be HDL with fixed netlists instantiated as black-boxes. In this panel you locate the netlist files associated with your peripheral. This selection is done by browsing to the directory containing the file.

This tool does not allow you to associate different netlist files with different parameter values for your peripheral. Also, you must have at least one HDL file associated with your core. This could be the HDL file that just instantiates a black-box netlist.These files can be in any of the common formats, *e.g.* NGC/NGO (`.ngc` and `.ngo`) or EDIF (`.edn` or `.edf`).

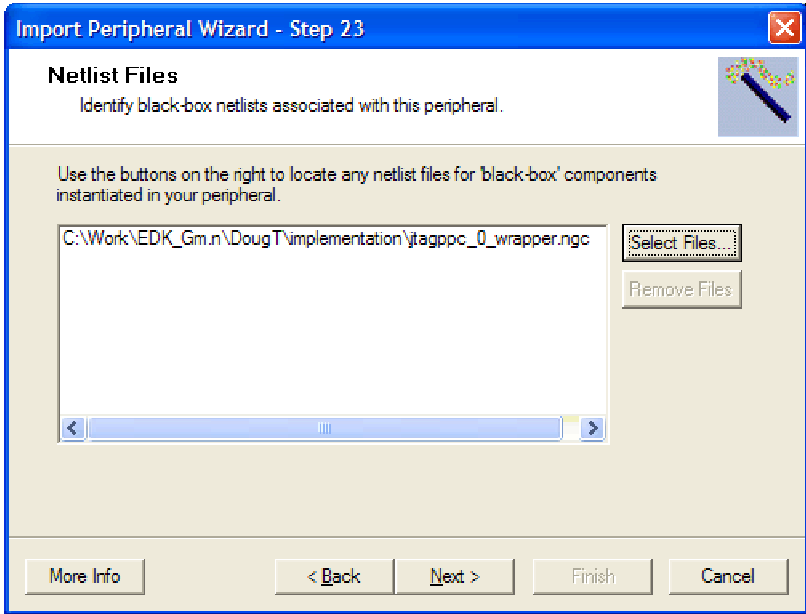


Figure 4-32: **Selecting Netlist Files**

Documentation Files

Documentation files are selected by browsing to the file. These files can be in any of the common formats, *e.g.* PDF (`.pdf`) or TEXT (`.txt`).

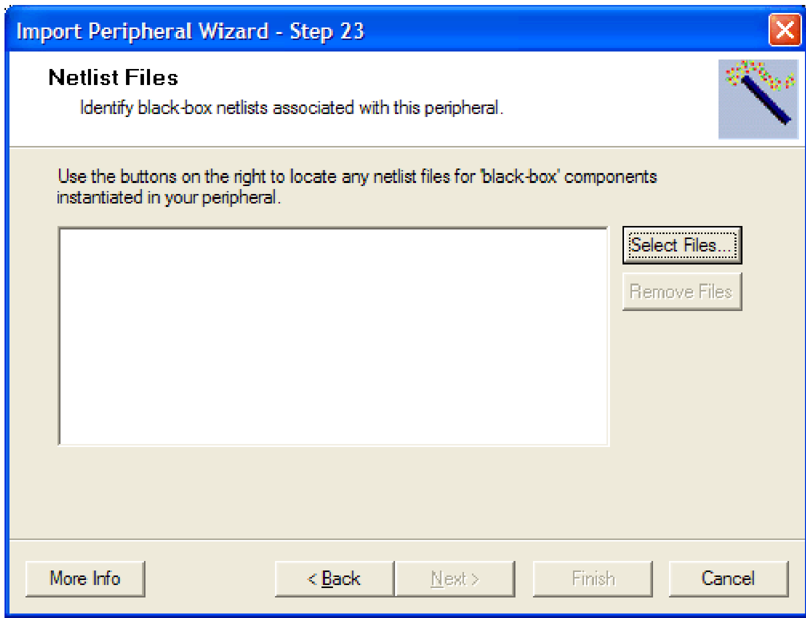


Figure 4-33: **Selecting Documentation Files**

Finishing Peripheral Import

Once all the required data has been collected from the user, this tool does the following:

- Copy over the user HDL, netlist and documentation files into the XPS project into a directory structure determined by the PSF specification. If the peripheral was being outputted into a XPS project, the core is outputted in a directory named `pcores` located in the project directory. If the target was a XPS repository directory, then the core is outputted under `MyProcessorIPLib/pcores` under the repository directory.
- Generate the interface files required by the various tools in the EDK. These include the MPD, PAO, BBD files.

If you already have any files in the target area, they would be backed up unless you instruct otherwise.

Note that your source HDL, netlist and documentation files are getting copied over. If you make in any changes you may have to run this tool again. Additionally, the output of this tool is highly dependent on the port/parameter interface and the HDL analyze order. If any of these change you may want to re-run this tool.

Organization of Generated Files

This tool generates files based on user input. Table 4-5 describes what files are generated and how they are used.

Table 4-5: **Files and Directories Generated by the Create/Import IP Wizard**

Directory or file	Description
<*pcores-directory*>	This one of the following: <*EDK-Repository-Dir*>`/MyProcessorIPLib/pcores` or <*Directory-containing-XPS-Project-File*>`/pcores` See section "Identifying the Physical Location of Your Peripheral" for how this is specified.
<*logical-library-name*>	This is the logical library name as defined in section "Identifying Module and Version"
<*peripheral-name*>	This is the peripheral name as defined in section "Identifying Module and Version"
<*peripheral-version*>	This is the peripheral version as defined in section "Identifying Module and Version"
<*peripheral-directory*>	<*pcores-directory*>/<*logical-library-name*>
<*devl*>	<*peripheral-directory*>`/devl` This is a directory containing collateral to help user develop the user-logic component of the core.
<*devl*>`/README.txt`	File explaining the output generated by this tool. We recommend that the user go through this file. It has a lot of documentation about exactly what the user needs to do to complete the implementation of the user-logic part of the core.
<*devl*>`/ipwiz.log`	File containing a list of messages outputted by this tool.
<*devl*>`/ipwiz.opt`	File capturing the data inputted by the user in the wizard GUI. Presently, the user does not need to use this file for any purpose.
<*projnav-dir*>	<*devl*>`/projnav` This is a directory containing a Project Navigator project file. This directory will contain files used by Project Navigator if you choose to develop the user-logic part of the peripheral using Project Navigator.
<*projnav-dir*>/<*peripheral-name*>`.npl`	Project Navigator project file you can open to complete the development of the peripheral using Project Navigator.
<*projnav-dir*>/<*peripheral-name*>`.cli`	Not presently used for any purpose.

Table 4-5: **Files and Directories Generated by the Create/Import IP Wizard**

Directory or file	Description
<*synthesis-dir*>	<*devl*>`/synthesis` This is a directory containing files that will help you synthesize the peripheral using XST.
<*synthesis-dir*>/<*peripheral-name*>`_xst.prj`	XST project file. In case you add more files HDL to your peripheral, you need to add them to this file.
<*synthesis-dir*>/<*peripheral-name*>`_xst.scr`	A simple XST script file that uses the XST project file and can be passed to XST to generate the netlist representing the peripheral.
<*simulation-dir*>	<*devl*>`/bfmsim` This is a directory containing files that will help you simulate the design using BFM.
<*simulation-dir*>/README.txt	README file that contains instructions on how to start BFM simulation.
<*simulation-dir*>/bfm_sim_cmd.make	Makefile for command line usage only, contains targets for BFM simulation platform compilation and launching simulator.
<*simulation-dir*>/bfm_sim_xps.make	Makefile for XPS usage only, contains targets for BFM simulation platform compilation and launching simulator.
<*simulation-dir*>/bfm_system.mhs	BFM simulation system testbench description file, input to SimGen for behavioural simulation generation.
<*simulation-dir*>/bfm_system.mss	Empty system driver file to work around XPS warning.
<*simulation-dir*>/bfm_system.xmp	XPS project file for BFM simulation only.
<*simulation-dir*>/scripts/sample.bfl	CoreConnect bus transactions described in Bus Functional Language, input to Bus Functional Compiler to generate simulator commands for simulation.
<*simulation-dir*>/scripts/wave.do	Signal dataset file for viewing all interested signals in waveform window.
<*simulation-dir*>/scripts/run.do	Top level simulator script file, contains commands to compile, load modules and start simulation.
<*ip-testbench-name*>	<*peripheral-name*>_tb
<*ip-testbench-dir*>	<*simulation-dir*>`/pcores/`<*ip-testbench-name*>_<*peripheral-version*> This is a directory containing the IP testbench file for the peripheral.
<*ip-testbench-dir*>/hdl/vhdl/<*ip-testbench-name*>.vhd	IP testbench file, defines processes to test the peripheral under test, provides constant interface to the system testbench and mechanism to communicate with Bus Functional Language commands for synchronization.

Table 4-5: **Files and Directories Generated by the Create/Import IP Wizard**

Directory or file	Description
<*ip-testbench-dir*>/data/	Directory containing EDK interface files (MPD & PAO) for the IP testbench.
<*peripheral-directory*>`/data`	Directory containing EDK interface files (MPD & PAO) for the core.
<*peripheral-directory*>`/hdl/vhdl`	Directory containing generated (or imported) VHDL files representing the core. In case you need more VHDL files to represent your peripheral, you can add them here.
<*peripheral-directory*>`/hdl/verilog`	Directory containing generated (or imported) Verilog files representing the core. In case you need more VHDL files to represent your peripheral, you can add them here.

Limitations

This tool has a number of limitations

Create Peripheral Mode

- Verilog peripherals are not supported.
- Only simple mode DMA is supported in this release.
- Only ModelSim BFM simulation is supported in this release.
- Master-only templates are not supported, as resource-wide the master-only interface doesn't save anything for you compared to master-slave combined templates with minimal slave functionality.

Import Peripheral Mode

- Master-only bus interfaces are not supported. Such peripherals are rare.
- References to fixed netlists cannot be parameterized. This implies that you cannot create a peripheral that is just a set of fixed netlists and no associated HDL. Typically, such peripherals are supported by BBD files only with no associated PAO file.
- XPS repository or projects with spaces in the pathname are not supported.

Chapter 5

Platform Generator

The hardware component is defined by the Microprocessor Hardware Specification (MHS) file. An MHS file defines the configuration of the embedded processor system, and includes the following:

- Bus architecture
- Peripherals
- Connectivity of the system
- Interrupt request priorities
- Address space

Hardware generation is done with the Platform Generator (PlatGen) tool and an MHS file. This will construct the embedded processor system in the form of hardware netlists (HDL and implementation netlist files).

This chapter contains the following sections:

- "Tool Requirements"
- "Tool Usage"
- "Tool Options"
- "Load Path"
- "Output Files"
- "About Memory Generation"
- "Reserved MHS Parameters"
- "Synthesis Netlist Cache"
- "Current Limitations"

Tool Requirements

Set up your system to use the Xilinx Development System. Verify that your system is properly configured. Consult the release notes and installation notes that came with your software package for more information.

Tool Usage

Run PlatGen as follows:

```
platgen -p virtex2p system.mhs
```

Tool Options

The following are the options supported in the current version:

-h (Help)

The -**h** option displays the usage menu and quits.

-v (Display version)

The -**v** option displays the version and quits.

-f <*filename*>

Read command line arguments and options from file.

-iobuf yes | **no**

IOB insertion at the top-level. The default is yes.

This option is deprecated. Please use the '-toplevel' option.

-lang verilog | **vhdl**

HDL language output. The default is vhdl.

-log <*logfile*[**.log**]>

Specify log file. The default is `platgen.log`. Currently, not implemented.

-lp <*library_path*>

Add <*library_path*> to the list of IP search directories. A library is a collection of repository areas.

-od <*output_dir*>

Output directory path. The default is the current directory.

-p <*partname*>

Use specified part type to implement the design.

-st xst | **none**

Generate synthesis project files. The default is xst.

PlatGen produces a synthesis vendor specific project file.

-ti <*instname*>

Top-level instance name.

-tm <*top_module*>

Name top-level module as desired.

-tn <*compname*>

Top-level entity/module name.

This option is deprecated. Please use the '-**tm**' option.

-toplevel yes | **no**

Input design represents a whole design or a level of hierarchy. Default is **yes**.

Load Path

Refer to Figure 5-1 for a depiction of the peripheral directory structure.

To specify additional directories, use one of the following options:

- Current directory (where PlatGen was launched; not where the MHS resides)
- Set the EDK tool option -**lp** option

PlatGen uses a search priority mechanism to locate peripherals, as follows:

1. Search the pcores directory in the project directory
2. Search <library_path>/<Library Name>/pcores as specified by the -**lp** option
3. Search XILINX_EDK/hw/<Library Name>/pcores

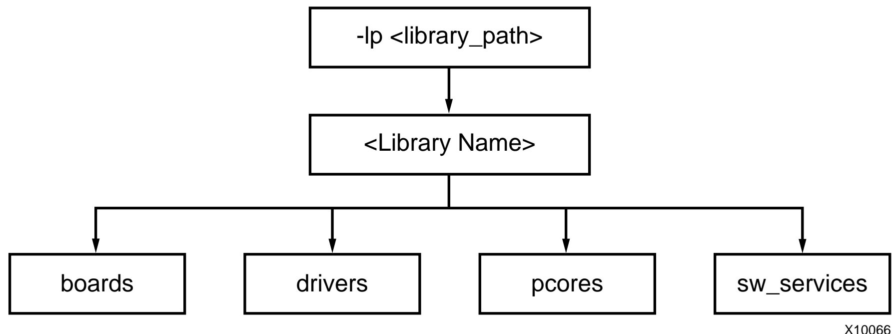


Figure 5-1: **Peripheral Directory Structure**

From the pcores directory, the peripheral name is the name of the root directory. From the root directory, the underlying directory structure is as follows:

```
data
hdl
netlist
```

Output Files

PlatGen produces the following directories and files. From the project directory, this is the underlying directory structure:

```
hdl
implementation
synthesis
```

HDL Directory

The hdl directory contains the following:

system.[vhd|v]

This is the HDL file of the embedded processor system as defined in the MHS. This file contains IOB primitives if the -**toplevel yes** option is specified.

*system*_stub.[vhd|v]

This is the toplevel template HDL file of the instantiation of the system and IOB primitives. Use this file as a starting point for your own toplevel HDL file. This file is

generated when the **-toplevel no** option is specified. Otherwise, the *system*.[vhd|v] file is the toplevel.

<*inst*>_wrapper.[vhd|v]

This is the HDL wrapper file for the of individual IP components defined in the MHS.

Implementation Directory

The implementation directory contains the following:

*peripheral*_wrapper.ngc

Implementation netlist file of the peripheral.

Synthesis Directory

The synthesis directory contains the following:

system.[prj|scr]

Synthesis project file.

About Memory Generation

PlatGen generates the necessary banks of memory and the initialization files for the BRAM Block (bram_block). The BRAM Block is coupled with a BRAM controller.

Current BRAM controllers include the following:

- DSOCM BRAM Controller (dsbram_if_cntlr) - PowerPC only
- ISOCM BRAM Controller (isbram_if_cntlr) - PowerPC only
- LMB BRAM Controller (lmb_bram_if_cntlr) - MicroBlaze only
- OPB BRAM Controller (opb_bram_if_cntlr)
- PLB BRAM Controller (plb_bram_if_cntlr)

The BRAM block (bram_block) and one of the BRAM controllers are tightly bound. Meaning that the associated options of the BRAM controller define the resulting BRAM block. Theses options are listed in every BRAM controller MPD file. For example, the OPB BRAM controller MPD defines the following:

```
OPTION NUM_WRITE_ENABLES = 4
OPTION ADDR_SLICE = 29
OPTION DWIDTH = 32
OPTION AWIDTH = 32
```

The definition of AWIDTH and DWIDTH is applied to C_AWIDTH and C_DWIDTH of the BRAM block, respectively. The port dimensions on ports A and B are symmetrical on the bram_block. PlatGen overwrites all user-defined settings on the BRAM block to have uniform port widths.

You can only connect BRAM controllers of the same options values to the same BRAM block instance. For example, you can connect a OPB BRAM controller and LMB BRAM controller to the same BRAM block. However, you can not connect a OPB BRAM controller and a PLB BRAM controller to the same BRAM block instance. You can connect a LMB BRAM controller and a DSOCM BRAM controller to the same BRAM block instance.

The BRAM controller's MHS options, C_BASEADDR and C_HIGHADDR (see Chapter 15, "Microprocessor Hardware Specification (MHS)," in the *Platform Specification Format Reference Manual* for more information), define the different depth sizes of memory.

The MicroBlaze processor is a 32-bit machine, therefore, has data and instruction bus widths of 32-bit. Only predefined memory sizes are allowed. Otherwise, MUX stages have to be introduced to build bigger memories, thus slowing memory access to the memory banks. For Spartan-II, the maximum allowed memory size is 4 kBytes which uses 8 Select BlockRAM. For Spartan-IIE, the maximum allowed memory size is 8 kBytes which uses 16 Select BlockRAM. For Virtex/VirtexE, the maximum allowed memory size is 16 kBytes which uses 32 Select BlockRAM. For Virtex-II, it is 64 kBytes which also uses 32 Select BlockRAMs.

Table 5-1: **Predefined Memory Sizes**

Architecture	Memory Size (kBytes) 32-bit byte-write	Memory Size (kBytes) 64-bit byte-write
Spartan-II	2, 4	4,
Spartan-IIE	2, 4, 8, 16	4, 8, 16, 32
Spartan-3	8, 16, 32, 64	16, 32, 64, 128
Virtex	2, 4, 8, 16	4, 8, 16, 32
VirtexE	2, 4, 8, 16	4, 8, 16, 32
Virtex-II	8, 16, 32, 64	16, 32, 64, 128
Virtex-II PRO	8, 16, 32, 64	16, 32, 64, 128
Virtex-4	2, 4, 8, 16, 32, 64, 128	4, 8, 16, 32, 64, 128, 256

Be sure to check your FPGA resources can adequately accommodate your executable image. For example, the smallest Spartan-II device, xc2s15, only 4 Select BlockRAMs are available for a maximum memory size of 2 kBytes. Whereas, the largest Spartan-II device, xc2s200, 14 Select BlockRAMs are available for a maximum memory size of 7 kBytes.

For example, for a memory size of 4 kBytes on a Virtex device, PlatGen uses 8 Select BlockRAMs.

BMM Policy

A BMM (BlockRAM Memory Map) file contains a syntactic description of how individual BlockRAMs constitute a contiguous logical data space. PlatGen has the following policy for writing a BMM file:

- If PORTA is connected and PORTB is not connected, then the BMM generated will be from PORTA point of reference.
- If PORTA is not connected and PORTB is connected, then the BMM generated will be from PORTB point of reference.
- If PORTA is connected and PORTB is connected, then the BMM generated will be from PORTA point of reference.

BMM Flow

The EDK tools Implementation Tools flow using Data2MEM.

```
ngdbuild -bm <system>.bmm <system>.ngc
map
par
bitgen -bd <system>.elf
```

BitGen outputs <system>_bd.bmm that contains the physical location of BlockRAMs. The <system>_bd.bmm and <system>.bit files are input to Data2MEM. Data2MEM translates contiguous fragments of data into the proper initialization records for Virtex series BlockRAMs.

Reserved MHS Parameters

PlatGen automatically expands and populates certain reserved parameters. This can help prevent errors when your peripheral requires information on the platform that is generated. The following table lists the reserved parameter names:

Table 5-2: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_FAMILY	FPGA Device Family
C_INSTANCE	Instance name of component
C_KIND_OF_EDGE	Vector of edge sensitive (rising/falling) of interrupt signals
C_KIND_OF_LVL	Vector of level sensitive (high/low) of interrupt signals
C_KIND_OF_INTR	Vector of interrupt signal sensitivity (edge/level)
C_NUM_INTR_INPUTS	Number of interrupt signals
C_MASK	LMB Decode Mask (deprecated)
C_NUM_MASTERS	Number of OPB masters (deprecated)
C_NUM_SLAVES	Number of OPB slaves (deprecated)
C_DCR_AWIDTH	DCR Address width
C_DCR_DWIDTH	DCR Data width
C_DCR_NUM_SLAVES	Number of DCR slaves
C_LMB_AWIDTH	LMB Address width
C_LMB_DWIDTH	LMB Data width
C_LMB_MASK	LMB Decode Mask
C_LMB_NUM_SLAVES	Number of LMB slaves
C_OPB_AWIDTH	OPB Address width
C_OPB_DWIDTH	OPB Data width
C_OPB_NUM_MASTERS	Number of OPB masters

Table 5-2: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_OPB_NUM_SLAVES	Number of OPB slaves
C_PLB_AWIDTH	PLB Address width
C_PLB_DWIDTH	PLB Data width
C_PLB_MID_WIDTH	PLB master ID width
C_PLB_NUM_MASTERS	Number of PLB masters
C_PLB_NUM_SLAVES	Number of PLB slaves

Synthesis Netlist Cache

An IP rebuild occurs with one of the following fundamental changes:

- Instance name change
- Parameter value change
- Core version change
- Core is specified with the MPD “CORE_STATE=DEVELOPMENT” option

At least one of the above conditions is occurring to trigger an IP rebuild.

Current Limitations

The current limitations of the PlatGen flow are:

- Vector slicing is not allowed.

Simulation Model Generator

This chapter introduces the basics of HDL simulation and describes the Simulation Model Generator tool and COMPEDKLIB utility tool usage. It contains the following sections.

- "Overview"
- "Simulation Basics"
- "Simulation Libraries"
- "COMPEDKLIB Utility Tool"
- "Simulation Models"
- "SimGen Syntax"
- "Output Files"
- "Memory Initialization"
- "Simulating Your Design"
- "Current Limitations"

Overview

The Simulation Model Generator (SimGen) creates and configures various VHDL and Verilog simulation models for a specified hardware. It takes a Microprocessor Hardware Specification (MHS) file as input which describes the instantiations and connections of hardware components.

SimGen is also capable of creating scripts for a specified vendor simulation tool. The scripts compile the generated simulation models.

The hardware component is defined by the Microprocessor Hardware Specification (MHS) file. Please refer to Chapter 2, "Microprocessor Hardware Specification (MHS)," in the *Platform Specification Format Reference Manual* for more information.

Simulation Basics

This section introduces the basic facts and terminology of HDL simulation in EDK. There are three stages in the FPGA design process in which you conduct verification through simulation. Figure 6-1 shows these stages.

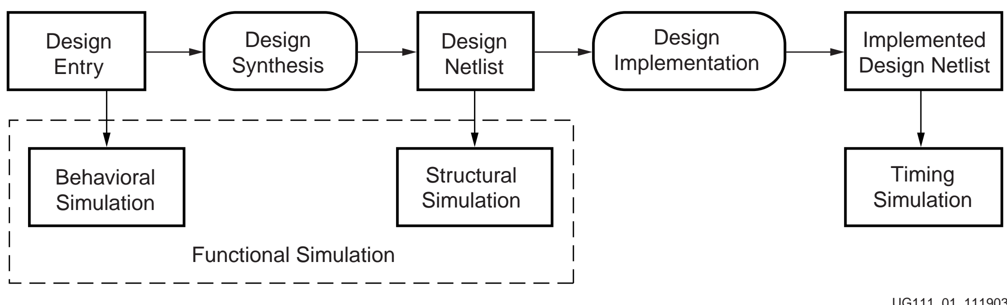


Figure 6-1: **FPGA Design Simulation Stages**

Behavioral Simulation

Behavioral simulation is used to verify the syntax and functionality without timing information. The majority of the design development is done through behavioral simulation until the required functionality is obtained. Errors identified early in the design cycle are inexpensive to fix compared to functional errors identified during silicon debug.

We refer to behavioral files to all pre-synthesis HDL files. These files may be written in a purely behavioral manner using behavioral constructs, such as operators, vhdl process or verilog always statements. These may also be written in a structural manner only doing instantiations of lower level components or they can be written in a mixed behavioral-and-structural manner.

Structural Simulation

After behavioral simulation is error free, the HDL design is synthesized to gates. The post-synthesized structural simulation is a functional simulation with no timing information. The simulation can be used to identify initialization issues and to analyze don't care conditions. The post synthesis simulation generally uses the same testbench as functional simulation.

The hdl files at this stage will not contain any behavioral constructs, such as operators, vhdl *process* or verilog *always* constructs.

Timing Simulation

Timing simulation is a structural back-annotated timing simulation. Timing simulation is important in verifying the operation of your circuit after the worst case place and route delays are calculated for your design. The back annotation process produces a netlist of library components annotated in an SDF file with the appropriate block and net delays from the place and route process. The simulation will identify any race conditions and setup-and-hold violations based on the operating conditions for the specified functionality.

Simulation Libraries

The following libraries are available for the Xilinx simulation flow.The HDL code must refer to the appropriate compiled library. The HDL simulator must map the logical library to the physical location of the compiled library.

Xilinx Libraries

The following libraries are provided by Xilinx for simulation. These libraries can be compiled using COMPXLIB. Please refer to Chapter 6, "Verifying Your Design" in the *Synthesis and Verification Design Guide* in your ISE 6.1 distribution to learn more about compiling and using Xilinx simulation libraries.

UNISIM Library

This is a library of functional models used for behavioral and structural simulation. It includes all of the Xilinx Unified Library components that are inferred by most popular synthesis tools. The UNISIM library also includes components that are commonly instantiated such as I/Os and memory cells.

You can instantiate the UNISIM library components in your design (VHDL or Verilog) and simulate them during behavioral simulation. Structural simulation models generated by SimGen will instantiate UNISIM library components.

All asynchronous components in the UNISIM library have zero delay. All synchronous components have a unit delay to avoid race conditions. The clock to out delay for these is 100 ps.

SIMPRIM Library

This is a library used for timing simulation. This library includes all of the Xilinx Primitives Library components that are used by Xilinx implementation tools.

Timing simulation models generated by SimGen will instantiate SIMPRIM library components.

XilinxCoreLib Library

The Xilinx CORE Generator is a graphical intellectual property design tool for creating high-level modules like FIR Filters, FIFOs, CAMs as well as other advanced IP. You can customize and pre-optimize modules to take advantage of the inherent architectural features of Xilinx FPGA devices, such as block multipliers, SRLs, fast carry logic and on-chip, single-port or dual-port RAM.

The CORE Generator HDL library models are used for behavioral simulation. You can select the appropriate HDL model to integrate into your HDL design. The models do not use library components for global signals.

EDK Library

Used for behavioral simulation. It contains all the EDK IP components, precompiled for ModelSim SE and PE or NcSim. This library eliminates the need to recompile EDK components on a per project basis, minimizing the time required to compile behavioral models on each project. EDK IP components library is provided for VHDL only.

The EDK library can be compiled with the COMPEDKLIB utility, which is described in the following section.

COMPEDKLIB Utility Tool

COMPEDKLIB is a utility tool provided by Xilinx® to compile the EDK HDL based simulation libraries using the tools provided by various simulator vendors.

Usage

```
compedklib [ -h ] [ -o output-dir-name ] [ -lp repository-dir-name ]
     [ -E compedklib-output-dir-name ] [ -lib core-name ]
     [ -compile_sublibs ] [ -exclude deprecated|obsolete ]
     -s mti_se|mti_pe|ncsim -X compxlib-output-dir-name
```

This tool compiles the HDL in EDK *pcore* libraries for simulation using the simulators supported by the EDK. Currently, the only supported simulator is MTI PE/SE and NCSIM.

COMPEDKLIB Command Line Examples

Use Case I: Compiling HDL Sources in the Built-In Repositories in the EDK

The most common use case is as follows:

```
compedklib -o <compedklib-output-dir-name>
     -X <compxlib-output-dir-name>
```

In this case the *pcores* available in the EDK install are compiled and the stored in <compedklib-output-dir-name> . The value to the '-X' option indicates the directory containing the models outputted by 'compxlib'. such as the 'unisim', 'simprim' and 'XilinxCoreLib' compiled libraries.

Pcores can be in development, active, deprecated and obsolete state. Adding a '-exclude obsolete' has the effect of not compiling obsolete cores. '-exclude deprecated' excludes deprecated & obsolete cores. Other settings are not valid as of this version.

Use Case II: Compiling HDL Sources in Your Own Repository

If you had your own repository of EDK style *pcores*, you may to compile them into <compedklib-output-dir-name> as follows:

```
compedklib -o <compedklib-output-dir-name>
     -X <compxlib-output-dir-name>
     -E <compedklib-output-dir-name>
     -lp <Your-Repository-Dir>
```

In this form, the '-E' value accounts for the possibility that some of the *pcores* in your repository may need to access the compiled models generated by Use Case I. This is very likely because the *pcores* in your repository are likely to refer to HDL sources in the EDK built-in repositories.

You can limit the compilation to named cores in the repository:

```
compedklib -o <compedklib-output-dir-name>
     -X <compxlib-output-dir-name>
     -E <compedklib-output-dir-name>
     -lp <Your-Repository-Dir>
```

```
-lib core1
-lib core2
```

In this case, the entire repository will be read but only the *pcores* indicated by the -c options will be compiled.

You can add a '-compile_sublibs' option to the above to compile the *pcores* that the indicated *pcore* depend on.

Other Details

- If the simulator is not indicated, then MTI is assumed.
- You can supply multiple '-X' and '-E' arguments. The order is important. If you have the same *pcore* in two places, the first one is used.
- Some *pcores* are secure in that their source code is not available. In such cases, the repository contains the compiled models. These are copied out into `<compedklib-output-dir-name>`.
- If your *pcores* are in your XPS project, you do not need to bother about Use Case 2. XPS/SIMGEN will create the scripts to compile them.
- Presently only VHDL is supported.
- The execution log is available in `compedklib.log`.
- If you have the MODELSIM environment variable set, the modelsim.ini file that it points to gets modified when this tool is compiling the HDL sources for MTI SE/PE.

Changes for EDK 6.3

- The user MODELSIM environment variable is never modified
- The -c option has been deprecated in favor of the -lib option
- New options: -exclude & -compile_sublibs

Simulation Models

This section describes how to generate each of the three FPGA simulation stages. For each stage, a different simulation model can be created by SimGen.

Behavioral Models

To create a behavioral simulation model, SimGen requires an MHS file as input. SimGen will create a set of hdl files that model the functionality of the design. Optionally, SimGen can generate a compile script for a specified vendor simulator. Also not required but if specified, SimGen can generate hdl files with data to initialize brams associated with any

processor that may exist in the design. This data is obtained from an existing executable elf file.

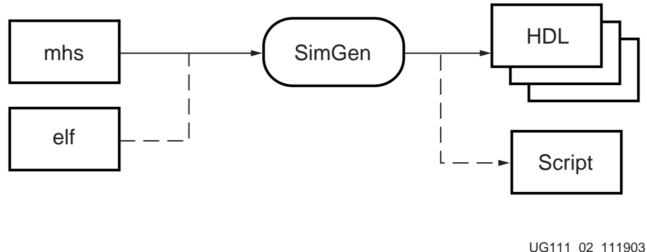


Figure 6-2: **Behavioral Simulation Model Generation**

Structural Models

To create a structural simulation model, SimGen requires an MHS file as input and associated synthesized netlist files. From these netlist files SimGen will create a set of hdl files that structurally model the functionality of the design. Optionally, SimGen can generate a compile script for a specified vendor simulator. Also not required but if specified, SimGen can generate hdl files with data to initialize brams associated with any processor that may exist in the design. This data is obtained from an existing executable elf file.

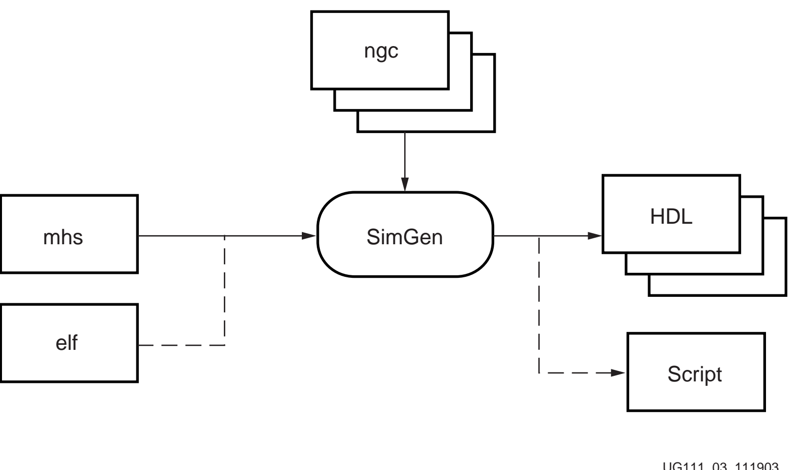


Figure 6-3: **Structural Simulation Model Generation**

Note: The EDK design flow is modular. PlatGen will generate a set of netlist files that are used by SimGen to generate structural simulation models.

Timing Models

To create a timing simulation model, SimGen requires an MHS file as input and associated implemented netlist file. From this netlist file SimGen will create an hdl file that models the design and an SDF file with appropriate timing information for it. Optionally, SimGen can generate a compile script for a specified vendor simulator. Also not required but if specified, SimGen can generate hdl files with data to initialize brams associated with any

processor that may exist in the design. This data is obtained from an existing executable elf file.

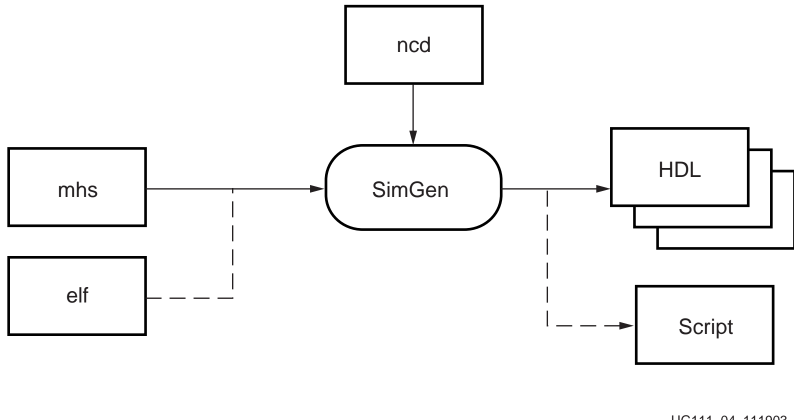


Figure 6-4: **Timing Simulation Model Generation**

Single and Mixed Language Models

SimGen allows the use of mixed language components in behavioral files for simulation. By default, SimGen will take the native language in which each component is written. Note that each component however may not be mixed language. To use this feature, a mixed language simulator is required.

All Xilinx IP components are written in VHDL. If a mixed language simulator is not available, SimGen may generate single language models by translating the hdl files that are not in the desired language. The resulting translated hdl files will be structural files.

All Structural and Timing simulation models are always single language.

SimGen Syntax

At the prompt, execute SimGen with the MHS file and appropriate options as inputs.

For example,

simgen *system_name.mhs* [*options*]

Requirements

Set up your system to use the Xilinx ISE tools. Verify that your system is properly configured. Consult the release notes and installation notes that came with your software package for more information.

Options

The following options are supported in the current version:

Help

-h, -help

The **-h** option displays the usage menu and quits.

Version

-v

The **-v** option displays the version and quits.

Options File

-f <filename>

Read command line arguments and options from file

HDL Language

-lang vhdl | verilog

The **-lang** option specifies the HDL Language.

Default: vhdl

Log Output

-log <logfile[.log]>

The **-log** option specifies the log file.

Default: simgen.log

Library Directories

-lp <*library_path*>

The **-lp** option allows you to specify library directory paths. This option may be specified more than once for multiple library directories.

Simulation Model Type

-m beh | str | tim

The **-m** option allows you to select the type of simulation models to be used. The supported simulation model types are behavioral (beh), structural (str) and timing (tim).

Default: beh

Mixed Language

-mixed yes | no

Allow the use of mixed language behavioral files.

yes - Use native language for peripherals and allow mixed language systems

no - Use structural files for peripherals not available in selected language

Default: yes

Note: Only valid when "-m beh" is used

Output Directory

-od <*output_dir*>

The **-od** option specifies the project directory path. The default is the current directory.

Target Part or Family

-p <***partname***>

The **-p** option allows you to target a specific part or family. This option must be specified.

Processor Elf Files

-pe <***proc_instance***> <***elf_file***> **{**<***elf_file***>**}**

Specify a list of elf files to be associated with the processor with instance name as defined in the MHS.

Simulator

-s mti | ncs

Generate compile script for vendor simulator.

mti - ModelSim

ncs - NcSim

Source Directory

-sd <***source_dir***>

Source directory to search for netlist files.

Top-Level Instance

-ti <***top_instance***>

When design represents a submodule, use top_instance for the top-level instance name. This switch is only valid when the "-toplevel no" switch is used.

Top-Level Module

-tm <***top_module***>

When the design represents a submodule, use top_module for the top-level entity/module name. This switch is only valid when the "-toplevel no" switch is used.

Top-Level

-toplevel yes|no

yes - Design represents a whole design

no - Design represents a level of hierarchy (submodule)

Default: yes

EDK Library Directory

-E <***edklib_dir***>

Path to EDK simulation libraries directory. This is the output directory of the compedklib tool.

Xilinx Library Directory

-X <***xlib_dir***>

Path to Xilinx simulation libraries (unisim, simprim, XilinxCoreLib) directory. This is the output directory of the compxlib tool.

Output Files

SimGen produces all simulation files in the simulation directory within the output directory, and inside a subdirectory for each of the simulation models.

```
<output_directory>/simulation/<sim_model>
```

After a successful simgen execution, the simulation directory contains the following files:

`*peripheral*_wrapper.[vhd|v]`

Modular simulation files for each component. Not applicable for timing models.

`*system_name*.[vhd|v]`

The top level HDL file of the design.

`*system_name*.sdf`

The Standard Delay Format file with the appropriate block and net delays from the place and route process used only for timing simulation.

`*system_name*.[do|sh]`

Script to compile the hdl files and load the compiled simulation models in the simulator.

Memory Initialization

If a design contains banks of memory for a system, the corresponding memory simulation models can be initialized with data. With the -pe switch, a list of executable elf files to associate to a given processor instance can be specified.

The compiled executable files are generated with the appropriate gcc compiler or assembler, from corresponding C or assembly source code.

Note: Memory initialization of structural simulation models is only supported when the netlist file has hierarchy preserved.

VHDL

For vhdl simulation models, execute SimGen with the -pe option to generate a VHDL file. This file will contain a configuration for the system with all initialization values. For example:

```
simgen system.mhs -pe mblaze executable.elf -l vhdl ...
```

This command generates the VHDL system configuration in the file *system_init.vhd*. This file is used along with your system to initialize memory. The bram blocks connected to the processor mblaze will contain the data in executable.elf.

Verilog

For verilog simulation models, execute SimGen with the -pe option to generate a verilog file. This file will contain defparam constructs that initialize memory. For example:

```
simgen system.mhs -pe mblaze executable.elf -l verilog ...
```

This command generates the verilog memory initialization file *system_init.v.* This file is used along with your system to initialize memory. The bram blocks connected to the processor mblaze will contain the data in executable.elf.

Simulating Your Design

When simulating your design, there are some special considerations you need to keep in mind such as the global reset and tristate nets. Xilinx ISE Tools provide detailed information on how to simulate your VHDL or Verilog design. Please refer to Chapter 6, "Verifying Your Design" in the ISE *Synthesis and Verification Design Guide* for more information. A PDF version of this document can be found at

/doc/usenglish/books/docs/sim/sim.pdf

in your XILINX install area, or online at

http://www.xilinx.com/support/sw_manuals/xilinx6/index.htm

Current Limitations

SimGen does not support generation of mixed level simulation models.

SimGen does not provide automated generation of simulation testbenches.

SimGen does not provide simulation models for external memories and does not have automated support for any. External memory models need to be instantiated and connected in the simulation testbench, and initialized according to the model specifications.

Chapter 7

Library Generator

This chapter describes the Library Generator (LibGen) utility needed for the generation of libraries and drivers for embedded soft processors. It also describes how the user can customize peripherals and associated drivers. The chapter contains the following sections:

- "Overview"
- "Tool Usage"
- "Tool Options"
- "Load Path"
- "Output Files"
- "Libraries and Drivers Generation"
- "MSS Parameters"
- "Drivers"
- "Libraries"
- "OS"
- "Interrupts and Interrupt Controller"
- "XMDSTUB Peripherals (MicroBlaze Specific)"
- "STDIN and STDOUT Peripherals"

Overview

LibGen is generally the first tool run to configure libraries and device drivers. LibGen takes an MSS (Microprocessor Software Specification) file created by the user as input. The MSS file defines the drivers associated with peripherals, standard input/output devices, interrupt handler routines, and other related software features. LibGen configures libraries and drivers with this information. For further description of the MSS file format, refer to Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual.*

Note: The EDK offers a RevUp tool to convert any older MSS file format to a new MSS format. See Chapter 9, "Format Revision Tool" for more information.

Tool Usage

LibGen is run as follows:

```
libgen [options] filename.mss
```

Tool Options

The following options are supported in this version:

-h, -help (Help)

This option causes LibGen to display the usage menu and exit.

-v (display version information)

This option displays the version number of LibGen.

-log *logfile[.log]*

This option specifies the log file. The default is `libgen.log`.

-p *part_name* (architecture family)

This option defines the target device defined either as architecture family or partname. Use -**h** option to get a list of values for target family.

-od *output_dir* (specify output directory)

This option specifies the output directory output_dir. The default is the current directory. All output files and directories are generated in the output directory. The input file `filename.mss` is taken from the current working directory. This output directory is also called OUTPUT_DIR, and the directory from which LibGen is invoked is called USER_PROJECT for convenience in the documentation.

-sd *source_dir* (specify source directory)

This option specifies the source directory source_dir for searching the input files (MHS). The default is the current working directory.

-lp *library_path* (specify library path for user peripherals and drivers repositories)

This option specifies a library containing repositories of user peripherals, drivers, OS's, and libraries. LibGen looks for:

- Drivers in the directory library_path/<sub_dir>/drivers/
- Libraries in the directory library_path/<sub_dir>/sw_services/
- OS's in the directory library_path/<sub_dir>/bsp/

Here <sub_dir> is a subdirectory under library_path.

-mhs *mhsfile.mhs* (specify MHS file to be used)

This option specifies the MHS file to be used for the LibGen run. The following is the order used by LibGen to find the name of an MHS file. The following is the order LibGen uses to search and locate `mhsfile.mhs` for a run:

- Current working directory (USER_PROJECT/).
- If no -**mhs** option is used, look in the MSS file for the parameter HW_SPEC_FILE to get the `mhsfilename.`

- If no HW_SPEC_FILE parameter is found in the MSS file, use the base name `mssfile` (name without .mss extension) with the .mhs extension as the `mhsfilename`.

-lib

This option can be used to copy libraries and drivers but not to compile them.

Load Path

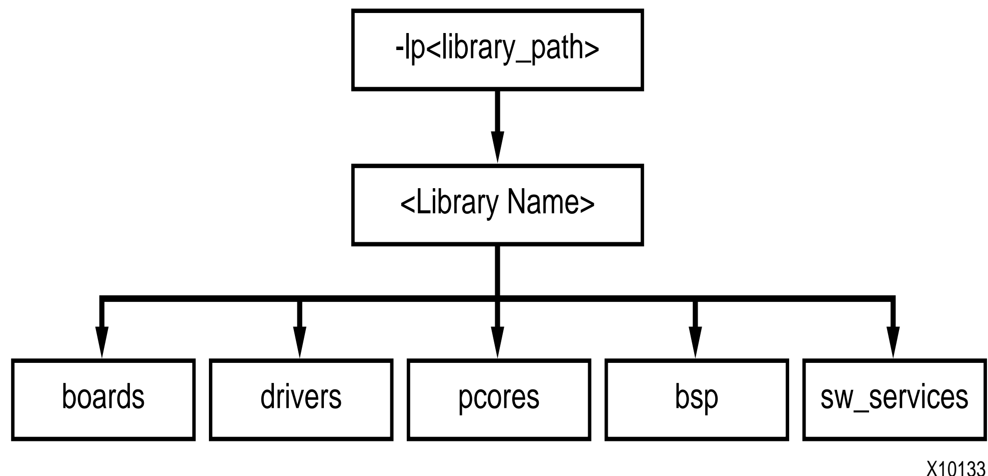


Figure 7-1: **Peripheral/Drivers/Libraries/OS's Directory Structure**

Refer to Figure 7-1 and Figure 7-2 for diagrams of the drivers/libraries/OS's directory structure.

On a UNIX system, the drivers/libraries/BSP reside in the following locations:

Drivers:

$XILINX_EDK/sw/<Library Name>/drivers

Libraries:

$XILINX_EDK/sw/<Library Name>/sw_services

OS's:

$XILINX_EDK/sw/<BSP Name>/bsp

On a PC, the drivers/libraries reside in the following location:

Drivers:

%XILINX_EDK%\sw\<Library Name>\drivers

Libraries:

%XILINX_EDK%\sw\<Library Name>\sw_services

OS's:

%XILINX_EDK%\sw\<BSP Name>\bsp

To specify additional directories, use one of the following options:

- Current working directory from which LibGen was launched.
- Set the EDK tool option -**lp**. LibGen looks for drivers, OS's and libraries under each of the subdirectories of the path specified in the -**lp** option.

LibGen uses a search priority mechanism to locate drivers/libraries, as follows:

1. Searching the current working directory:
   a. Drivers: Search for drivers inside the drivers or pcores directory in the current working directory in which LibGen is invoked.
   b. Libraries: Search for libraries inside `sw_services` directory in the current working directory in which LibGen is invoked.
   c. OS: Search for OS's inside the `bsp` directory in the current working directory from which LibGen is invoked
2. Searching the repositories under the library path directory specified using the -**lp** option:
   a. Drivers: For drivers, search <library_path>/<Library Name>/drivers and <library_path>/<Library Name>/pcores (UNIX) or <library_path>\<Library Name>\drivers and <library_path>\<Library Name>\pcores (PC) as specified by the -**lp** option.
   b. Libraries: For Libraries, search <library_path>/<Library Name>/sw_services (UNIX) or <library_path>/<Library Name>\sw_services (PC) as specified by the -**lp** option. Here <library_path> is the directory argument to -**lp** option and <Library Name> is a subdirectory under <library_path>.
   c. OS's: For OS's, search <library_path>/<OS Name>/bsp (UNIX) or <library_path>/<OS Name>\bsp (PC) as specified by the -**lp** option. Here <library_path> is the directory argument to the -**lp** option and <OS Name> is a subdirectory under <library_path>.
3. Searching the EDK install area:
   a. Drivers: Search $XILINX_EDK/sw/<Library Name>/drivers (UNIX) or %XILINX_EDK%\sw\<Library Name>\drivers (PC)
   b. Libraries: Search $XILINX_EDK/sw/<Library Name>/sw_services (UNIX) and %XILINX_EDK%\sw\<Library Name>\sw_services
   c. OS's: Search $XILINX_EDK/sw/<Library Name>/bsp (UNIX) and %XILINX_EDK%\sw\<Library Name>\bsp

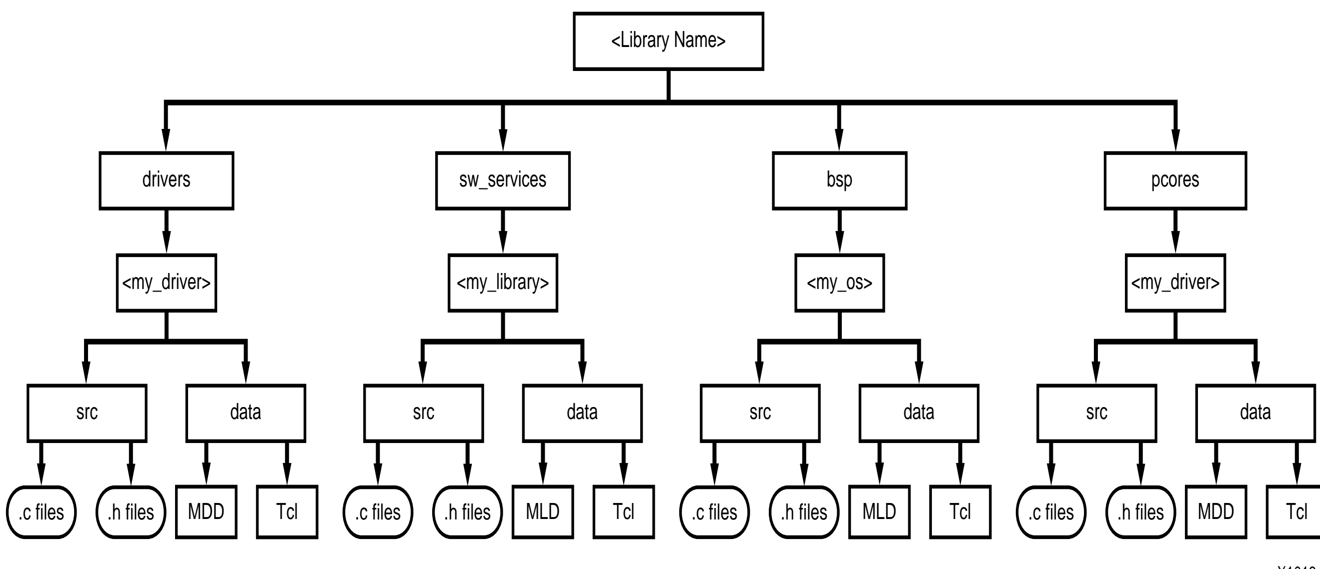


Figure 7-2: **Directory Structure of Drivers, OS's, and Libraries**

Output Files

LibGen generates directories and files in the `USER_PROJECT` directory. For every processor instance in the MSS file, LibGen generates a directory with the name of the processor instance. Within each processor instance directory, LibGen generates the following directories and files:

include directory

The include directory contains C header files that are needed by drivers. The include file `xparameters.h` is also created through LibGen in this directory. This file defines base addresses of the peripherals in the system, **#defines** needed by drivers, OS's, libraries and user programs, as well as function prototypes. The MDD file for each driver specifies the definitions that must be customized for each peripheral that uses the driver. Refer to Chapter 8, "Microprocessor Driver Definition (MDD)," in the *Platform Specification Format Reference Manual* for more information. The MLD file for each OS and library specifies the definitions that must be customized. Refer to Chapter 7, "Microprocessor Library Definition (MLD)," in the *Platform Specification Format Reference Manual* for more information.

lib directory

The `lib` directory contains `libc.a`, `libm.a`, and `libxil.a` libraries. The `libxil` library contains driver functions that the particular processor can access. More information on the libraries can be found in the "Xilinx Microkernel (XMK)" chapter in the *EDK OS and Libraries Reference Manual*.

libsrc directory

The `libsrc` directory contains intermediate files and makefiles that are needed to compile the OS's, libraries, and drivers. The directory contains peripheral-specific driver files, BSP files for the OS, and library files that are copied from the EDK and user driver/OS/library directories. Refer to the "Drivers", "OS", and "Libraries" sections of this chapter for more information.

code directory

The code directory is a repository for EDK executables. LibGen creates `xmdstub.elf` (for MicroBlaze on-board debug) in this directory.

Note: LibGen removes all the above directories every time the tool is run. Users must put in their sources/executables or any other files in a user created area.

Libraries and Drivers Generation

Basic Philosophy

This section describes the basic philosophy of library and drivers generation.

The MHS and the MSS files define a system. For each processor in the system, LibGen finds the list of addressable peripherals. For each processor, a unique list of drivers and libraries are built. LibGen runs the following for each processor:

- Build the directory structure as defined in the "Output Files" section.
- Copies the necessary source files for the drivers/OS's/libraries into the processor instance specific area: `OUTPUT_DIR/processor_instance_name/libsrc`.
- Calls the design rule check (defined as an option in the MDD/MLD file) procedure for each of the drivers, OS's, and libraries visible to the processor.
- Calls the *generate* Tcl procedure (if defined in the Tcl file associated with an MDD/MLD) for each of the drivers/OS's/libraries visible to the processor. This generates the necessary configuration files for each of the drivers/OS's/libraries in the include directory of the processor.
- Calls the *post_generate* Tcl procedure (if defined in the Tcl file associated with an MDD/MLD) for each of the drivers/OS's/libraries visible to the processor.
- Runs *make* (with targets "*include*" and "*libs*") for the OS's, drivers, and libraries specific to the processor.
- Calls the *execs_generate* Tcl procedure (if defined in the Tcl file associated with an MDD/MLD) for each of the drivers/OS's/libraries visible to the processor.

MDD/MLD and Tcl

A Driver/Library has two data files associated with it:

- Data Definition File (MDD/MLD): This file defines the configurable parameters for the driver/OS/library.
- Data Generation File (Tcl): This file uses the parameters configured in the MSS file for a driver/OS/library to generate data. Data generated includes but is not limited to generation of header files, C files, running DRCs for the driver/OS/library and generating executables. The Tcl file includes procedures that are called by LibGen at

various stages of its execution. Various procedures in a Tcl file includes **DRC** (name of DRC given in the MDD/MLD file), **generate** (LibGen-defined procedure) called after files are copied, **post_generate** (LibGen-defined procedure) called after **generate** has been called on all drivers, OS's and libraries, **execs_generate** (LibGen-defined procedure) called after the BSPs, libraries and drivers have been generated.

Note: A driver/OS/library need not have the data generation (Tcl) file.

For more information about the Tcl procedures and MDD/MLD related parameters, refer to chapter Chapter 8, "Microprocessor Driver Definition (MDD)," in the *Platform Specification Format Reference Manual* and Chapter 7, "Microprocessor Library Definition (MLD)," in the *Platform Specification Format Reference Manual*.

MSS Parameters

For a complete description of the MSS format and all the parameters that MSS supports, refer to Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual*.

Drivers

Most peripherals require software drivers. The EDK peripherals are shipped with associated drivers, libraries and BSPs. Refer to "Device Driver Programmer Guide" chapter in the *Processor IP Reference Guide* for more information on driver functions.

The MSS file includes a driver block for each peripheral instance. The block contains a reference to the driver by name (DRIVER_NAME parameter) and the driver version (DRIVER_VER). There is no default value for these parameters. A driver LEVEL is also specified depending on the driver functionality required. The driver directory contains C source and header files for each level of drivers and a makefile for the driver.

A Driver has an MDD file and/or a Tcl file associated with it. The MDD file for the driver specifies all configurable parameters for the drivers. This is the data definition file. Each MDD file has a corresponding Tcl file associated with it. This Tcl file generates data that includes generation of header files, generation of C files, running DRCs for the driver and generating executables. Refer to Chapter 8, "Microprocessor Driver Definition (MDD)," in the *Platform Specification Format Reference Manual* and Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual* for more information.

Users can write their own drivers. These drivers must be in a specific directory under USER_PROJECT/drivers or library_name/drivers, as shown in Figure 7-1. The DRIVER_NAME attribute allows the user to specify any name for their drivers, which is also the name of the driver directory. The source files and makefile for the driver must be in the **src/** subdirectory under the *driver_name* directory. The makefile should have the targets *"include"* and *"libs"*. Each driver must also contain an MDD file and a Tcl file in the **data/** subdirectory. Refer to the existing EDK drivers to get an understanding of the structure of the drivers. Refer to Chapter 8, "Microprocessor Driver Definition (MDD)," in the *Platform Specification Format Reference Manual* for details on how to write an MDD and its corresponding Tcl file.

Libraries

The MSS file now includes a library block for each library. The library block contains a reference to the library name (LIBRARY_NAME parameter) and the library version

(LIBRARY_VER). There is no default value for these parameters. The library directory contains C source and header files and a makefile for the library.

The MLD file for each driver specifies all configurable options for the drivers. Each MLD file has a corresponding Tcl file associated with it. Refer to Chapter 7, "Microprocessor Library Definition (MLD)," in the *Platform Specification Format Reference Manual* and Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual* for more information.

Users can write their own libraries. These libraries must be in a specific directory under `USER_PROJECT/sw_services` or `library_name/sw_services` as shown in Figure 7-1. The `LIBRARY_NAME` attribute allows the user to specify any name for their libraries, which is also the name of the library directory. The source files and makefile for the library must be in the *src* subdirectory under the *library_name* directory. The makefile should have the targets "*include*" and "*libs*". Each library must also contain an MLD file and a Tcl file in the `data` subdirectory. Refer to the existing EDK libraries to get an understanding of the structure of the libraries. Refer to Chapter 7, "Microprocessor Library Definition (MLD)," in the *Platform Specification Format Reference Manual* for details on how to write an MLD and its corresponding Tcl file.

OS

The MSS file now includes an OS block for each processor instance. The OS block contains a reference to the OS name (OS_NAME parameter), and the OS version (OS_VER). There is no default value for these parameters. The `bsp` directory contains C source and header files and a makefile for the OS.

The MLD file for each OS specifies all configurable options for the OS. Each MLD file has a corresponding Tcl file associated with it. Refer to Chapter 7, "Microprocessor Library Definition (MLD)," in the *Platform Specification Format Reference Manual* and Chapter 6, "Microprocessor Software Specification (MSS)," in the *Platform Specification Format Reference Manual* for more information.

Users can write their own OS's. These OS's must be in a specific directory under USER_PROJECT/bsp or library_name/bsp as shown in Figure 7-1, page 131. The OS_NAME attribute allows the user to specify any name for an OS, which is also the name of the OS directory. The source files and makefile for the OS must be in the `src` subdirectory under the `os_name` directory. The makefile should have the targets "*include*" and "*libs*". Each OS must also contain an MLD file and a Tcl file in the `data` subdirectory. Refer to the existing EDK OS's to get an understanding of the structure of the OS's. Refer to Chapter 7, "Microprocessor Library Definition (MLD)," in the *Platform Specification Format Reference Manual* for details on how to write an MLD and its corresponding Tcl file.

Interrupts and Interrupt Controller

Importance of Instantiation

An interrupt controller peripheral must be instantiated if the MHS file has multiple interrupt ports connected. LibGen statically configures interrupts and interrupt handlers through the Tcl file for the interrupt controller. Alternately, users can dynamically register interrupt handlers in the user code. Interrupts for the peripherals need to be enabled in the user code.

Interrupt Controller Driver Customization

In the MSS file, the INT_HANDLER parameter allows an interrupt handler routine to be associated with the interrupt signal. The Interrupt Controller's Tcl file uses this parameter to configure the interrupt controller handler to call the appropriate peripheral handlers on an interrupt. The functionality of these handler routines is left to the user to implement. If the INT_HANDLER parameter is not specified, a default dummy handler routine for the peripheral is used.

For MicroBlaze: if there is only one interrupt driven peripheral, an interrupt controller need not be used. However, the peripheral should still have an interrupt handler routine specified. Otherwise a default one is used.

When MicroBlaze is the processor to which the interrupt controller is connected, and when `mb-gcc` is the compiler used to compile drivers, the Tcl file associated with the MicroBlaze driver MDD designates the interrupt controller handler as the main interrupt handler.

For the PowerPC processor, the user is responsible for setting up the exception table. Refer to Chapter 5, "Interrupt Management" in the *Platform Studio User Guide* for more information.

XMDSTUB Peripherals (MicroBlaze Specific)

These are peripherals that are used specifically for debug with the **xmdstub** program (For more information about the debug program **xmdstub**, refer to Chapter 14, "Xilinx Microprocessor Debugger (XMD)"). The attribute XMDSTUB_PERIPHERAL is used for denoting the debug peripheral instance. LibGen uses this attribute to generate the debug program **xmdstub**.

STDIN and STDOUT Peripherals

Peripherals that handle I/O need drivers to access data. Two files `inbyte.c` and `outbyte.c` are automatically generated with calls to the driver I/O functions for STDIN and STDOUT peripherals. The driver I/O functions are specified in the MDD as the parameters INBYTE and OUTBYTE. These inbyte and outbyte functions are used by C library functions such as *scanf* and *printf*. The peripheral instance should be specified as STDIN or STDOUT in the MSS file. The STDIN/STDOUT parameters are attributes of the *standalone* OS. The inbyte and outbyte functions are generated only when the STDIN and STDOUT attributes are specified in MSS file for the *standalone* OS. Each OS is responsible for handling the STDIN/STDOUT functionality.

Platform Specification Utility

This chapter describes the various features and the usage of the Platform Specification Utility (PsfUtil) tool that enables automatic generation of Microprocessor Peripheral Description (MPD) files required to create an IP core compliant with the Embedded Development Kit (EDK). Features provided by this tool may be used with the help of Create/Import Peripheral Wizard in the Xilinx Platform Studio (XPS) GUI tool.

This chapter contains the following sections.

- "Tool Options"
- "Overview of the MPD Creation Process"
- "Detailed Use Models for Automatic MPD Creation"
- "DRC Checks in PsfUtility"
- "HDL Peripheral Definitions"

Tool Options

-h

Display Usage

-v

Display version

-hdl2mpd <hdlfile>

Generate MPD from VHDL/Ver src/prj file.

Sub-options:

-lang <ver|vhdl|pao>

Specify language

-top <design>

Specify top level entity/module name

{-bus <opb|plb|dcr|lmb> <m|s|ms>}

Specify one or more Bus Interfaces of the core

{-tbus <transparent_bus_name> bram_port}

Specify one or more Transparent Bus Interfaces of the core

-o <outfile>

Specify output filename, Default : stdout

-pao2mpd <paofile>

Generate MPD from Peripheral Analyze Order (PAO) file.

Sub-options:

-lang <ver | vhdl | pao>

Specify language

-top <design>

Specify top level entity/module name

{-bus <opb | plb | dcr | lmb> <m | s | ms>}

Specify one or more Bus Interfaces of the core

{-tbus <transparent_bus_name> bram_port}

Specify one or more Transparent Bus Interfaces of the core

-o <outfile>

Specify output filename, Default : stdout

Overview of the MPD Creation Process

PsfUtility may be used automatically create MPD specifications from the VHDL specification of the core. The steps involved to create a core and deliver it through EDK are

- Code the IP in VHDL or Verilog using strict naming conventions for all Bus signals, Clock signals, Reset signals and Interrupt signals. These naming conventions are described in detail in VHDL IP Peripheral Guide. ***Following these naming conventions will enable PsfUtility create a correct and complete MPD.***
- Create an XST project file or a Peripheral Analyze Order (PAO) file that lists all the HDL sources required to implement the IP. Invoke PsfUtility by providing the XST project file or the PAO file with additional options. For more information on invoking PsfUtility with different options, see "Detailed Use Models for Automatic MPD Creation".

Detailed Use Models for Automatic MPD Creation

PsfUtility may be invoked in a variety of ways depending on the bus standard and type of bus interfaces of the peripheral and the number of bus interfaces a peripheral contains. Bus standards and types may be one of

- OPB SLAVE
- OPB MASTER
- OPB MASTER_SLAVE
- PLB SLAVE
- PLB MASTER
- PLB MASTER_SLAVE
- DCR SLAVE
- LMB SLAVE
- TRANSPARENT BUS (special case)

Peripherals with a Single Bus Interface

Majority of processor peripherals fall into this category. This is also the simplest usage model for PsfUtility. For most peripherals, complete MPD specifications can be obtained without specification of any additional attributes in the source code.

Signal Naming Conventions

The signal names must follow conventions as specified in the HDL Peripheral Definition Guide. Since there is only one bus interface, no *bus identifier* needs to be specified for the bus signals.

Invoking PsfUtility

The command line for invoking PsfUtility is as follows

```
psfutil -hdl2mpd <hdlfile> -lang <vhdl|ver> -top <top_entity> -bus
<busstd> <bustype> -o <mpdfile>
```

For example, to create an MPD specification for an OPB SLAVE peripheral, say uart, the command would be

```
psfutil -hdl2mpd uart.prj -lang vhdl -top uart -bus opb s -o uart.mpd
```

Peripherals with Multiple Bus Interfaces

Some peripherals may have multiple bus interfaces associated with it. These interfaces may be Exclusive bus interfaces or Non-exclusive bus interfaces or a combination of both. All bus interfaces of the peripheral that can be connected to the peripheral at the same time are exclusive interfaces. For example, an OPB Slave bus interface and a DCR Slave bus interface are exclusive bus interfaces on a peripheral as they can both be connected at the same time. ***Peripherals with exclusive bus interfaces CAN NOT have any ports that can be connected to more than one of the exclusive interfaces.***

Non-exclusive bus interfaces are those interfaces that cannot be connected at the same time. ***Peripherals with non-exclusive bus interfaces WILL HAVE ports that can be connected to more than one of the non-exclusive interfaces. Further, non-exclusive interfaces WOULD have the same bus interface standard.*** For example, an OPB Slave interface and a OPB Master Slave interface are non-exclusive if they are connected to the same slave ports of the peripheral.

Non-Exclusive Bus Interfaces

Signal Naming Conventions

The signal names must follow conventions as specified in the HDL Peripheral Definition Guide. For non-exclusive bus interfaces, *bus identifiers* need not be specified for the bus signals.

Invoking PsfUtility with buses specified in command line

Buses can be specified on the command line when the bus signals are not prefixed with bus identifiers. The command line for invoking PsfUtil is as follows

```
psfutil -hdl2mpd <hdlfile> -lang <vhdl|ver> -top <top_entity> {-bus
<busstd> <bustype>} -o <mpdfile>
```

For example, to create an MPD specification for a peripheral with a PLB slave interface and a PLB Master Slave interface, say gemac, the command would be

```
psfutil -hdl2mpd gemac.prj -lang vhdl -top gemac -bus plb s -bus plb ms -o
gemac.prj
```

Exclusive Bus Interfaces

Signal Naming Conventions

The signal names must follow conventions as specified in the HDL Peripheral Definition Guide. B*us identifiers* need to be specified only when the peripheral has more than one bus interface of the same bus standard and type.

Invoking PsfUtility with buses specified in command line

Buses can be specified on the command line when the bus signals are not prefixed with bus identifiers. The command line for invoking PsfUtil is as follows

```
psfutil -hdl2mpd <hdlfile> -lang <vhdl|ver> -top <top_entity> {-bus
<busstd> <bustype>} -o <mpdfile>
```

For example, to create an MPD specification for a peripheral with a PLB slave interface and a DCR Slave interface, the command would be

```
psfutil -hdl2mpd mem.prj -lang vhdl -top mem -bus plb s -bus dcr s -o
mem.prj
```

Peripherals with TRANSPARENT Bus Interfaces

Some peripherals like bram controllers might have transparent bus interfaces (BUS_STD=TRANSPARENT, BUS_TYPE = UNDEF).

BRAM PORTS

To add a transparent BRAM bus interface to your core, invoke psfutil with an additional -tbus option

```
psfutil -hdl2mpd bram_ctlr.prj -lang vhdl -top bram_ctlr -bus opb s
-tbus PORTA bram_port
```

Note that the BRAM ports should follow signal naming conventions as specified in the HDL Peripheral Definition document.

DRC Checks in PsfUtility

The following DRC errors are reported by PsfUtility to enable generation of correct and complete MPDs from HDL sources. The DRC checks are listed in the order that the checks are performed.

HDL Source Errors

PsfUtility returns a failure status if errors were found in the HDL source files.

Bus Interface Checks

Given the list of bus interface of the cores, PsfUtility verifies the following

- Check and report any missing Bus Signals for every specified bus interface
- Check and report any repeated Bus Signals for every specified bus interface

PsfUtility will not generate an MPD unless all bus interface checks are completed.

HDL Peripheral Definitions

The top-level VHDL source file for an IP core defines the interface of the design. The VHDL source file has the following characteristics:

- Lists ports and default connectivity for bus interfaces
- Lists parameters (generics) and default values
- Any HDL source parameter is overwritten by the equivalent MHS assignment

Individual peripheral documentation contains information on all source file options.

Bus Interface Naming Conventions

A bus interface is a grouping of interface signals which are related. For the automation tools to function properly, certain conventions must be adhered to in the naming of the signals and parameters associated with a bus interface. When the signal naming conventions are followed, the following interface types will be automatically recognized and the MPD file will contain the BUS_INTERFACE label shown in Table 8-1.

Table 8-1: **Recognized Bus Interfaces**

Description	Bus label in MPD
Slave DCR interface	SDCR
Slave LMB interface	SLMB
Master OPB interface	MOPB
Master/slave OPB interface	MSOPB
Slave OPB interface	SOPB
Master PLB interface	MPLB
Master/slave PLB interface	MSPLB
Slave PLB interface	SPLB

For components that have more than one bus interface of the same type, a naming convention must be followed so that the automation tools can group the bus interfaces.

Naming Conventions for VHDL Generics

A key concept for cores with more than one bus interface port is the use of a *bus identifier*, which is attached to all signals grouped together in a port as well as the generics that are associated with the bus interface port. The bus identifier is discussed below.

Generic names must be VHDL compliant. Additional conventions for IP cores are:

- The generic must start with "C_".

- If more than one instance of a particular bus interface type is used on a core, a bus identifier, <*BI*>, must be used in the signal. If a bus identifier is used for the signals associated with a port, then the generics associated with that port may also optionally use the <*BI*>. If no <*BI*> string is used in the name, then the generics associated with bus parameters are assumed to be global. For example, C_DOPB_DWIDTH has a bus identifier of "D" and is associated with the bus signals that also have a bus identifier of "D". If only C_OPB_DWIDTH is present, it is associated with all OPB buses regardless of the bus identifier on the port signals.
- For cores that have only a single bus interface (which is the case for most peripherals), the use of the bus identifier string in the signal and generic names is optional and the bus identifier will not typically be included.
- All generics that specify a base address must end with _BASEADDR, and all generic that specify a high address must end with _HIGHADDR. Further, to tie these addresses with buses, these must also follow the conventions for parameters as listed above. For peripherals with more than one type of bus interface, the parameters must have the bus standard type specified in the name. For example, an address on the PLB bus must be specified as C_PLB_BASEADDR and C_PLB_HIGHADDR.

The Platform Generator automatically expands and populates certain reserved generics. In order for this to work correctly, a bus tag must be associated with these parameters. In order to have PsfUtility automatically infer this information, all the above specified conventions must be followed for all reserved generics as well. This can help prevent errors when your peripheral requires information on the platform that is generated. The following table lists the reserved generic names:

Figure 8-1: **Automatically Expanded Reserved Generics**

Parameter	Description
C_BUS_CONFIG	Bus Configuration of MicroBlaze
C_FAMILY	FPGA Device Family
C_INSTANCE	Instance name of component
C_KIND_OF_EDGE	Vector of edge sensitive (rising/falling) of interrupt signals
C_KIND_OF_LVL	Vector of level sensitive (high/low) of interrupt signals
C_KIND_OF_INTR	Vector of interrupt signal sensitivity (edge/level)
C_NUM_INTR_INPUTS	Number of interrupt signals
C_<*BI*>OPB_NUM_MASTERS	Number of OPB masters
C_<*BI*>OPB_NUM_SLAVES	Number of OPB slaves
C_<*BI*>DCR_AWIDTH	DCR Address width
C_<*BI*>DCR_DWIDTH	DCR Data width
C_<*BI*>DCR_NUM_SLAVES	Number of DCR slaves
C_<*BI*>LMB_AWIDTH	LMB Address width
C_<*BI*>LMB_DWIDTH	LMB Data width
C_<*BI*>LMB_NUM_SLAVES	Number of LMB slaves

Figure 8-1: **Automatically Expanded Reserved Generics *(Continued)***

Parameter	Description
C_<*BI*>OPB_AWIDTH	OPB Address width
C_<*BI*>OPB_DWIDTH	OPB Data width
C_<*BI*>OPB_NUM_MASTERS	Number of OPB masters
C_<*BI*>OPB_NUM_SLAVES	Number of OPB slaves
C_<*BI*>PLB_AWIDTH	PLB Address width
C_<*BI*>PLB_DWIDTH	PLB Data width
C_<*BI*>PLB_MID_WIDTH	PLB master ID width
C_<*BI*>PLB_NUM_MASTERS	Number of PLB masters
C_<*BI*>PLB_NUM_SLAVES	Number of PLB slaves

Reserved Parameters

C_BUS_CONFIG

The C_BUS_CONFIG parameter defines the bus configuration of the MicroBlaze processor. This parameter is automatically populated by Platform Generator.

C_FAMILY

The C_FAMILY parameter defines the FPGA device family. This parameter is automatically populated by Platform Generator.

C_INSTANCE

The C_INSTANCE parameter defines the instance name of the component. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_MASTERS

The C_OPB_NUM_MASTERS parameter defines the number of OPB masters on the bus. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_SLAVES

The C_OPB_NUM_SLAVES parameter defines the number of OPB slaves on the bus. This parameter is automatically populated by Platform Generator.

C_DCR_AWIDTH

The C_DCR_AWIDTH parameter defines the DCR address width. This parameter is automatically populated by Platform Generator.

C_DCR_DWIDTH

The C_DCR_DWIDTH parameter defines the DCR data width. This parameter is automatically populated by Platform Generator.

C_DCR_NUM_SLAVES

The C_DCR_NUM_SLAVES parameter defines the number of DCR slaves on the bus. This parameter is automatically populated by Platform Generator.

C_LMB_AWIDTH

The C_LMB_AWIDTH parameter defines the LMB address width. This parameter is automatically populated by Platform Generator.

C_LMB_DWIDTH

The C_LMB_DWIDTH parameter defines the LMB data width. This parameter is automatically populated by Platform Generator.

C_LMB_NUM_SLAVES

The C_LMB_NUM_SLAVES parameter defines the number of LMB slaves on the bus. This parameter is automatically populated by Platform Generator.

C_OPB_AWIDTH

The C_OPB_AWIDTH parameter defines the OPB address width. This parameter is automatically populated by Platform Generator.

C_OPB_DWIDTH

The C_OPB_DWIDTH parameter defines the OPB data width. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_MASTERS

The C_OPB_NUM_MASTERS parameter defines the number of OPB masters on the bus. This parameter is automatically populated by Platform Generator.

C_OPB_NUM_SLAVES

The C_OPB_NUM_SLAVES parameter defines the number of OPB slaves on the bus. This parameter is automatically populated by Platform Generator.

C_PLB_AWIDTH

The C_PLB_AWIDTH parameter defines the PLB address width. This parameter is automatically populated by Platform Generator.

C_PLB_DWIDTH

The C_PLB_DWIDTH parameter defines the PLB data width. This parameter is automatically populated by Platform Generator.

C_PLB_MID_WIDTH

The C_PLB_MID_WIDTH parameter defines the PLB master ID width. This is set to log2(S). This parameter is automatically populated by Platform Generator.

C_PLB_NUM_MASTERS

The C_PLB_NUM_MASTERS parameter defines the number of PLB masters on the bus. This parameter is automatically populated by Platform Generator.

C_PLB_NUM_SLAVES

The C_PLB_NUM_SLAVES parameter defines the number of PLB slaves on the bus. This parameter is automatically populated by Platform Generator.

Signal Naming Conventions

This section provides naming conventions for bus interface signal names. These conventions are flexible to accommodate embedded processor systems that have more than one bus interface and more than one bus interface port per component. A key concept for cores with more than one bus interface port is the use of a ***bus identifier***, which is attached to all signals grouped together in a port as well as the parameters that are associated with the bus interface port. The bus identifier is discussed below.

The names must be HDL compliant. Additional conventions for IP cores are:

- The first character in the name must be alphabetic and uppercase.
- The fixed part of the identifier for each signal must appear exactly as shown in the applicable section below. Each section describes the required signal set for one type of bus interface.
- If more than one instance of a particular bus interface type is used on a core, a bus identifier, *<BI>*, must be used in the signal identifier. The bus identifier can be as simple as a single letter or as complex as a descriptive string with a trailing underscore. The *<BI>* must be included in the port's signal identifiers in the following cases:
    - The core has more than one slave PLB port.
    - The core has more than one master PLB port.
    - The core has more than one slave LMB port.
    - The core has more than one slave DCR port.
    - The core has more than one master DCR port.
    - The core has more than one OPB port of any type (master, slave, or master/slave).
    - The core has more than one port of any type and the choice of *<Mn>* or *<Sln>* causes ambiguity in the signal names. For example, a core with both a master OPB port and master PLB port and the same *<Mn>* string for both ports would require a *<BI>* string to differentiate the ports since the address bus signal would be ambiguous without *<BI>*.

For cores that have only a single bus interface (which is the case for most peripherals), the use of the bus identifier string in the signal names is optional and the bus identifier will not typically be included.

Global Ports

The names for the global ports of a peripheral (such as clock and reset signals) are standardized. You can use any name for other global ports (such as the interrupt signal).

LMB - Clock and Reset

```
LMB_Clk
LMB_Rst
```

OPB - Clock and Reset

```
OPB_Clk
OPB_Rst
```

PLB - Clock and Reset

```
PLB_Clk
PLB_Rst
```

Slave DCR Ports

Slave DCR ports must follow these naming conventions:

- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, "DCR" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- <*nDCR*> is a meaningful name or acronym for the slave input. The last three characters of <*nDCR*> must contain the string, "DCR" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single slave DCR port, and required for peripherals with multiple slave DCR ports. <*BI*> must not contain the string, "DCR" (upper or lower case or mixed case). For peripherals with multiple slave DCR ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> is optional.

DCR Slave Outputs

For interconnection to the DCR, all slaves must provide the following outputs:

```
<BI><Sln>_dcrDBus : out std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
<BI><Sln>_dcrAck  : out std_logic;
```

Examples:

```
Uart_dcrAck   : out std_logic;
Intc_dcrAck   : out std_logic;
Memcon_dcrAck : out std_logic;
Bus1_timer_dcrAck  : out std_logic;
Bus1_timer_dcrDBus : out std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
Bus2_timer_dcrAck  : out std_logic;
Bus2_timer_dcrDBus : out std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
```

DCR Slave Inputs

For interconnection to the DCR, all slaves must provide the following inputs:

```
<BI><nDCR>_ABus     : in  std_logic_vector(0 to C_<BI>DCR_AWIDTH-1);
<BI><nDCR>_DBus     : in  std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
<BI><nDCR>_Read     : in  std_logic;
<BI><nDCR>_Write    : in  std_logic;
```

Examples:

```
DCR_DBus      : in  std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
Bus1_DCR_DBus : in  std_logic_vector(0 to C_<BI>DCR_DWIDTH-1);
```

Slave LMB Ports

Slave LMB ports must follow these naming conventions:

- *<Sln>* is a meaningful name or acronym for the slave output. *<Sln>* must not contain the string, "LMB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- *<nLMB>* is a meaningful name or acronym for the slave input. The last three characters of *<nLMB>* must contain the string, "LMB" (upper or lower case or mixed case).
- *<BI>* is a Bus Identifier; it is optional for peripherals with a single slave LMB port, and required for peripherals with multiple slave LMB ports. *<BI>* must not contain the string, "LMB" (upper or lower case or mixed case). For peripherals with multiple slave LMB ports, the *<BI>* strings must be unique for each bus interface.
- If *<BI>* is present, then *<Sln>* is optional.

LMB Slave Outputs

For interconnection to the LMB, all slaves must provide the following outputs:

```
<BI><Sln>_DBus  : out std_logic_vector(0 to C_<BI>LMB_DWIDTH-1);
<BI><Sln>_Ready : out std_logic;
```

Examples:

```
D_Ready : out std_logic;
I_Ready : out std_logic;
```

LMB Slave Inputs

For interconnection to the LMB, all slaves must provide the following inputs:

```
<BI><nLMB>_ABus        : in  std_logic_vector(0 to C_<BI>LMB_AWIDTH-1);
<BI><nLMB>_AddrStrobe  : in  std_logic;
<BI><nLMB>_BE         : in  std_logic_vector(0 to C_<BI>LMB_DWIDTH/8-1);
<BI><nLMB>_Clk         : in  std_logic;
<BI><nLMB>_ReadStrobe  : in  std_logic;
<BI><nLMB>_Rst         : in  std_logic;
<BI><nLMB>_WriteDBus   : in  std_logic_vector(0 to C_<BI>LMB_DWIDTH-1);
<BI><nLMB>_WriteStrobe : in  std_logic;
```

Examples:

```
LMB_ABus  : in  std_logic_vector(0 to C_LMB_AWIDTH-1);
DLMB_ABus : in  std_logic_vector(0 to C_DLMB_AWIDTH-1);
```

Master OPB Ports

The signal list shown below applies to master OPB ports that are independent of slave OPB ports. For the signal list for cores that use a combined master/slave bus interface, see XXX.

Master OPB ports must follow these naming conventions:

- *<Mn>* is a meaningful name or acronym for the master output. *<Mn>* must not contain the string, "OPB" (upper or lower case or mixed case), so that master outputs will not be confused with bus outputs.

- <*nOBP*> is a meaningful name or acronym for the master input. The last three characters of <*nOPB*> must contain the string, "OPB" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single OPB port (of any type), and required for peripherals with multiple OPB ports (of any type or mix of types). <*BI*> must not contain the string, "OPB" (upper or lower case or mixed case). For peripherals with multiple OPB ports, the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Mn*> is optional.

OPB Master Outputs

For interconnection to the OPB, all masters must provide the following outputs:

```
<BI><Mn>_ABus     : out std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><Mn>_BE       : out std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><Mn>_busLock  : out std_logic;
<BI><Mn>_DBus     : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Mn>_request  : out std_logic;
<BI><Mn>_RNW      : out std_logic;
<BI><Mn>_select   : out std_logic;
<BI><Mn>_seqAddr  : out std_logic;
```

Examples:

```
IM_request     : out std_logic;
Bridge_request : out std_logic;
O2Ob_request   : out std_logic;
```

OPB Master Inputs

For interconnection to the OPB, all masters must provide the following inputs:

```
<BI><nOPB>_Clk      : in  std_logic;
<BI><nOPB>_DBus     : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_errAck   : in  std_logic;
<BI><nOPB>_MGrant   : in  std_logic;
<BI><nOPB>_retry    : in  std_logic;
<BI><nOPB>_Rst      : in  std_logic;
<BI><nOPB>_timeout  : in  std_logic;
<BI><nOPB>_xferAck  : in  std_logic;
```

Examples:

```
IOPB_DBus      : in  std_logic_vector(0 to C_IOPB_DWIDTH-1);
OPB_DBus       : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
Bus1_OPB_DBus  : in  std_logic_vector(0 to C_Bus1_OPB_DWIDTH-1);
```

Slave OPB Ports

The signal list shown below applies to master OPB ports that are independent of slave OPB ports. For the signal list for cores that use a combined master/slave bus interface, see XXX.

Slave OPB ports must follow these naming conventions:

- <*Sln*> is a meaningful name or acronym for the slave output. <*Sln*> must not contain the string, "OPB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.

- <*nOPB*> is a meaningful name or acronym for the slave input. The last three characters of <*nOPB*> must contain the string, "OPB" (upper or lower case or mixed case).
- <*BI*> is a Bus Identifier; it is optional for peripherals with a single OPB port, and required for peripherals with multiple OPB ports (of any type). <*BI*> must not contain the string, "OPB" (upper or lower case or mixed case). For peripherals with multiple OPB ports (of any type or mix of types), the <*BI*> strings must be unique for each bus interface.
- If <*BI*> is present, then <*Sln*> is optional.

OPB Slave Outputs

For interconnection to the OPB, all slaves must provide the following outputs:

```
<BI><Sln>_DBus    : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Sln>_errAck  : out std_logic;
<BI><Sln>_retry   : out std_logic;
<BI><Sln>_toutSup : out std_logic;
<BI><Sln>_xferAck : out std_logic;
```

Examples:

```
Tmr_xferAck   : out std_logic;
Uart_xferAck  : out std_logic;
Intc_xferAck  : out std_logic;
```

OPB Slave Inputs

For interconnection to the OPB, all slaves must provide the following inputs:

```
<BI><nOPB>_ABus    : in  std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><nOPB>_BE      : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><nOPB>_Clk     : in  std_logic;
<BI><nOPB>_DBus    : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_Rst     : in  std_logic;
<BI><nOPB>_RNW     : in  std_logic;
<BI><nOPB>_select  : in  std_logic;
<BI><nOPB>_seqAddr : in  std_logic;
```

Examples:

```
OPB_DBus       : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
IOPB_DBus      : in  std_logic_vector(0 to C_IOPB_DWIDTH-1);
Bus1_OPB_DBus  : in  std_logic_vector(0 to C_Bus1_OPB_DWIDTH-1);
```

Master/Slave OPB Ports

The signal list shown below applies to master/slave type OPB ports that attach to the same OPB bus and share the input and output data buses. This type of bus interface is typically used when a peripheral has both master and slave functionality (typical when DMA is included with the peripheral) and it is advantageous for the master and slave to share the input and output data buses.

Master/Slave OPB ports must follow these naming conventions:

- <*Mn*> is a meaningful name or acronym for the master output. <*Mn*> must not contain the string, "OPB" (upper or lower case or mixed case), so that master outputs will not be confused with bus outputs.

- *<Sln>* is a meaningful name or acronym for the slave output. *<Sln>* must not contain the string, "OPB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- *<nOPB>* is a meaningful name or acronym for the slave input. The last three characters of *<nOPB>* must contain the string, "OPB" (upper or lower case or mixed case).
- *<BI>* is a Bus Identifier; it is optional for peripherals with a single OPB port, and required for peripherals with multiple OPB ports (of any type). *<BI>* must not contain the string, "OPB" (upper or lower case or mixed case). For peripherals with multiple OPB ports (of any type or mix of types), the *<BI>* strings must be unique for each bus interface.
- If *<BI>* is present, then *<Sln>* and *<Mn>* are optional.

OPB Master/Slave Outputs

For interconnection to the OPB, all master/slaves must provide the following outputs:

```
<BI><Sln>_ABus     : out std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><Sln>_BE       : out std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><Sln>_busLock  : out std_logic;
<BI><Sln>_request  : out std_logic;
<BI><Sln>_RNW      : out std_logic;
<BI><Sln>_select   : out std_logic;
<BI><Sln>_seqAddr  : out std_logic;
<BI><Sln>_DBus    : out std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><Sln>_errAck  : out std_logic;
<BI><Sln>_retry   : out std_logic;
<BI><Sln>_toutSup : out std_logic;
<BI><Sln>_xferAck : out std_logic;
```

Examples:

```
IM_request     : out std_logic;
Bridge_request : out std_logic;
O2Ob_request   : out std_logic;
```

OPB Master/Slave Inputs

For interconnection to the OPB, all master/slaves must provide the following inputs:

```
<BI><nOPB>_ABus     : in  std_logic_vector(0 to C_<BI>OPB_AWIDTH-1);
<BI><nOPB>_BE       : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH/8-1);
<BI><nOPB>_Clk      : in  std_logic;
<BI><nOPB>_DBus     : in  std_logic_vector(0 to C_<BI>OPB_DWIDTH-1);
<BI><nOPB>_errAck   : in  std_logic;
<BI><nOPB>_MGrant   : in  std_logic;
<BI><nOPB>_retry    : in  std_logic;
<BI><nOPB>_RNW      : in  std_logic;
<BI><nOPB>_Rst      : in  std_logic;
<BI><nOPB>_select   : in  std_logic;
<BI><nOPB>_seqAddr  : in  std_logic;
<BI><nOPB>_timeout  : in  std_logic;
<BI><nOPB>_xferAck  : in  std_logic;
```

Examples:

```
IOPB_DBus     : in  std_logic_vector(0 to C_IOPB_DWIDTH-1);
OPB_DBus      : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
Bus1_OPB_DBus : in  std_logic_vector(0 to C_Bus1_OPB_DWIDTH-1);
```

Master PLB Ports

Master PLB ports must follow these naming conventions:

- *<Mn>* is a meaningful name or acronym for the master output. *<Mn>* must not contain the string, "PLB" (upper or lower case or mixed case), so that master outputs will not be confused with bus outputs.
- *<nPLB>* is a meaningful name or acronym for the master input. The last three characters of *<nOPB>* must contain the string, "PLB" (upper or lower case or mixed case).
- *<BI>* is a Bus Identifier; it is optional for peripherals with a single master PLB port, and required for peripherals with multiple master PLB ports. *<BI>* must not contain the string, "PLB" (upper or lower case or mixed case). For peripherals with multiple master PLB ports, the *<BI>* strings must be unique for each bus interface.
- If *<BI>* is present, then *<Mn>* is optional.

PLB Master Outputs

For interconnection to the PLB, all masters must provide the following outputs:

```
<BI><Mn>_ABus      : out std_logic_vector(0 to C_<BI>PLB_AWIDTH-1);
<BI><Mn>_BE        : out std_logic_vector(0 to C_<BI>PLB_DWIDTH/8-1);
<BI><Mn>_RNW       : out std_logic;
<BI><Mn>_abort     : out std_logic;
<BI><Mn>_busLock   : out std_logic;
<BI><Mn>_compress  : out std_logic;
<BI><Mn>_guarded   : out std_logic;
<BI><Mn>_lockErr   : out std_logic;
<BI><Mn>_MSize     : out std_logic;
<BI><Mn>_ordered   : out std_logic;
<BI><Mn>_priority  : out std_logic_vector(0 to 1);
<BI><Mn>_rdBurst   : out std_logic;
<BI><Mn>_request   : out std_logic;
<BI><Mn>_size      : out std_logic_vector(0 to 3);
<BI><Mn>_type      : out std_logic_vector(0 to 2);
<BI><Mn>_wrBurst   : out std_logic;
<BI><Mn>_wrDBus    : out std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
```

Examples:

```
IM_request     : out std_logic;
Bridge_request : out std_logic;
O2Ob_request   : out std_logic;
```

PLB Master Inputs

For interconnection to the PLB, all masters must provide the following inputs:

```
<BI><nPLB>_Clk         : in  std_logic;
<BI><nPLB>_Rst         : in  std_logic;
<BI><nPLB>_AddrAck     : in  std_logic;
<BI><nPLB>_Busy        : in  std_logic;
<BI><nPLB>_Err         : in  std_logic;
<BI><nPLB>_RdBTerm     : in  std_logic;
<BI><nPLB>_RdDAck      : in  std_logic;
<BI><nPLB>_RdDBus     : in  std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
<BI><nPLB>_RdWdAddr    : in  std_logic_vector(0 to 3);
<BI><nPLB>_Rearbitrate : in  std_logic;
```

```
<BI><nPLB>_SSize        : in  std_logic_vector(0 to 1);
<BI><nPLB>_WrBTerm      : in  std_logic;
<BI><nPLB>_WrDAck       : in  std_logic;
```

Examples:

```
IPLB_MBusy      : in  std_logic;
Bus1_PLB_MBusy : in  std_logic;
```

Slave PLB Ports

Slave PLB ports must follow these naming conventions:

- *<Sln>* is a meaningful name or acronym for the slave output. *<Sln>* must not contain the string, "PLB" (upper or lower case or mixed case), so that slave outputs will not be confused with bus outputs.
- *<nPLB>* is a meaningful name or acronym for the slave input. The last three characters of *<nPLB>* must contain the string, "PLB" (upper or lower case or mixed case).
- *<BI>* is a Bus Identifier; it is optional for peripherals with a single slave PLB port, and required for peripherals with multiple slave PLB ports. *<BI>* must not contain the string, "PLB" (upper or lower case or mixed case). For peripherals with multiple PLB ports, the *<BI>* strings must be unique for each bus interface.
- If *<BI>* is present, then *<Sln>* is optional.

PLB Slave Outputs

For interconnection to the PLB, all slaves must provide the following outputs:

```
<BI><Sln>_addrAck      : out std_logic;
<BI><Sln>_MErr        : out std_logic_vector(0 to C_<BI>PLB_NUM_MASTERS-1);
<BI><Sln>_MBusy       : out std_logic_vector(0 to C_<BI>PLB_NUM_MASTERS-1);
<BI><Sln>_rdBTerm      : out std_logic;
<BI><Sln>_rdComp       : out std_logic;
<BI><Sln>_rdDAck       : out std_logic;
<BI><Sln>_rdDBus       : out std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
<BI><Sln>_rdWdAddr     : out std_logic_vector(0 to 3);
<BI><Sln>_rearbitrate : out std_logic;
<BI><Sln>_SSize        : out std_logic(0 to 1);
<BI><Sln>_wait         : out std_logic;
<BI><Sln>_wrBTerm      : out std_logic;
<BI><Sln>_wrComp       : out std_logic;
<BI><Sln>_wrDAck       : out std_logic;
```

Examples:

```
Tmr_addrAck   : out std_logic;
Uart_addrAck : out std_logic;
Intc_addrAck : out std_logic;
```

PLB Slave Inputs

For interconnection to the PLB, all slaves must provide the following inputs:

```
<BI><nPLB>_Clk       : in  std_logic;
<BI><nPLB>_Rst       : in  std_logic;
<BI><nPLB>_ABus      : in  std_logic_vector(0 to C_<BI>PLB_AWIDTH-1);
<BI><nPLB>_BE        : in  std_logic_vector(0 to C_<BI>PLB_DWIDTH/8-1);
<BI><nPLB>_PAValid  : in  std_logic;
<BI><nPLB>_RNW       : in  std_logic;
```

```
<BI><nPLB>_abort    : in  std_logic;
<BI><nPLB>_busLock  : in  std_logic;
<BI><nPLB>_compress : in  std_logic;
<BI><nPLB>_guarded  : in  std_logic;
<BI><nPLB>_lockErr  : in  std_logic;
<BI><nPLB>_masterID : in  std_logic_vector(0 to C_<BI>PLB_MID_WIDTH-1);
<BI><nPLB>_MSize    : in  std_logic_vector(0 to 1);
<BI><nPLB>_ordered  : in  std_logic;
<BI><nPLB>_pendPri  : in  std_logic_vector(0 to 1);
<BI><nPLB>_pendReq  : in  std_logic;
<BI>
_reqpri   : in  std_logic_vector(0 to 1);
<BI><nPLB>_size     : in  std_logic_vector(0 to 3);
<BI><nPLB>_type     : in  std_logic_vector(0 to 2);
<BI><nPLB>_rdPrim   : in  std_logic;
<BI><nPLB>_SAValid  : in  std_logic;
<BI><nPLB>_wrPrim   : in  std_logic;
<BI><nPLB>_wrBurst  : in  std_logic;
<BI><nPLB>_wrDBus   : in  std_logic_vector(0 to C_<BI>PLB_DWIDTH-1);
<BI><nPLB>_rdBurst  : in  std_logic;
```

Examples:

```
PLB_size  : in  std_logic_vector(0 to 3);
IPLB_size : in  std_logic_vector(0 to 3);
DPLB_size : in  std_logic_vector(0 to 3);
```

Chapter 9

Format Revision Tool

The contents for this chapter include:

- "Revup to EDK 6.3"
- "Revup from EDK 3.2 to EDK 6.1"

Revup to EDK 6.3

The Format Revision Tool (revup) updates an existing EDK 6.1 or EDK 6.2 project to an EDK 6.3 project. Note that if you open a project from 6.1 or 6.2 in XPS 6.3, then it will automatically revup the project to the new release. If you have a project which is from EDK release 3.2 or 3.1, XPS will not update that project. You must update the project yourself from the command line shell using **revup32to61** utility. Please refer to section "Revup from EDK 3.2 to EDK 6.1" for details.

The revup in EDK 6.3 creates backup of the current project files and then updates the existing ones.

The following files are backedup before revup:

- <system>.xmp as <system>_xmp.62
- <system>.mhs as <system>_mhs.62
- <system>.mss as <system>_mss.62
- <system>.log as <system>_log.62

The contents of the log file are also cleared after creating a backup.

If your project is already an EDK 6.2 project, there are no changes to the MHS, and MSS files from 6.2 to 6.3. Also, none of the IP or driver data files (MPD, MDD etc.) need any update in EDK 6.3.

However, if your project is a EDK 6.1 project, then revup will update the MSS files. It will also create Software Applications based on previous project settings. All changes are done automatically and no user input is required.

Tool Usage

Run the revup tool as follows from the command line:

```
revup <system>.xmp
```

The following are the options supported:

-**h** (Help)

The -**h** option displays the usage menu and quits.

Limitations

The limitations of the revup tool are:

- It can only revup EDK 6.1 or EDK 6.2 projects. Older projects must be reved up separately to EDK 6.1.
- It only performs format revup. If any IP or driver has been marked OBSOLETE in EDK 6.3, users need to change the design manually to latest versions of IP.

Revup from EDK 3.2 to EDK 6.1

The Format Revision Tool (revup32to61) updates an existing EDK 3.1 or 3.2 project to a format for EDK 6.1. Note that if you open an old project with XPS, then it will automatically revup the project to the new format. A project revup will also automatically cause revup of all the hardware repository data files (MPD, BBD, and PAO) referred to by that project and that of the local **myip** and **pcores** directories. RevUp can optionally update just the hardware repository data files . The upgrade is a format update and not an IP upgrade. Note that there is no update required for software repository (MDD, MLD) files.

In EDK 6.1, the PSF version is 2.1.0. Previous supported versions include 2.0.0 for MPD, BBD, and PAO files and version 2.1.0 for MDD, and MLD files.

EDK tools are always running with the latest formats. Only RevUp needs to maintain compatibility with older versions.

Tool Usage

Run revup32to61 as follows from the command line:

```
revup32to61 <system>.xmp
revup32to61 -rd <repository_dir>
```

The following are the options supported:

-**h** (Help)

The -**h** option displays the usage menu and quits.

-**rd** (repository directory)

The -**rd** option allows you to specify the repository directory which needs revup. The repository directory is the parent directory of the **pcores** or **myip** directory. If this option is specified, then you can not revup an old EDK project (XMP) at the same time.

Limitations

The limitations of the revup32to61 are:

- If you have any IP in **myip** directory, even though revup will update the format of data files, you must manually move those IP to **pcores** directory. EDK 6.1 tools do not search for IPs in **myip** directory.
- If you have a EDK 3.1 project, the software repository revup does not happen automatically. If you your own MDD files, you must manually update them to 2.1.0 format. A manual update of MDD files was required even when reving up from EDK 3.1 to EDK 3.2.

Chapter 10

Bitstream Initializer

This chapter describes the Bitstream Initializer (BitInit) utility. The chapter contains the following sections.

- "Overview"
- "Tool Usage"
- "Tool Options"

Overview

The Bitstream Initializer tool initializes the instruction memory of processors on the FPGA. The instruction memory of processors are stored in BlockRAMs in the FPGA. This utility reads an MHS file, and invokes the Data2MEM utility provided in ISE to initialize the FPGA BlockRAMs.

Tool Usage

The BitInit tool is invoked as follows:

```
bitinit <mhsfile> [options]
```

Note: Please specify <mhsfile> before specifying other tool options.

Tool Options

The following options are supported in the current version of BitInit:

-h (Display Help)

The **-h** option displays the usage menu and quits.

-v (Display Version)

The **-v** option displays the version and quits.

-bm (Input BMM file)

The **-bm** option specifies the input BMM file which contains the address map and the location of the instruction memory of the processor.

Default: implementation/<sysname>_bd.bmm

-bt (Bitstream file)

The **-bt** option specifies the input bitstream file that does not have it's memory initialized.

Default: implementation/<sysname>.bit

-o (Output Bitstream file)

The **-o** option specifies the name of the output file to generate the bitstream with initialized memory.

Default: implementation/download.bit

-pe (Specify the Processor Instance name and list of elf files)

The **-pe** option specifies the name of the processor instance in the MHS and it's associate list of ELF files that form it's instruction memory. This option may be repeated several times based on the number of processor instances in the design.

-lp (Libraries path)

The **-lp** option specifies the path to repository libraries. This option may be repeated to specify multiple libraries.

-log (Log file name)

The **-log** option specifies the name of log file to capture the log.

Default: bitinit.log

-quiet

Runs the tool in quiet mode.

Note: The tool also produces a file named "data2mem.dmr" that is the log file generated during invocation of the Data2MEM utility.

Chapter 11

Programming Flash Memory

Overview

The **Program Flash Memory** dialog allows users to program external parallel flash parts on their board, connected through the *opb_emc/plb_emc* external memory controller IP cores. The programming is done using a small in-system flash writing program, that executes on the target processor of your design. A host TCL script, drives the in-system flashwriter with commands and data and completes the flash programming. The dialog does not process/interpret the image file to be programmed and routinely programs the file as-is onto flash memory. The user's software and hardware application setup, must infer the contents of the flash as desired.

Prerequisites

The dialog must be invoked from within XPS having the EDK project that has the *OPB/PLB* EMC interface to the flash part, being open. The dialog assumes that your hardware design has correct connections to the flash part and at least one processor in the user's design is interfaced to the EMC controller. Since the dialog works by downloading and executing a flashwriter program, the user must make sure that the target board is connected to the host via JTAG and the FPGA is initialized with the bitstream of the project. The dialog also requires that the **Debug Settings Dialog** has been used to specify the up-to-date debugger information about the design. The flashwriter requires at least 32KB of free space to execute from in the user's design. The flashwriter can also work in batch or one-shot mode, by storing the entire image file to be programmed into memory and then quickly iterating through it and writing to flash. This mode can be useful for programming large images, say of size 1 MB or greater. The one-shot mode also greatly improves the programming speed of the flashwriter.

Supported Flash Hardware

The flashwriter program uses the *"Compact Flash Interface"* (CFI) to query the flash parts. Hence it requires that the flash part be CFI compliant. Table 11-1 lists the flash configurations that are supported.

Table 11-1: **Supported Flash Configurations**

Supported Flash Configurations
Single 16-bit device forming a 16-bit data bus
Paired 8-bit devices forming a 16-bit data bus
Single 32-bit device forming a 32-bit data bus

Table 11-1: **Supported Flash Configurations *(Continued)***

Paired 16-bit devices forming a 32-bit data bus
Four 8-bit devices forming a 32-bit data bus

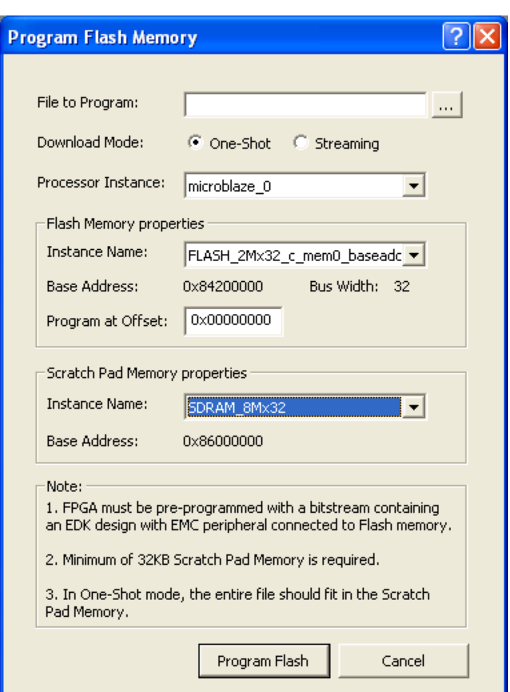
The above physical layout, geometry information and other logical information such as command sets understood are determined by using the CFI. Currently, the flashwriter can handle flash parts that can understand the CFI defined command sets listed in Table 11-2.

Table 11-2: **CFI Defined Command Sets**

Vendor ID	OEM Sponsor	Interface Name
1	Intel/Sharp	Intel/Sharp Extended Command Set
2	AMD/Fujitsu	AMD/Fujitsu Standard Command Se
3	Intel	Intel Standard Command Set
4	AMD/Fujitsu	AMD/Fujitsu Extended Command Set

Using the Program Flash Memory Dialog

Click on **Tools -> Program Flash Memory** to reach the flash programmer dialog. This dialog is shown below,



The user needs to enter the following information in the flash programmer dialog.

File to Program

Choose the file that you wish to program onto the flash part, by browsing to the file and selecting it, or entering the file's path in the text box.

Download Mode

Select from **one-shot** or **streaming**. One-shot mode requires that you have enough memory to accommodate both the flashwriter program and the image file. i.e at least *(32K + size of the image file)* bytes of memory free. If choosing streaming mode, then you require at least 32K of free memory.

Processor Instance

Choose the processor that is connected to your flash parts via an EMC controller. This processor will be used to execute the flashwriter program.

Flash Memory Properties

Instance Name

Choose the instance name of the memory controller that interfaces to the flash parts on your target board.

Program At Offset

Choose an offset within the flash to start programming the image file to. If you want to program different file images at different parts of the flash, you would be changing this parameter each time to choose a different position within the flash to program the file.

Scratch Pad Memory Properties

Instance Name

Choose the instance name of the memory controller that connects to the free scratch pad memory, that you wish to use for storing the flashwriter (and the image file if one-shot programming mode). Note that it must satisfy size constraints as described earlier. Please do not select the same memory controller as that of the Flash Memory.

Program Flash

Click on the **Program Flash** button to start the flashwriter. The dialog launches XMD in the XPS console. The rest of the algorithm proceeds from the XMD console. Click **Cancel**, if you wish to cancel your programming session.

Customizing Flash Programming

The flash programming setup that has been described above, might not fit your requirements exactly - there could be hardware incompatibilities, flash command set incompatibilities, memory size constraints etc. This section briefly describes the internals of the flash programming algorithm. Knowing this, you can plug in and replace pieces of the flow to customize it for your particular setup.

When you click on the Program flash button, then the following sequence of events happen.

1. A *flash_params.tcl* file is written out to the etc/ folder. This contains parameters describe the flash programming session and is used by the flashwriter TCL.

2. XPS launches XMD with the flashwriter TCL script executing on it with a command such as, *xmd -tcl flashwriter.tcl.* This flashwriter host TCL comes from the installation. If you wish to run your own driver TCL when the **Program Flash** button is clicked, place a copy of the *flashwriter.tcl* file, in your project's root directory. XMD searches for the specified file to be present in your project directory first, before looking for it in the repository.
3. The flashwriter TCL script, copies over the flashwriter application's source files from the installation to the etc/flashwriter folder. It compiles the application locally to execute out of the scratch memory address that was chosen. Here again, if you wish to compile your own flash writer sources, then you would modify your local copy of the *flashwriter.tcl* script to compile your own sources instead of the sources from the installation.
4. The script downloads the flashwriter to the processor. The script communicates with the flashwriter program, through mailboxes in memory. In other words, it writes parameters to the memory locations corresponding to variables in the flashwriter address space and lets the flashwriter execute.
5. The script waits for the flashwriter to invoke a callback function at the end of each operation and stops the application at the callback function by setting a breakpoint at the function. Once the flashwriter is stopped, the host TCL processes the results and then continues with more commands as required.
6. While programming, the flashwriter erases as only many flash blocks as required, to store the image in.
7. If programming in one shot, the TCL downloads the entire image to memory and lets the flashwriter complete the programming operation. If programming in streaming mode, it iteratively streams each block of the image file and lets the flashwriter program the flashpart in chunks. It stores these chunks in a memory buffer located within the flashwriter.
8. Once the programming is done, the flashwriter TCL sends an exit command to the flashwriter and terminates the XMD session.

Using Flash Memory

Typically, you can program three different kinds of things in flash

- Executable/bootable images of applications
- Hardware bitstreams for your FPGA
- File system images, data files such as sample data, algorithmic tables etc.

The first use case is most common. When the processor on your design comes out of reset, it will start executing code stored in BRAM at the processor's reset location. Typically, BRAM is too small (in the order of a few kilobytes) to accommodate your entire software application image. Therefore, you can store your software application image in flash memory (typically in the order of megabytes). A small bootloader is then designed to fit in BRAM and upon reset, start reading the software application image from flash memory, copy it over to larger and more available external memory and then transfer control to your software application, which then continues.

Your software application that you build out of your project is in executable ELF format. Storing and bootloading the ELF image itself is not usually done. This is because, bootloading an ELF image, increases the complexity of the bootloader. Instead, this ELF

image is converted to one of the common bootloadable image formats, such as SREC or IHEX. The bootloader then becomes very simple and thus smaller.

Sample Bootloader

To help you get quickly started with bootloading your software application from flash, a simple bootloader is provided with EDK. It is capable of booting an image file which is in the SREC format (Motorola™ S-record format), given the location of the image in some memory. This bootloader has been designed to obtain the image file from flash memory, which is expected to have been programmed with the image, prior to invoking the bootloader.

Here are the steps you need to follow to include this simple bootloader in your design:

1. Create a folder such as *bootloader*, within your EDK project.
2. Copy over all the source files *(*.c *.h)* from the *<edk_install>/data/xmd/bootloader/src* folder to the local folder which you created for storing the bootloader files, where *edk_install* is the root of your EDK installation.
3. Create a new software application project in XPS. Include the local copy of the source files as the source files of the project.
4. Set compiler flags and other compilation parameters as you require and wish.
5. Modify the C define - *FLASH_IMAGE_BASEADDR* in the *blconfig.h* header file to point to the memory location from which the bootloader has to pick up the flash image from. For example, if you have stored your executable image at absolute address 0xFF800000 in the flash, then modify define so that it becomes,

   ```
   #define FLASH_IMAGE_BASEADDR 0xFF800000
   ```
6. Build your project and include the bootloader in your bitstream. The next time your bitstream is loaded onto the FPGA, then your bootloader will start loading the image that you have stored in flash.

You can also subsequently modify these sources to adapt the bootloader for any specific scenario that you might require it for.

Here is how you create an SREC image of your software application. Lets assume that your final software application image is named, *myexecutable.elf*. In the console of your operating system, (*xygwin* on windows platforms), navigate to the folder containing this ELF file and key in,

```
<platform>-objcopy -O srec myexecutable.elf myexecutable.srec
where platform is,
  powerpc-eabi if your processor is PPC405
  mb if your processor is Microblaze
```

This creates an SREC file, which you can supply to the **Program Flash Dialog**, to be written onto the flash memory. *mb-objcopy* and *powerpc-eabi-objcopy* are GNU binary utilities that ship with EDK.

Chapter 12

GNU Compiler Tools

This chapter describes the various options supported by MicroBlaze and PowerPC GNU tools. The MicroBlaze GNU tools include **mb-gcc** compiler, **mb-as** assembler and **mb-ld** loader/linker. The PowerPC tools include **powerpc-eabi-gcc** compiler, **powerpc-eabi-as** assembler and the **powerpc-eabi-ld** linker. The EDK GNU tools also support C++.

This chapter discusses only those options which have been added or enhanced for the Embedded Development Kit (EDK). The chapter contains the following sections.

- "GNU Compiler Framework"
- "Compiler Usage and Options"
- "File Extensions"
- "Compiler Interface"
- "MicroBlaze GNU Compiler"
- "PowerPC GNU Compiler"

GNU Compiler Framework

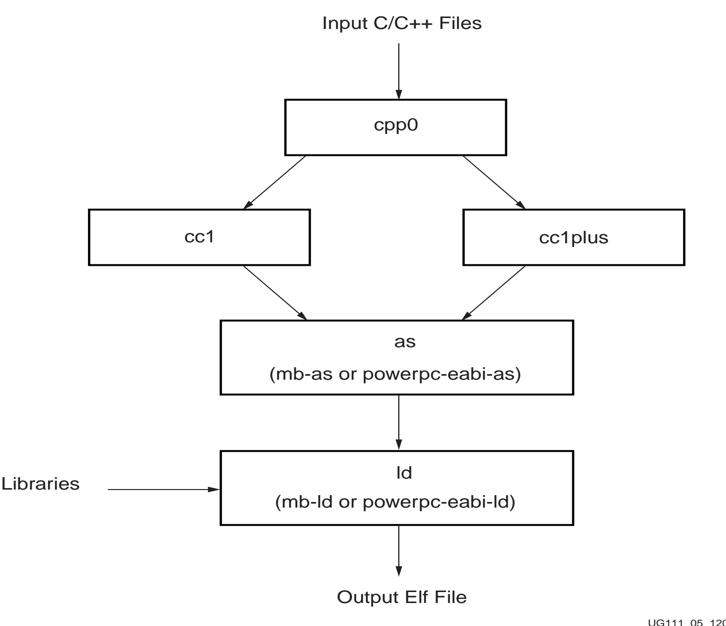


Figure 12-1: **GNU Tool Flow**

This section discusses the common features of both the MicroBlaze as well as PowerPC compiler. Figure 12-1 shows the GNU tool flow. The GNU compiler is named **mb-gcc** for **MicroBlaze** and **powerpc-eabi-gcc** for **PowerPC**. The GNU compiler is a wrapper which in turn calls four different executables:

1. Pre-processor: (**cpp0**)
   - This is the first pass invoked by the compiler.
   - The pre-processor replaces all macros with definitions as defined in the source and header files.
2. Machine and Language specific Compiler (**cc1)**
   - The compiler works on the pre-processed code, which is the output of the first stage.

   a. C Compiler (**cc1**)
   - The compiler is responsible for most of the optimizations done on the input C code and generates an assembly code.

   b. C++ Compiler (**cc1plus**)
   - The compiler is responsible for most of the optimizations done on the input C++ code and generates an assembly code.
3. Assembler (**mb-as [***For MicroBlaze***]** and **powerpc-eabi-as [***for PowerPC***]**)
   - The assembly code has mnemonics in assembly language.The assembler converts these to machine language.
   - The assembler also resolves some of the labels generated by the compiler.
   - The assembler creates an object file, which is passed on to the linker
4. Linker (**mb-ld [***For MicroBlaze***]** and **powerpc-eabi-ld [***for PowerPC***]**)
   - The linker links all the object files generated by the assembler.
   - If libraries are provided on the command line, the linker resolves some of the undefined references in the code, by linking in some of the functions from the assembler.

Options for all these executables in discussed in this chapter.

Note: Any reference to gcc in this chapter indicates reference to both MicroBlaze compiler (**mb-gcc**) as well as PowerPC compiler (**powerpc-eabi-gcc**).

Compiler Usage and Options

Usage

GNU Compiler usage is as follows

```
Compiler_Name [options] files...
```

Where `Compiler_Name` is **powerpc-eabi-gcc** or **mb-gcc**

Quick Reference

Table 12-1 briefly describes the commonly used compiler options. These options are common to both the compilers, i.e MicroBlaze and PowerPC. Please note that the compiler options are case sensitive.

Table 12-1: **Commonly Used Compiler Options**

Options	Explanation
-E	Preprocess only; Do not compile, assemble and link. The preprocessed output is displayed on the standard out device
-S	Compile only; Do not assemble and link (Generates **.s** file)
-c	Compile and Assemble only; Do not link (Generates **.o** file)
-g	Add debugging information, which is used by GNU debugger (**mb-gdb** or **powerpc-eabi-gdb**)
-gstabs	Add debugging information to the compiled assembly file. Pass this option directly to the GNU assembler or through the -**Wa** option to the Compiler
-**Wa,***option*	Pass comma-separated *options* to the assembler
-**Wp,***option*	Pass comma-separated *options* to the preprocessor
-**Wl,***option*	Pass comma-separated *options* to the linker
-B *directory*	Add *directory* to the C-run time library search paths
-L *directory*	Add *directory* to library search path
-I *directory*	Add *directory* to header search path
-l *library*	Search *library*[a] for undefined symbols.
-v	(Verbose). Display the programs invoked by the compiler
-o *filename*	Place the output in the *filename*
-save-temps	Store the intermediate files, i.e files produced at the end of each pass,
--help	Display a short listing of options.
-O *n*	Specify Optimization level *n = 0,1,2,3*

a. The compiler prefixes “lib” to the library name indicated in this command line switch.

Compiler Options

Some of the compiler options are discussed in details in this section

-g

This option adds debugging information to the output file. The debugging information is required by the GNU Debugger (**mb-gdb** or **powerpc-eabi-gdb**). The debugger provides debugging at the source as well as the assembly level. This option adds debugging information only when the input is a C/C++ source file.

-gstabs

Use this option for adding debugging symbols to assembly(.S) files. This is a assembler option and should be provided directly to the GNU assembler (**mb-as** or **powerpc-eabi-as**). If an assembly file is compiled using the compiler (**mb-gcc** or **powerpc-eabi-gcc**), prefix the option with -**Wa,** .

-O*n*

The GNU compiler provides optimizations at different levels. These optimization levels are applied only to the C and C++ source files.

Table 12-2: **Optimizations for Different Values of n**

n	Optimization
0	No Optimization
1	Medium Optimization
2	Full optimization
3	full optimization, and also attempt automatic inlining of small subprograms.
S	Optimize for speed

Note: Optimization levels 1 and above will cause code re-arrangement. While debugging your code, use of no optimization level is advocated. When an optimized program is debugged through gdb, the displayed results might seem inconsistent.

-v

This option executes the compiler and all the tools underneath the compiler in verbose mode. This option gives complete description of the options passed to all the tools. This description is helpful in finding out the default options for each tool.

-save-temps

The GNU compiler provides a mechanism to save all the intermediate files generated during the compilation process. The compiler stores the following files

- Preprocessor output (*input_file_name.i* for C code and *input_file_name.ii* for C++ code)
- Compiler (cc1) output in assembly format (*input_file_name.s*)
- Assembler output in elf format (*input_file_name.s*)

The default output of the entire compilation is stored as *a.out*.

-o *Filename*

The default output of the compilation process is stored in an elf file name *a.out*. The default name can be changed using the *-o output_file_name*. The output file is created in elf format.

-Wp,*option*

-Wa,*option*

-Wl,*option*

As described earlier in this chapter, the compiler (**mb-gcc** or **powerpc-eabi-gcc**) is a wrapper around other executables such as the preprocessor, compiler (cc1), assembler and the linker. These components of the compiler can be executed through the top level compiler or individually.

There are certain options which are required by tool, but might not be necessary for the top level compiler. These command can be issues using the options as indicated in Table 12-3

Table 12-3: **Tool-Specific Options Passed to the Top-Level GCC Compiler**

Option	Tool
-**Wp**,*option*	Preprocessor
-**Wa**,*option*	Assembler
-**Wl**,*option*	Linker

--help

Use this option with any GNU compiler to get more information about the available options or consult the GCC manual available online at this location:

http://www.gnu.org/manual/manual.html

Library Search Options

-l *libraryname*

The compiler, by default, searches only the standard libraries such as libc, libm and libxil. The users can create their own libraries containing some commonly used functions. The users can indicate to the compiler, the name of the library, where the compiler can find the definition of these functions. The compiler prefixes the word **"lib"** to the *libraryname* provided by the user.

The compiler is sensitive to the order in which the various options are provided, especially the -**l** command line switch. This switch should be provided only after all the sources in the command line.

For example, if a user creates his own library called **libproject.a.**, he/she can include functions from this library using the following command:

```
Compiler Source_Files -L${LIBDIR} -lproject
```

Caution! If the library flag **-llibrary name** is given before the source files, the compiler will not be able to find the functions called from any of the sources. The compiler search is only done in one direction and does not keep a list of libraries available.

-L *Lib Directory*

This option indicates to the compiler, the directories to search for the libraries. The compiler has a default library search path, where it looks for the standard library. By providing -**L** option, the user can include some additional directories in the compiler search path.

Header File Search Option

-I *Directory Name*

The option -**I**, indicates to the compiler to search for header files in the directory *Directory Name* before searching the header files in the standard path.

Linker Options

-defsym _STACK_SIZE=*value*

The total memory allocated for the stack and the heap can be modified by using the above linker option. The variable STACK_SIZE is the total space allocated for heap as well as the stack. The variable STACK_SIZE is given the default value of 100 words (i.e 400 bytes). If any user program is expected to need more than 400 bytes for stack and heap together, it is recommended that the user should increase the value of STACK_SIZE using the above option. This option expects value in **bytes**.

In certain cases, a program might need a bigger stack. If the stack size required by the program is greater than the stack size available, the program will try to write in other forbidden section of the code, leading to wrong execution of the code.

Note: For MicroBlaze systems, minimum stack size of 16 bytes (0x0010) is required for programs linked with the C runtime routines (crt0.o and crt1.o).

Linker Scripts

The linker utility makes use of the linker scripts to divide the user's program on different blocks of memories. To provide a linker script on the gcc command line, use the following command line option:

```
<compiler> -Wl,-T -Wl,linker_script <Other Options and Input Files>
```

If the linker is executed on its own, the linker script could be included as follows:

```
<linker> -T linker_script <Other Options and Input Files>
```

For more information about usage of linker scripts, please refer to Chapter 4, "Address Management," in the *Platform Studio User Guide*.

Search Paths

The compilers (**mb-gcc** and **powerpc-eabi-gcc**) search certain paths for libraries and header files.

On Solaris

Libraries are searched in the following order:

1. Directories passed to the compiler with the **`-L dir name`** option.
2. Directories passed to the compiler with the **`-B dir name`** option.
3. `${XILINX_EDK}/gnu/processor[1]/sol/microblaze/lib`
4. `${XILINX_EDK}/lib/processor`

Header files are searched in the following order:

1. Directories passed to the compiler with the -**I dir name** option.$
2. `${XILINX_EDK}/gnu/processor/sol/processor/include`

Initialization files are searched in the following order[2]:

1. Directories passed to the compiler with the -**B dir name** option.

1. Processor indicates **powerpc-eabi** for PowerPC and **microblaze** for MicroBlaze

2. **${XILINX_EDK}**/gnu/*processor*/sol/*processor*/lib

On Windows Xygwin Shell

The GNU compilers (**mb-gcc** and **powerpc-eabi-gcc)** search certain paths for libraries and header files.

Libraries are searched in the following order:

1. Directories passed to the compiler with the **-L dir name** option.
2. Directories passed to the compiler with the **-B dir name** option.
3. **%XILINX_EDK%**/gnu/*processor*/nt/*processor*/lib
4. **%XILINX_EDK%**/lib/*processor*

Header files are searched in the following order:

1. Directories passed to the compiler with the -**I dir name** option.$
2. **%XILINX_EDK%**/gnu/*processor*/nt/*processor*/include

Initialization files are searched in the following order:

1. Directories passed to the compiler with the -**B dir name** option.
2. %XILINX_EDK%/gnu/*processor*/nt/*processor*/lib

File Extensions

File Types and Extensions

The GNU compiler can determine the type of your file depending on the extension.Table 12-4 illustrates the valid extension and the corresponding file type.The gcc wrapper will call the appropriate lower level tools by recognizing these file types.

Table 12-4: **File Extensions**

Extension	File type
.c	C File
.C	C++ File
.cxx	C++ File
.cpp	C++ File
.c++	C++ File
.cc	C++ File
.S	Assembly File, but might have preprocessor directives
.s	Assembly File with no preprocessor directives

2. Initialization files such as crt0.o are searched by the compiler only for **mb-gcc.** For **powerpc-eabi-gcc**, the C runtime library is a part of the library and is picked up by default from the library *libxil.a*

Libraries

Both the compiler (**powerpc-eabi-gcc** and **mb-gcc**) use certain libraries. The following libraries are needed for all the program.

Table 12-5: **Libraries Used by the Compilers**

Library	Particular
libxil.a	Contain drivers, software services (such as XilNet & XilMFS) and initialization files developed for the EDK tools
libc.a	Standard C libraries, including functions like **strcmp, strlen** etc
libm.a	Math Library, containing functions like **cos, sine** etc

All the libraries are linked in automatically by both the compiler. The search path for these libraries might have to be given to the compiler, if the standard libraries are overridden. The libxil.a is modified by the Library Generator tool to add driver and library routines.

Compiler Interface

Input Files

The compiler (mb-gcc and the powerpc-eabi-gcc) take one or more of the following files are input

- C source files.
- C++ source files.
- Assembly Files.
- Object Files.
- Linker scripts (These are optional and if not specified, the default linker script embedded in the linker (mb-ld or powerpc-eabi-ld) will be used.

The default extensions for each of these types is detailed in Table 12-4. In addition to the files mentioned above, the compiler implicitly refers to the following files.

- Libraries (libc.a, libm.a and libxil.a). The default location for these files is the EDK installation directory.

Output Files

The compiler generates the following files as output

- An elf file (The default output file name is **a.out** on Solaris and **a.exe** on Windows)
- Assembly file (if -save-temps or -S option is used)
- Object file (if -save-temps or -c option is used)
- Preprocessor output (.i or .ii file) (if -save-temps option is used)

MicroBlaze GNU Compiler

The MicroBlaze GNU compiler is an enhancement over the standard GNU tools and hence provides some additional options, which are specific to the MicroBlaze system.These options are available only in the MicroBlaze GNU compiler.

Quick Reference

Table 12-6: **MicroBlaze-Specific Options**

Options	Explanation
-xl-mode-executable	Default mode for compilation.
-xl-mode-xmdstub	Software intrusive debugging on the board. Should be used only with xmdstub downloaded on to MicroBlaze
-xl-mode-xilkernel	Use this option to compile "ELF processes" that execute on Xilkernel. Refer to the Xilkernel reference guide for more information on Xilkernel ELF processes. You do not need this option for non-xilkernel based executables.
-mxl-gp-opt	Use the small data area anchors. Optimization for performance and size.
-mxl-soft-mul	Use the software routine for all multiply operations. This option should be used for devices without the hardware multiplier. This is the default option in mb-gcc
-mno-xl-soft-mul	Do not use software multiplier. Compiler generates "mul" instructions.
-mxl-soft-div	Use the software routine for all divide operations.This is the default option.
-mxl-no-soft-div	Use the hardware divide available in the MicroBlaze
-mxl-stack-check	Generates code for checking stack overflow.
-mxl-barrel-shift	Use barrel shifter. Use this option when a barrel shifter is present in the device

MicroBlaze Compiler

The mb-gcc compiler for Xilinx's MicroBlaze soft processor introduces some new options as well as modifications to certain options supported by the gnu compiler tools. The new and modified options are summarized in this chapter.

-mxl-soft-mul

In some devices, a hardware multiplier is not present. In such cases, the user has the option to either build the multiplier in hardware or use the software multiplier library routine provided. MicroBlaze compiler mb-gcc assumes that the target device does not have a hardware multiplier and hence every multiply operation is replaced by a call to **mulsi3_proc** defined in library **libc.a**. Appropriate arguments are set before calling this routine.

-mno-xl-soft-mul

Certain devices such as Virtex II have a hardware multiplier integrated on the device. Hence the compiler can safely generate the **mul** or **muli** instruction. Using a hardware multiplier gives better performance, but can be done only on devices with hardware multiplier such as Virtex II.

-mxl-soft-div

The MicroBlaze processor does not come with a hardware divide unit. The users would need the software routine in the libraries for the divide operation. This option is turned on by default in mb-gcc.

-mno-xl-soft-div

In MicroBlaze version 2.00 and beyond, the user can instantiate a hardware divide unit in MicroBlaze. If such a unit is present, this option should be provided to mb-gcc compiler. Refer to the MicroBlaze Reference Guide for more details about the usage of hardware divide option in the MicroBlaze.

-mxl-stack-check

This option lets users check if the stack overflows during the execution of the program. The compiler inserts code in the prologue of the every function, comparing the stack pointer value with the available memory. If the stack pointer exceeds the available free memory, the program jumps to a the subroutine **`_stack_overflow_exit`**. This subroutine sets the value of the variable **`_stack_overflow_error`** to 1.

The standard stack overflow handler can be overridden by providing the function **`_stack_overflow_exit`** in the source code, which acts as the stack overflow handler.

-mxl-barrel-shift

The MicroBlaze processor can be configured to be built with a barrel shifter. In order to use the barrel shift feature of the processor, use the option **`-mxl-barrel-shift`**. The default option is to assume that no barrel shifter is present and hence the compiler will use add and multiply operations to shift the operands. Barrel shift can increase the speed significantly, especially while doing floating point operations.Refer to the MicroBlaze Reference Guide for more details about the usage of the barrel shifter option in the MicroBlaze.

-mxl-gp-opt

If the memory location requires more than 32K, the load/store operation requires two instructions. MicroBlaze ABI offers two global small data areas, which can contain up to 64K bytes of data each. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value. Hence needing only one instruction for load/store to the small data area.This optimization can be turned ON with the -mxl-gp-opt command line parameter. Variables of size lesser than a certain threshold value are stored in these areas. The value of the pointers is determined during linking.

-xl-mode-executable

This is the default mode used for compiling programs with mb-gcc. The final executable created starts from address location 0x0 and links in crt0.o. This option need not be provided on the command line for mb-gcc.

-xl-mode-xmdstub

Xilinx Microprocessor Debugger (XMD) allows three different modes of debugging an user program for MicroBlaze. The three debugging options are

- Simulator mode (Does not require a board)

- XMDStub mode (Requires the XMDStub to be a part of the bitstream)
- MDM mode (Hardware debugging enabled. Bitstream does not contain the XMDStub)

For more information about the XMD tool, refer to the "Xilinx Microprocessor Debugger (XMD)" chapter in the guide.

For programs compiled with the "XMDStub" mode, the address locations 0x0 to 0x3ff are reserved for the XMDStub. Hence the user program can start only at 0x800.

The usage of -xl-mode-xmdstub has two effects:

- The start address of the user program is set to 0x800. Users can change this address by overriding the _TEXT_START_ADDR in the linker script or through linker options. For more details about linker options, refer to the"Linker Options" section. If the start address is defined to be less than 0x800, XMD issues an address overlap error.
- crt1.o is used as the initialization file. The crt1.o file returns the control back to the XMDStub when the user program execution is complete.

Note: **-xl-mode-xmdstub** should be used for designs when XMDStub is part of the bitstream. This mode should not be used when the system is complied for No Debug or when "Hardware Debugging" is turned ON. For more details on debugging with xmd, please refer to Chapter 14, "Xilinx Microprocessor Debugger (XMD)".

-xl-mode-xilkernel

The Embedded Development Kit provides a microkernel (XMK). Any application which needs to be executed on top of this kernel should be compiled with the **-xl-mode-xilkernel**. Refer to the EST Libraries Guide for more information regarding the various option provided by the Xilinx MicroKernel.

> ***Caution!*** mb-gcc will signal fatal error if more than one mode of execution is supplied on the command line.

MicroBlaze Assembler

The mb-as assembler for Xilinx's MicroBlaze soft processor supports the same set of options supported by the standard GNU compiler tools. It also supports the same set of assembler directives supported by the standard gnu assembler.

The mb-as assembler supports all the opcodes in the MicroBlaze machine instruction set, with the exception of the imm instruction. The mb-as assembler generates `imm` instructions when large immediate values are used. The assembly language programmer is never required to write code with imm instructions. For more information on the MicroBlaze instruction set, refer to the MicroBlaze Reference Guide.

The mb-as assembler requires all Type B MicroBlaze instructions (instructions with an immediate operand) to be specified as a constant or a label. If the instruction requires a PC-relative operand, then the mb-as assembler will compute it, and will include an imm instruction if necessary. For example, the Branch Immediate if Equal (beqi) instruction requires a PC-relative operand. The assembly programmer should use this instruction as follows:

```
beqi r3, mytargetlabel
```

where `mytargetlabel` is the label of the target instruction. The mb-as assembler computes the immediate value of the instruction as `mytargetlabel - PC`. If this immediate value is greater than 16 bits, the mb-assembler automatically inserts an imm instruction. If the value of `mytargetlabel` is not known at the time of compilation, the mb-as assembler

always inserts an imm instruction. The `relax` option of the linker should be used to remove any imm instructions that are found to be unnecessary.

Similarly, if an instruction needs a large constant as an operand, the assembly language programmer should use the operand as-is, without using an imm instruction. For example, the following code is used to add the constant 200,000 to the contents of register r3, and store the result in register r4:

```
addi r4, r3, 200000
```

The mb-assembler will recognize that this operand needs an imm instruction, and insert one automatically.

In addition to the standard MicroBlaze instruction set, the mb-as assembler also supports some pseudo-opcodes to ease the task of assembly programming. The supported pseudo-ops are listed in Table 12-7.

Table 12-7: **Pseudo-Opcodes Supported by the Gnu Assembler**

Pseudo Opcodes	Explanation
nop	No operation. Replaced by instruction: **or** R0, R0, R0
la Rd, Ra, Imm	Replaced by instruction: **addik** Rd, Ra, imm; = Rd = Ra + Imm;
not Rd, Ra	Replace by instruction: **xori** Rd, Ra, -1
neg Rd, Ra	Replace by instruction: **rsub** Rd, Ra, R0
sub Rd, Ra, Rb	Replace by instruction: **rsub** Rd, Rb, Ra

MicroBlaze Linker

The mb-ld linker for Xilinx's MicroBlaze soft processor introduces some new options in addition to those supported by the gnu compiler tools. The new options are summarized in this section.

-defsym _TEXT_START_ADDR=*value*

By default, the text section of the output code starts with the base address 0x0. This can be overridden by using the above options. If this is supplied to **mb-gcc**, the text section of the output code will now start from the given *value*. When the compiler is invoked with -**xl-mode-xmdstub**, the user program starts at 0x800 by default.

The user does not have to use -**defsym _TEXT_START_ADDR**, if they wish to use the default start address set by the compiler.

This is a linker option and should be used when the user is invoking the linker separately. If the linker is being invoked as a part of the **mb-gcc** flow, the user has to use the following option

```
-Wl,-defsym -Wl,_TEXT_START_ADDR=value
```

-relax

This is a linker option, used to remove all the unwanted **imm** instructions generated by the assembler. The assembler generates **imm** instruction for every instruction where the value

of the immediate can not be calculated during the assembler phase. Most of these instructions won't need an **imm** instruction. These are removed by the linker when the -**relax** command line option is provided to the linker.

This option is required only when linker is invoked on its own. When linker is invoked through the **mb-gcc** compiler, this option is automatically provided to the linker.

-N

This option sets the text and data section to be readable and writable. It also does not page-align the data segment. This option is required only for MicroBlaze programs. The top level gcc compiler automatically includes this option, while invoking the linker, but if you intend to invoke the linker without using gcc, you should have use this option.

For more details on this option, please refer to the GNU manuals online at this location:

http://www.gnu.org/manual/manual.html.

Initialization Files

The final executable needs certain registers such as the small data area anchors (R2 and R13) and the stack pointer (R1) to be initialized. These C-Runtime files also set up the interrupt and exception handler routines.

These initialization files are distributed with the Embedded Development Kit. In addition to the precompiled object files, source files are also distributed in order to help user make their own changes as per their requirements. Initialization can be done using one of the three C runtime routines:

crt0.o

This initialization file is to be used for programs which are to be executed standalone, i.e without the use of any bootloader or debugging stub (such as **xmdstub**).

crt1.o

This file is located in the same directory and should be used when software intrusive debugging (**XMDstub**) is used. **crt1.o** returns the control of the program back to the XMDStub on completion of user program.

crt4.o

Xilkernel supports creating processes out of separate ELF files. For these separate ELF files, a special CRT is required to ensure that the run-time initialization performed by the kernel is not overwritten. Therefore, when compiling such ELF files, crt4.o is used as the startup file by the compiler. **crt4.o** does not set up the interrupt and exception handlers since the default handling of the interrupts and exceptions are done by the kernel. This crt also return the control back to the kernel on completion of the user program.

The source for initialization files is available in the

<XILINX_EDK>/sw/lib/microblaze/src directory,

- *<XILINX_EDK>* : Installation area

These files can be changed as per the requirements of the project. These changed files have to be then assembled to generate an object file (.o format). To refer to the newly created

object files instead of the standard files, use the **`-B directory-name`** command line option while invoking **`mb-gcc`**.

According to the C standard specification, all global and static variables need to be initialized to 0. This is a common functionality required by all the crt's above. Hence another routine **_crtinit** is defined in **crtinit.o** file. This file is part of the **libc.a** library.

The **_crtinit** routine will initialize memory in the **bss** section of the program, defined by the default linker script. If you intend to provide your own linker script, you will need to compile a new **_crtinit** routine. The default **crtinit.S** file is provided in assembly source format as a part of the Embedded Development Kit.

Command Line Arguments

MicroBlaze programs can not take in command line arguments. The command line arguments argc and argv are initialized to 0 by the C runtime routines.

Interrupt Handlers

Interrupt handlers need to be compiled in a different manner as compared to the normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers have to save the volatile registers which are being used. Interrupt handler should also store the value of the machine status register (RMSR), when an interrupt occurs.

_interrupt_handler attribute

In order to distinguish an interrupt handler from a sub-routine, mb-gcc looks for an attribute (interrupt_handler) in the declaration of the code. This attribute is defined as follows:

```
void function_name () __attribute__ ((interrupt_handler));
```

Note: Attribute for interrupt handler is to be given only in the prototype and not the definition.

Interrupt handlers might also call other functions, which might use volatile registers. In order to maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function[1].

Interrupt handlers can also be defined in the MicroBlaze Hardware Specification (MHS) and the MicroBlaze Software Specification (MSS) file. These definitions would automatically add the attributes to the interrupt handler functions. For more information please refer MicroBlaze Interrupt Management document.

The interrupt handler uses the instruction `rtid` for returning to the interrupted function.

_save_volatiles attribute

The MicroBlaze compiler provides the attribute save_volatiles, which is similar to the _interrupt_handler attribute, but returns using `rtsd` instead of `rtid`.

This attributes save all the volatiles for non-leaf functions and only the used volatiles in case of leaf functions.

```
void function_name () __attribute__((save_volatiles));
```

---

1. Functions which have calls to other sub-routines are called non-leaf functions.

The attributes with their functions are tabulated in Table 12-8.

Table 12-8: **Use of Attributes**

Attributes	Functions
`interrupt_handler`	This attribute saves the machine status register and all the volatiles in addition to the non-volatile registers. **`rtid`** is used for returning from the interrupt handler. If the interrupt handler function is a leaf function, only those volatiles which are used by the function are saved.
`save_volatiles`	This attribute is similar to interrupt_handler, but it used **`rtsd`** to return to the interrupted function, instead of **`rtid`**.

PowerPC GNU Compiler

Compiler Options

The PowerPC GNU compiler **(powerpc-eabi-gcc)** is built using the GNU gcc version 2.95.3-4. No enhancements have been done to the compiler. The PowerPC compiler does not support any special options. All the listed common options are supported by the powerpc-eabi compiler. The PPC405 specific options that are supported by the compiler are,

-mhard-float

This option tells the compiler to produce code that uses the floating point registers and the floating point instructions native to the PPC405. If the PPC405 that you are targeting supports the native floating point instructions, then this option is required to leverage it. This is turned off by default.

-msoft-float

This option is enabled by default and tells the compiler to produce floating point that uses the software floating point libraries.

Linker Options

-defsym _START_ADDR=*value*

By default, the text section of the output code starts with the base address 0xffff0000, since this is the start address indicated in the default linker script. This can be overridden by

- using the above option OR
- providing a linker script, which lists the value for start address

The user does not have to use -**defsym _START_ADDR**, if they wish to use the default start address set by the compiler.

This is a linker option and should be used when the user is invoking the linker separately. If the linker is being invoked as a part of the **powerpc-eabi-gcc** flow, the user has to use the following option

```
-Wl,-defsym -Wl,_START_ADDR=value
```

Initialization Files

The compiler looks for certain initialization files (such as **boot.o, crt0.o**)**.** These files are compiled along with the drivers and archived in libxil.a library. This library is generated using LibGen by compiling the distributed sources in the Board Support Package (BSP). For more information about LibGen, refer to Chapter 7, "Library Generator".

Chapter 13

GNU Debugger

This chapter describes the general usage of the Xilinx GNU debugger for MicroBlaze and PowerPC. The chapter contains the following sections.

- "Overview"
- "MicroBlaze GDB Targets"
- "PowerPC Targets"
- "Console Mode"
- "GDB Command Reference"

Overview

GDB is a powerful yet flexible tool which provides a unified interface for debugging/verifying MicroBlaze and PowerPC systems during various development phases. It uses Xilinx Microprocessor Debugger (XMD) as the underlying engine to communicate to Processor targets.

Tool Usage

MicroBlaze GDB usage:

```
mb-gdb [options] [executable-file]
```

PowerPC GDB usage:

```
powerpc-eabi-gdb [options] [executable-file]
```

Tool Options

The most common options in the GNU debugger are:

--command=FILE

Execute GDB commands from FILE. Used for debugging in batch/script mode.

--batch

Exit after processing options. Used for debugging in batch/script mode.

--nw

Do not use a GUI interface.

-w

Use a GUI interface. (Default)

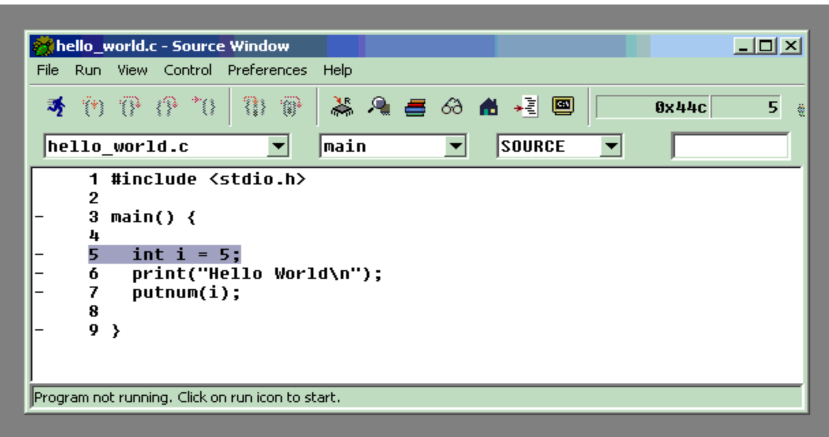
Debug Flow using GDB

1. Start XMD from XPS.
2. Connect to Processor target (Simulator/Hardware/Virtual Platform). Opens a GDB Server for the target.
3. Start GDB from XPS.
4. Connect to Remote GDB Server on XMD.
5. Download the Program and Debug application.

MicroBlaze GDB Targets

Currently, there are three possible remote targets that are supported by the MicroBlaze GNU Debugger and XMD tools.

```
xilinx > mb-gdb hello_world.elf
```

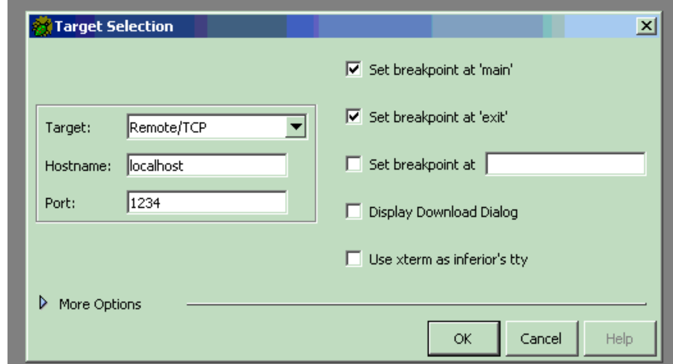


From the **Run** pull-down menu, select **Connect to target** in the mb-gdb window. In the Target Selection dialog, choose **Remote/TCP** targets.

In the target selection dialog, choose:

- **Target**: **Remote/TCP**
- **Hostname**: `localhost`
- **Port**: `1234`

Click **OK** and mb-gdb attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD was started.



At this point, **mb-gdb** is connected to XMD and controls the debugging. The simple but powerful GUI can be used to debug the program, read and write memory and registers.

Remote Targets

Remote debugging is done through XMD. The XMD server program can be started on a host computer with the Simulator target or the Hardware target or Virtual Platform target transparent to mb-gdb. The Cycle-Accurate Instruction Set Simulator and the Hardware interface provide powerful debugging tools for verifying a complete MicroBlaze system. **mb-gdb** connects to **xmd** using the GDB Remote Protocol over TCP/IP socket connection.

Simulator Target

The XMD simulator is a Cycle-Accurate Instruction Set Simulator of the MicroBlaze system which presents the simulated MicroBlaze system state to GDB.

Hardware Target

With the hardware target, XMD communicates with *opb_mdm* debug core or an *xmdstub program* running on a hardware board through the serial cable or JTAG cable, and presents the running MicroBlaze system state to GDB.

For more information about XMD refer to the XMD Chapter.

Virtual Platform Target

Virtual Platform is a Cycle-Accurate MicroBlaze fixed Reference design. It supports LMB and External Memory, UARTLite and GPIO interface.

For more information about Virtual Platform refer VP Reference Design.

Compiling for Debugging on MicroBlaze Targets

In order to debug a program, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code. The mb-gcc compiler for Xilinx's MicroBlaze soft processor includes this information when the appropriate modifier is specified.

The **-g** option in **mb-gcc** allows you to perform debugging at the source level. mb-gcc adds appropriate information to the executable file, which helps in debugging the code.

mb-gdb provides debugging at source, assembly and mixed (both source and assembly) together.

Note: While initially verifying the functional correctness of a C program, it is also advisable to not use any mb-gcc optimization option like -O2 or -O3 as mb-gcc does aggressive code motion optimizations which may make debugging difficult to follow.

Note: For debugging with **`xmd`** in hardware mode using XMDSTUB, the **`mb-gcc`** option **`-xl-mode-xmdstub`** must be specified. Refer to the XMD documentation for more information about compiling for specific targets.

PowerPC Targets

Debugging for the PowerPC405 is supported by **powerpc-eabi-gdb** and **xmd** through the GDB Remote TCP protocol. XMD supports two remote targets, *PowerPC Hardware* on (**Virtex-II Pro and Virtex4**) and *Cycle-Accurate PowerPC Instruction Set Simulator*.

To connect to a PowerPC target, first start **xmd** and connect to the board using the **connect ppc** command as described in the XMD chapter. Next, select **Run → Connect to target** from GDB and in the GDB target selection dialog, choose:

- Target: Remote/TCP
- Hostname: localhost
- Port: 1234

Click **OK** and **powerpc-eabi-gdb** attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD was started.

Console Mode

To start powerpc-eabi-gdb in the console mode type :

```
xilinx > powerpc-eabi-gdb -nw executable.elf
```

In the console mode, type the following two commands to connect to the board through xmd.

```
(gdb) target remote localhost:1234
(gdb) load
Loading section .text, size 0xfcc lma 0xffff8000
Loading section .rodata, size 0x118 lma 0xffff8fd0
Loading section .data, size 0x2f8 lma 0xffff90e8
Loading section .fixup, size 0x14 lma 0xffff93e0
Loading section .got2, size 0x20 lma 0xffff93f4
Loading section .sdata, size 0xc lma 0xffff9414
Loading section .boot0, size 0x10 lma 0xffffa430
Loading section .boot, size 0x4 lma 0xfffffffc
Start address 0xfffffffc, load size 5168
Transfer rate: 41344 bits/sec, 323 bytes/write.
(gdb) c
Continuing
```

For the consoles mode, these two commands can also be placed in the GDB startup file **`gdb.ini`** in the current working directory.

GDB Command Reference

For help on using mb-gdb, click on **Help** → **Help Topics** in the GUI mode or type "**help**" in the console mode.

In the GUI mode, to open a console window, click on **View** → **Console**

For a comprehensive online documentation on using GDB, refer to the GNU website.

For information about the mb-gdb Insight GUI, refer to the Red Hat Insight webpage at http://sources.redhat.com/insight.

Table 13-1 briefly describes the commonly used mb-gdb console commands. The

Table 13-1: **Commonly Used GDB Console Commands**

Command	Description
load [program]	load the program into the target
b main	Set a breakpoint in function main
c	Continue after a breakpoint ***Note:*** Run command should not be used.
l	View a listing of the program at the current point
n	Steps one line (stepping over function calls)
s	Step one line (stepping into function calls)
stepi	Step one assembly line
info reg	View register values
info target	View the number of instructions and cycles executed (for the built-in simulator only)
p *xyz*	Print the value of *xyz* data
hbreak main	Set Hardware breakpoint in function main
watch gvar1	Set Watchpoint on Global Variable gvar1
rwatch gvar1	Set Read Watchpoint on Global Variable gvar1

equivalent GUI versions can be easily identified in the mb-gdb GUI window icons. Some of the commands like info target, monitor info, may be available only in the console mode.

Chapter 14

Xilinx Microprocessor Debugger (XMD)

The Xilinx Microprocessor Debugger (XMD) is a tool that facilitates debugging programs and verifying systems using the PowerPC405GP (Virtex-II Pro & Virtex4) or MicroBlaze microprocessors. It supports debugging user programs running on a hardware board, Cycle-accurate Instruction Set Simulator (ISS) and MicroBlaze Cycle-accurate Virtual Platform (VP) system.

XMD provides a TCL (Tool Command Language) interface. This interface can be used for command line control and debugging of the target as well as for running complex verification test scripts to test complete system.

XMD also provides a TCP Socket based interface. This interface can be used by tools to execute Internal XMD commands. Refer to the "XMD TCP Socket Interface" section for details.

XMD supports GDB Remote TCP protocol to control debugging of a target. Graphical debugger's like PowerPC and MicroBlaze GDB (**powerpc-eabi-gdb & mb-gdb**) and **Platform Studio SDK** (Eclipse based Software IDE) use this interface for debugging. In either case, the debugger can connect to **xmd** running on the same computer or on a remote computer on the Internet.

XMD reads system files XMP/MHS/MSS to better understand the hardware system on which the program is debugged. The information is used to perform memory range test, determine Microblaze-MDM connectivity for faster download speeds and other system options.

This chapter contains the following sections.

- "XMD Usage"
- "XMD Command Reference"
- "Connect Command Options"
- "XMD Internal Tcl Commands"

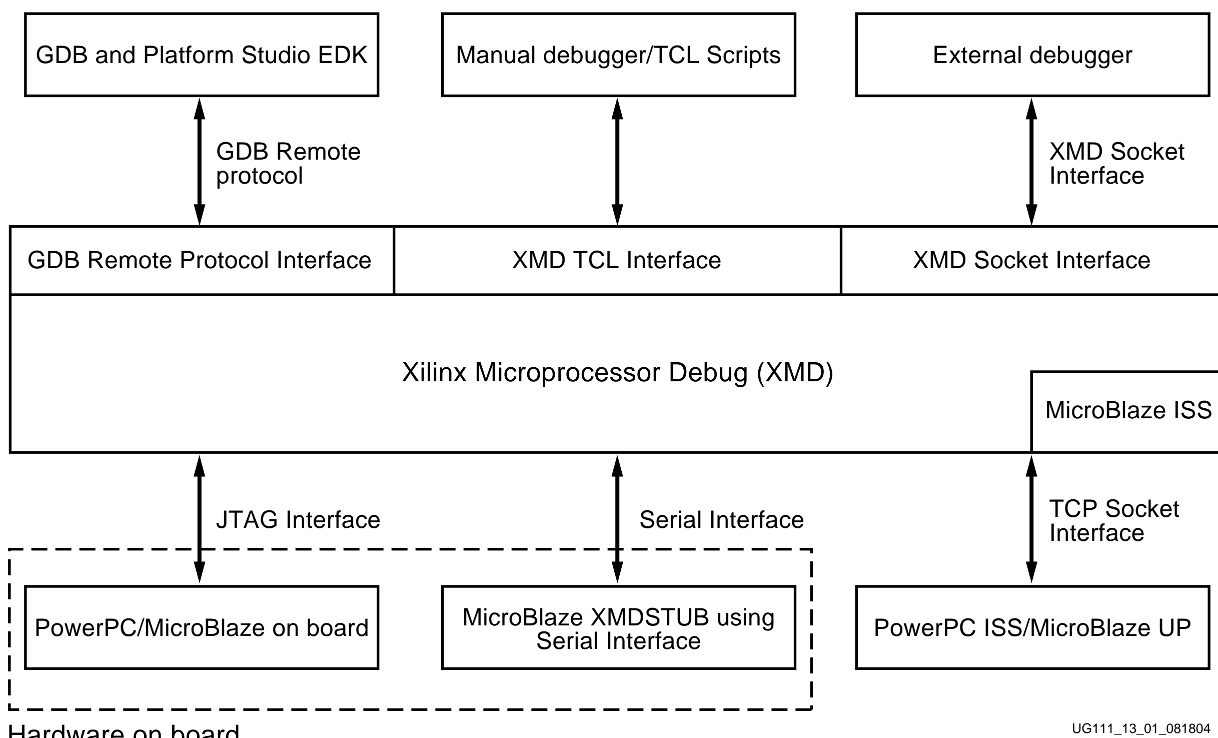


Figure 14-1: **XMD Targets**

XMD Usage

xmd *[-v] [-h] [-help] [-ipcport [<portnum>]] [-xmp <xmpfile>] [-opt <optfile>] [-tcl{<tcl_file_args>}]*

Options:

-h

Help -- display this message and quit.

-help

Help -- display this message and quit.

-v

Display Version and quit.

-ipcport [<portnum>]

Starts XMD server at <portnum>. Internal XMD commands can be issued over this TCP Port. If <portnum> is not specified a default value <2345> is used.

-xmp <xmpfile>

Specify the XMP file to load.

-opt <Connect Option file>

Specify the option file to use to Connect to Target.

-tcl <tclfile> [<tclarg>]

XMD TCL file to execute. <tclargs> are arguments to TCL script. This TCl file is sourced from XMD and quits after execution.

Note: No other option can follow -tcl option

On Startup, XMD does the following:

- If an *xmd TCL script* is specified, xmd will execute the script and quit.
- Otherwise, xmd will be started in an *interactive mode*. In this case, xmd does the following:
  - Source ***${HOME}/.xmdrc*** file. The configuration file can be used to form custom TCL commands using xmd commands.
  - Open XMD Socket server, if -**ipcport** option is given.
  - Load system XMP file, if -**xmp** option is given.
  - Use Connect option file to connect to processor target, if -**opt** option is given.
  - Source ***xmd.ini*** file, if present in the current directory.
  - The XMD% prompt is displayed. From the xmd Tcl prompt, `xmd` commands, described in the following section, can be used for debugging.

XMD Command Reference

Table 14-1: **XMD User Commands**

command [options]	Example Usage	Description
xload *[<xmp> <xmpfile>] [<mhs> <mhsfile>] [<mss> <mssfile>]*	xload xmp system.xmp xload mhs system.mhs xload mss system.mss	Load XMP/MHS/MSS system files. XMD reads MHS and MSS system files for the following reasons: • To infer the connectivity of **FSL** (Fast Synchronous Link) Bus between **opb_mdm** (Microprocessor Debug Module) module and MicroBlaze. This connectivity is used to download program and data at a very fast rate. The section "Fast Download on a MicroBlaze System" in the *Platform Studio User Guide* provides additional information. • To infer Instruction and Data memory address maps of the processor. This information is used to verify program and data downloaded to processor memory.
connect *<Target> <Connect Type> <Options>*	connect mb mdm connect ppc hw	Connect to Target. Valid target types: mb | ppc | mdm. For additional information see the "Connect Command Options" section of this document.
vpconnect *mb*	vpconnect mb	Connect to MicroBlaze Virtual Platform.

command [options]	Example Usage	Description
targets *[<target id>]*	targets targets 0	List information about all current targets or change the current target
disconnect *<target id>*	disconnect 0	Disconnect from the current processor target, close the corresponding GDB server and revert to the previous Processor target if any.
dow *[-data] filename [addr]*	dow executable.elf dow executable.elf 0x400 dow -data system.dat 0x400	Download the given ELF or data file (with -data option) onto the current target's memory. • If no address is provided along with ELF file, the download address is determined from the ELF file by reading its headers. • If an address is provided with the ELF file (only for MicroBlaze targets), it is treated as PIC code (Position Independent Code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics.
run	run	Run program from <Program Start Address>
con *[address]*	con con 0x400	Continue from current PC or "address".
stp *[no. of instructions]*	stp stp 10	Step one or "number" instructions
cstp *[no. of cycles]*	cstp cstp 10	Step one or "number" cycles. For ISS/VP targets.
rst	rst	Reset target.
stop	stop	Stop target.
rrd *[<reg num>]*	rrd rrd r1 (or) rrd R1 rrd 1	Read all registers or Read <reg num> register.
srrd *[<reg name>]*	srrd srrd pc	Read special registers or Read <reg name> register.
rwr *<reg_num | reg name> <word>*	rwr pc 0x400	Register Write.
mrd *<address> [num] [w|h|b]*	mrd 0x400 mrd 0x400 10 mrd 0x400 10 h	Read memory at "address". Default's to a 'w'ord read.

command [options]	Example Usage	Description
mrd_var *<variable> [Elf filename]*	mrd global_var1 executable.elf	Read Memory corresponding to global variable in the ELF file "filename" or in a previously downloaded ELF file
mwr *<address> <values> [<num>] [w|h|b]*	mwr 0x400 0x12345678 mwr 0x400 0x1234 h mwr 0x400 {0x12345678 0x87654321} 2	Write to memory at "address". Default's to a 'w'ord write.
bps *<address | function> [sw|hw]*	bps 0x400 bps main hw	Set software or hardware Breakpoint at "address" or start of "function". The last downloaded ELF file is used for "function" lookup. Default's to software breakpoint.
watch *<r | w> <address> [<data>]*	watch r 0x400 0x1234 watch r 0x40X 0x12X4 watch r 0b01000000XXXX 0b00010010XXXX0100 watch r 0x40X	Set read or write watchpoint at "address". If the value compares to "data" stop the processor. • Address and Data can be specified in hex "0x" format or binary "0b" format. • ***Don't care*** values are specified using 'X'. • Addresses can be only of *contiguous* range. • Default value of data is "0xXXXXXXXX", that is, matches any value. ***Note:*** For PowerPC only absolute values are supported.
bpr *<address | function | bp id | all>*	bpr 0x400 bpr main bpr all	Remove Breakpoint/Watchpoint.
bpl	bpl	List Breakpoints/Watchpoints.
tracestart *[<filename>]*	tracestart trace.txt tracestart	Start collecting trace information to "filename". • Trace collection can be stopped and started at any time of program execution. • "filename" should be specified only on first tracestart. • "filename" defaults to "*isstrace.out*". ***Note:*** Supported only on PowerPC Sim target.
tracestop *[done]*	tracestop tracestop done	Stop collecting trace information. Option "done" signifies end of tracing. ***Note:*** Supported only on PowerPC Sim target.

command [options]	Example Usage	Description
stats *[filename]*	stats trace.txt stats	Display execution statistics for the MicroBlaze or PowerPC simulator target. "filename" is the trace output from trace collection on PowerPC simulator target.
profile *[-o <GMON output file>]*	profile -o gproff.out	Write Profile output file, which can be interpreted by mb-gprof or powerpc-eabi-gprof to generate profiling information. Refer to the "Profiling Embedded Designs" chapter of the *Platform Studio User Guide* for details on Profiling using EDK.
state *[target id]*	state	Displays the current state of all targets or "target id" target.
dis *[address] [num_words]*	dis 0x400 10	Disassemble instruction. ***Note:*** Supported on Microblaze target.
terminal	terminal	JTAG-based hyperterminal to communicate with opb_mdm UART interface.
read_uart *<start | stop> [TCL Channel ID]*	read_uart start read_uart stop read_uart start $channel_id	Read from opb_mdm UART interface. O/P can be redirected to file by specifying a TCL channel ID.
verbose *[level]*	verbose	Toggle ON/OFF verbose mode. In verbose mode XMD prints debug information.
help *[<options>]*	help help init help connect help connect mb	List all commands.

Connect Command Options

XMD can debug programs on different targets (Processor or Virtual Platform or Peripheral). To communicate with a target, XMD should **connect** to the target. An unique *target ID* is assigned to each target after connection. When connecting to Processor or Virtual Platform target, gdbserver is started enabling communication with gdb or Platform Studio SDK.

Usage:

connect *<mb | ppc | mdm> <Connection Type> <Options>*

ppc

Connect to PowerPC Processor.

mb

Connect to MicroBlaze Processor.

mdm

Connect to opb_mdm peripheral.

Connection Type

Connection method, target dependent.

Options

Connection options.

The following sections describe connect options for different targets.

PowerPC Target

Xilinx Virtex-II Pro and Virtex4 devices contain one or two PowerPC405 processor core. `xmd` can connect to these PowerPC targets over a JTAG connection on the board. xmd also communicates over TCP socket interface to IBM PowerPC405 Instruction Set Simulator.

Use the **connect ppc** command to connect to the PowerPC target and start a remote GDB server. Once `xmd` is connected to the PowerPC target, `powerpc-eabi-gdb or Platform Studio SDK` can connect to the processor target through xmd and debugging can proceed.

PowerPC Hardware Connection

When connecting to PowerPC hardware target, `xmd` will detect the JTAG cable, chain and the PowerPC processors automatically and connect to the processor specified. Users can override or provide information using the following options.

Usage:

connect ppc **hw** *[-cable <JTAG Cable options>] {[-configdevice <JTAG chain options>]} [-debugdevice <PowerPC options>]*

JTAG cable options:

type *<cable type>*

Valid cable types are: **xilinx_parallel3, xilinx_parallel4, xilinx_svffile**

In the case of xilinx_svffile, the JTAG commands are written into a file specified by the **fname** option.

port *<parallel port name>*

Valid arguments for port are **lpt1, lpt2.**

fname *<filename>*

Filename for creating the SVF file.

JTAG Chain Options

The following option is needed to specify device information of Non-Xilinx devices in the JTAG chain. Refer to "Example Showing Special JTAG Chain Setup (Non-Xilinx Devices)".

devicenr *<device position>*

Position of the device in the JTAG chain

partname *<devicename>*

Name of the device

irlength *<length of the JTAG Instruction Register>*

Length of the IR register of the device. This information can be found in the device BSDL file.

idcode *<device idcode>*

JTAG Idcode of the device

PPC405 Options:

The following options allow users to specify FPGA device to debug, Processor number in the device, map special PowerPC features like ISOCM, Caches, TLB, DCR registers, etc. to unused memory addresses and then from the debugger access it as memory addresses. This is helpful for reading and writing to these registers/memory from GDB or XMD.

Note: These options **do not** create any real memory mapping in hardware.

devicenr *<PowerPC device position>*

Position of the Virtex-II Pro or Virtex4 device containing the PowerPC, in the JTAG chain

cpunr *<CPU Number>*

ID of the specific PowerPC to be debugged in a Virtex-II Pro or Virtex4 containing multiple PowerPC processors

romemstartadr *<ROM start address>*

Start address of Read-Only Memory. This can be used to specify flash memory range. XMD will set H/W breakpoint instead of software breakpoints.

romemsize *<ROM Size>*

Size of Read-Only Memory.

isocmstartadr *<ISOCM start address>*

Start address for the ISOCM

isocmsize *<ISOCM size>*

Size of the ISBRAM memory connected to the ISOCM interface

isocmdcrstartadr *<ISOCM DCR address>*

DCR address corresponding to the ISOCM interface specified using the C_ISOCM_DCR_BASEADDR parameter on PowerPC

icachestartadr *<I-Cache start address>*

Start address for reading or writing the instruction cache contents

dcachestartadr *<D-Cache start address>*

Start address for reading or writing the data cache contents

itagstartadr *<I-Cache start address>*

Start address for reading or writing the instruction cache tags

dtagstartadr <*D-Cache start address*>

Start address for reading or writing the data cache tags

tlbstartadr <*TLB start address*>

Start address for reading and writing the Translation Look-aside Buffer

dcrstartadr <*DCR start address*>

Start address for reading and writing the Device Control Registers. Using this option, the entire DCR address space (2^{10} addresses) can be mapped to addresses starting from <dcrstartadr> for debugging purposes from XMD and GDB

PowerPC Target Requirements

There are two possible methods for `xmd` to connect to the PowerPC 405 processors over a JTAG connection. The requirements for each of these methods are described below.

1. **Debug connection using the JTAG port of the Virtex-II Pro FPGA**

   If the JTAG ports of the PowerPC processors are connected to the JTAG port of the FPGA internally using the JTAGPPC primitive, then `xmd` can connect to any of the PowerPC processors inside the FPGA, as shown in Figure 14-2. Please refer to the "Virtex-II Pro PPC405 JTAG Debug Port" and "Virtex-4 PPC405 JTAG Debug Port" sections of the *PowerPC 405 Processor Block Reference Guide* for more information.

   Note: There is a core named `jtagppc_cntlr` in EDK that helps in setting up this connection.

2. **Debug connection using user IO pins connected to the JTAG port of the PowerPC**

   If the JTAG ports of the PowerPC processors are brought out of the FPGA using user IO pins, xmd can directly connect to the PowerPC for debugging. Please refer to the "Virtex-II Pro PPC405 JTAG Debug Port" section and "Virtex-4 PPC405 JTAG Debug Port" in the *PowerPC 405 Processor Block Reference Guide* for more information about this debug setup.

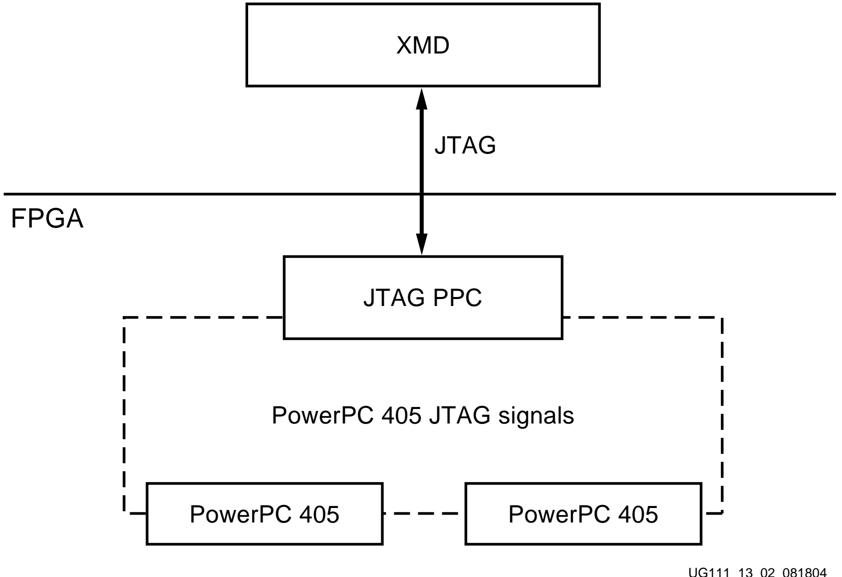


Figure 14-2: **PowerPC Target**

Example Debug Sessions

Example Using a PowerPC Target

This example demonstrates a simple debug session with a PowerPC target. Basic **xmd**-based commands are used after connecting to the PowerPC target using the “connect ppc hw” command. At the end of the session, GDB (powerpc-eabi-gdb) is connected to **xmd** using the GDB remote target. Refer to the GDB section of the est_guide for more information about connecting GDB to **xmd**.

```
XMD% connect ppc hw

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093          8         XC18V04
 2       0123e093         10         XC2VP4
assumption: selected device 2 for debugging.

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
PC: 0xffffef20
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
    ISOCM           :  Disabled
    TLB             :  Disabled
    DCR             :  Disabled
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD% rrd
    r0: ef0009f8      r8: 51c6832a     r16: 00000804     r24: 32a08800
    r1: 00000003      r9: a2c94315     r17: 00000408     r25: 31504400
    r2: fe008380     r10: 00000003     r18: f7c7dfcd     r26: 82020922
    r3: fd004340     r11: 00000003     r19: fbcbefce     r27: 41010611
    r4: 0007a120     r12: 51c6832a     r20: 0040080d     r28: fe0006f0
    r5: 000b5210     r13: a2c94315     r21: 0080040e     r29: fd0009f0
    r6: 51c6832a     r14: 45401007     r22: c1200004     r30: 00000003
    r7: a2c94315     r15: 8a80200b     r23: c2100008     r31: 00000003
    pc: ffff0700     msr: 00000000
XMD% srrd
    pc: ffff0700     msr: 00000000      cr: 00000000      lr: ef0009f8
   ctr: ffffffff     xer: c000007f     pvr: 20010820   sprg0: ffffe204
 sprg1: ffffe204   sprg2: ffffe204   sprg3: ffffe204    srr0: ffff0700
  srr1: 00000000     tbl: a06ea671     tbu: 00000010  icdbdr: 55000000
   esr: 88000000    dear: 00000000    evpr: ffff0000     tsr: fc000000
   tcr: 00000000     pit: 00000000    srr2: 00000000    srr3: 00000000
  dbsr: 00000300   dbcr0: 81000000    iac1: ffffe204    iac2: ffffe204
  dac1: ffffe204    dac2: ffffe204    dccr: 00000000    iccr: 00000000
   zpr: 00000000     pid: 00000000     sgr: ffffffff    dcwr: 00000000
  ccr0: 00700000   dbcr1: 00000000    dvc1: ffffe204    dvc2: ffffe204
  iac3: ffffe204    iac4: ffffe204    sler: 00000000   sprg4: ffffe204
 sprg5: ffffe204   sprg6: ffffe204   sprg7: ffffe204    su0r: 00000000
usprg0: ffffe204
XMD% rst
Sending System Reset
Target reset successfully
XMD% rwr 0 0xAAAAAAAA
```

```
XMD% rwr 1 0x0
XMD% rwr 2 0x0
XMD% rrd
    r0: aaaaaaaa      r8: 51c6832a     r16: 00000804     r24: 32a08800
    r1: 00000000      r9: a2c94315     r17: 00000408     r25: 31504400
    r2: 00000000     r10: 00000003     r18: f7c7dfcd     r26: 82020922
    r3: fd004340     r11: 00000003     r19: fbcbefce     r27: 41010611
    r4: 0007a120     r12: 51c6832a     r20: 0040080d     r28: fe0006f0
    r5: 000b5210     r13: a2c94315     r21: 0080040e     r29: fd0009f0
    r6: 51c6832a     r14: 45401007     r22: c1200004     r30: 00000003
    r7: a2c94315     r15: 8a80200b     r23: c2100008     r31: 00000003
    pc: fffffffc     msr: 00000000
XMD% mrd 0xFFFFFFFC
FFFFFFFC:   4BFFFC74
XMD% stp
fffffc70:
XMD% stp
fffffc74:
XMD% mrd 0xFFFFC000 5
FFFFC000:   00000000
FFFFC004:   00000000
FFFFC008:   00000000
FFFFC00C:   00000000
FFFFC010:   00000000
XMD% mwr 0xFFFFC004  0xabcd1234 2
XMD% mwr 0xFFFFC010  0xa5a50000
XMD% mrd 0xFFFFC000 5
FFFFC000:   00000000
FFFFC004:   ABCD1234
FFFFC008:   ABCD1234
FFFFC00C:   00000000
FFFFC010:   A5A50000
XMD%
XMD: Accepted a new GDB connection from nnn.nnn.n.nn on port nnnn
XMD%
XMD: Closed connection
XMD%
```

Example with a Program Running in ISOCM Memory and Accessing DCR Registers

```
XMD% connect ppc hw -debugdevice \
isocmstartadr 0xFFFFE000 isocmsize 8192 isocmdcrstartadr 0x15 \
dcrstartadr 0xab000000

JTAG chain configuration
--------------------------------------------------
Device    ID Code          IR Length     Part Name
 1        05026093            8          XC18V04
 2        0123e093           10          XC2VP4
assumption: selected device 2 for debugging.

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
PC: 0xffffe218
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
```

```
    ISOCM             :  Start Address - 0xffffe000
     TLB               :  Disabled
    DCR              :  Start Address - 0xab000000

Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD% stp
ffffe21c:
XMD% stp
ffffe220:
XMD% bps 0xFFFFE218
Setting breakpoint at 0xffffe218
XMD% con
Processor started. Type "stop" to stop processor
RUNNING>
8
Processor stopped at PC: 0xffffe218
XMD% bpl
HW BP: BP_ID 0 : addr = 0xffffe218 <--- Automatic Hardware Breakpoint
                                      for ISOCM
XMD% mrd 0xFFFFE218
Warning: Attempted to read location: 0xffffe218. Reading ISOCM memory
not supported in V2Pro
Cannot read from target
XMD%
XMD% mrd 0xab000060 8
AB000060:   00000000
AB000064:   00000000
AB000068:   FF000000 <--- DCR register : ISARC
AB00006c:   81000000 <--- DCR register : ISCNTL
AB000070:   00000000
AB000074:   00000000
AB000078:   FE000000 <--- DCR register : DSARC
AB00007c:   81000000 <--- DCR register : DSCNTL
XMD%
```

Note: In Virtex4 device ISOCM memory is readable. This enables better debugging of program running from ISOCM memory.

Example Showing Special JTAG Chain Setup (Non-Xilinx Devices)

This example demonstrates the use of -configdevice option to specify the JTAG chain on the board, in case **xmd** is unable to auto detect the JTAG chain. The auto detect in **xmd** might fail for non-xilinx devices on the board for which the JTAG IRLengths are not known. The JTAG (Boundary Scan) IRLength information is usually available in BSDL files provided by device vendors. For these "Unknown" devices, IRLength is the only critical information needed and the other fields like partname and idcode are optional.

Following is a description of the options use in the example below,

- Xilinx Parallel cable (III or IV) connection is done over the LPT1 parallel port.
- The two devices in the JTAG chain are explicitly specified
  - the IRLength, partname and idcode of the PROM are specified
  - only the IRLength of the 2nd device is specified. Partname is inferred from the idcode since **xmd** knows about the XC2VP4 device
- The debugdevice option explicitly specifies to **xmd** that the FPGA device of interest is the 2nd device in the JTAG chain. In the Virtex-II Pro, it is also explicitly

specified that the connection is for the 1st PowerPC processor (if there are more than one)

```
XMD% connect ppc hw -cable type xilinx_parallel4 port LPT1 \
> -configdevice devicenr 1 partname PROM irlength 8 idcode 0x05026093 \
> -configdevice devicenr 2 irlength 10 \
> -debugdevice devicenr 2 cpunr 1

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093          8         PROM_XC18V04
 2       0123e093         10         XC2VP4

XMD: Connected to PowerPC target. Processor Version No : 0x20010820
PC: 0xffffee18
Address mapping for accessing special PowerPC features from XMD/GDB:
    I-Cache (Data)  :  Disabled
    I-Cache (Tag)   :  Disabled
    D-Cache (Data)  :  Disabled
    D-Cache (Tag)   :  Disabled
    ISOCM           :  Disabled
    TLB             :  Disabled
    DCR             :  Disabled
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
XMD%
```

Adding Non-Xilinx Devices

XMD reads Device Information from ***${XILINX_EDK}/data/xmd/devicetable.lst*** file. To add support for a device in XMD, do the following:

a. Edit the devicetable.lst file. Add Device ID Code, Instruction Register Length and Name information.

b. If XMD is open, close XMD and restart. XMD will Auto-Discover the Device in JTAG chain.

PowerPC Simulator Target

xmd can connect to one or more PowerPC Instruction Set Simulator (ISS) targets through socket connection. Use the **connect ppc sim** command to start the PowerPC ISS on localhost , connect to it and start a remote GDB server. **cconnect ppc sim** can also connect to PowerPC ISS running on localhost or other machine. Once **`xmd`** is connected to the PowerPC target, **`powerpc-eabi-gdb`** can connect to the target through xmd and debugging can proceed.

Running PowerPC ISS

XMD starts the ISS with default configuration. The ISS executable can be found in *${XILINX_EDK}/third_party/bin/<platform>/* directory. The configuration file used is *${XILINX_EDK}/third_party/data/iss405.icf*. User can run ISS with different configuration option and xmd can connect to the ISS target. Refer to the *ISS User Guide* for more details. The following are the default configuration for ISS.

- Two local memory banks: Mem0 start address = 0x0, length = 0x80000 and speed = 0. Mem1 start address = 0xfff80000, length = 0x80000 and speed = 0.
- Connect to Debugger (xmd)
- Debugger Port at 6470..6490
- Data Cache size of 8k
- Instruction Cache size of 16k
- Non-Deterministic Multiply cycles
- Processor Clock Period and Timer Clock Period of 5ns (200 Mhz)

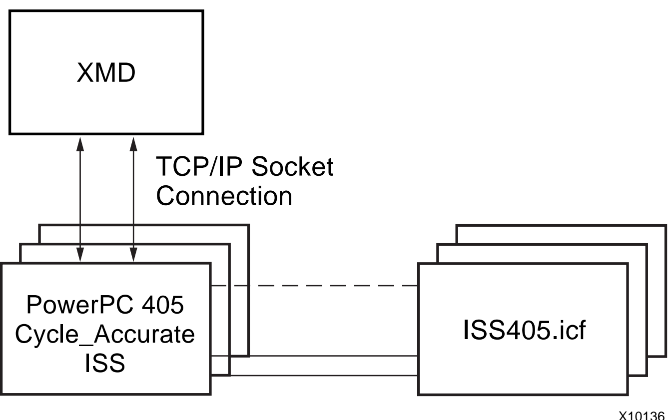


Figure 14-3: **PowerPC ISS Target**

Usage:

connect ppc **sim** *[-icf <Configuration File>] [-ipcport IP:port]*

-icf

The given ISS Configuration file is used instead of default configuration file. User's can customize the PowerPC ISS features like cache size, memory address map, memory latency, etc.

-ipcport

Specify the IP address and debug port of PowerPC ISS started by the user. XMD does not spawn a ISS, but connects to the user defined ISS.

Example Debug Session for PowerPC ISS Target

```
XMD% connect ppc sim
Instruction Set Simulator (ISS)
PPC405, PPC440
Version 1.5 (1.69)
(c) 1998, 2002 IBM Corporation
Waiting to connect to controlling interface (port=6470,
protocol=tcp)....
[XMD] Connected to PowerPC Sim
Controling interface connected....
Connected to PowerPC target. id = 0
Starting GDB server for target (id = 0) at TCP port no 1234
```

```
XMD% dow dhry2.elf
XMD% bps 0xffff09d0
XMD% tracestart trace.out
XMD% con
Processor started. Type "stop" to stop processor

RUNNING>
XMD% tracestop
XMD% tracestart
XMD% con
Processor started. Type "stop" to stop processor

RUNNING>
XMD% tracestop done
XMD% stats trace.out
Program Stats ::

       Instructions :  197491
              Loads :   20296
             Stores :   19273
    Multiplications :    3124
           Branches :   27262
     Branches taken :   20985
            Returns :  2070
```

MicroBlaze Processor Target

xmd can connect through JTAG to one or more MicroBlaze processors using the **opb_mdm** `(Microprocessor Debug Module)` peripheral. xmd can communicate with ROM monitor like xmdstub through JTAG or Serial interface. Users can also debug programs using built-in Cycle-accurate Microblaze ISS. The following sections describe various options for these targets.

Microblaze MDM Hardware Target

Use the command "`connect mb mdm`" in order to connect to the `mdm` target and start the remote GDB server. The MDM target supports non-intrusive debugging using hardware breakpoints and hardware single-step, without the need for a ROM monitor.

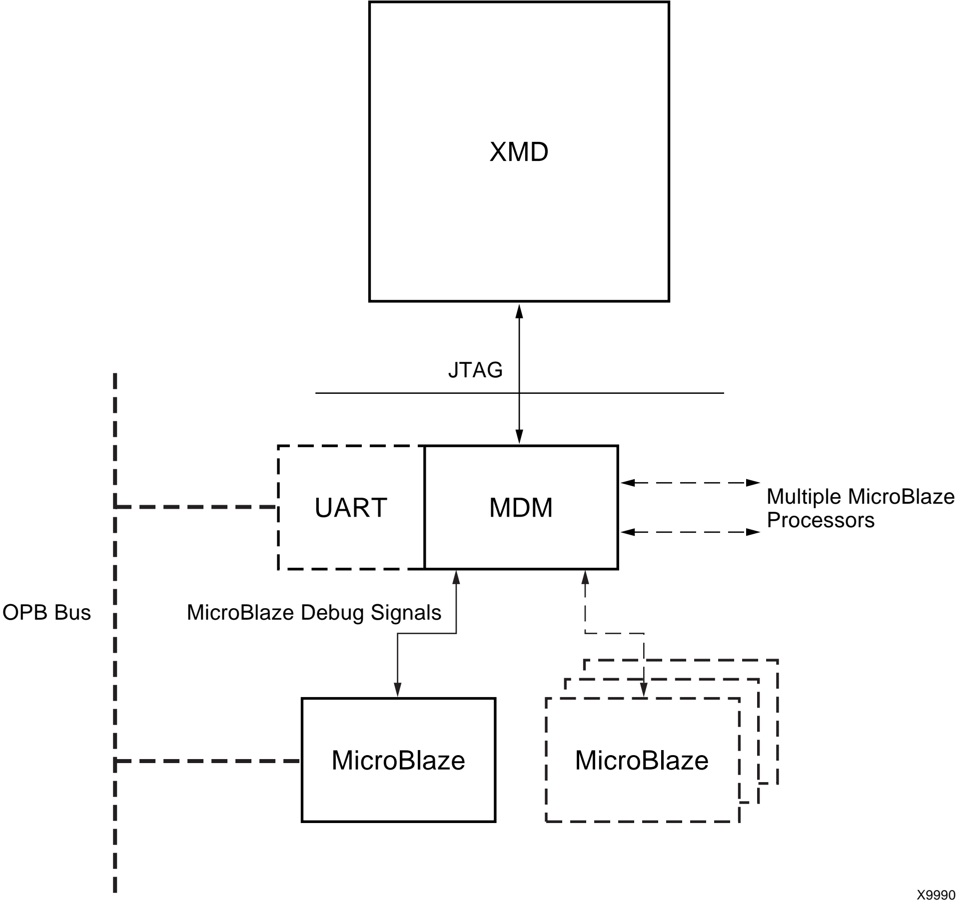


Figure 14-4: **MicroBlaze MDM Target**

When no option is specified to the `connect mb mdm`, xmd will automatically detect the JTAG cable, chain and the FPGA device containing the MicroBlaze-MDM system. If xmd is unable to detect the JTAG chain or the FPGA device automatically, users can explicitly specify them, using the following options.

Usage:

connect mb **hw** *[-cable <JTAG Cable options>] {[-configdevice <JTAG chain options>]} [-debugdevice <MicroBlaze options>] [-pfsl <MicroBlaze FSL options>]*

JTAG Cable Options and JTAG Chain Options

Refer PowerPC Hardware Target options.

MicroBlaze Options

devicenr *<PowerPC device position>*

Position of the FPGA device containing the MicroBlaze, in the JTAG chain

cpunr *<CPU Number>*

ID of the specific MicroBlaze to be debugged in a FPGA containing multiple MicroBlaze processors connected to opb_mdm.

romemstartadr *<ROM start address>*

Start address of Read-Only Memory. This can be used to specify flash memory range. XMD will set H/W breakpoint instead of s/w breakpoints.

romemsize *<ROM Size>*

Size of Read-Only Memory.

MicroBlaze FSL Options

These options specify the MicroBlaze FSL port to use for fast downloading.

port *<FSL port number>*

FSL port on MicroBlaze.

MicroBlaze MDM Target Requirements

1. To use the hardware debug features on MicroBlaze, such as hardware breakpoints, hardware debug control functions like stopping, stepping, etc, MicroBlaze's hardware debug port must be connected to the Microprocessor Debug Module, the **opb_mdm** core. The following MHS snippet demonstrates the debug port connection needed between the MDM and MicroBlaze.

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 3.00.a
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 8
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 PORT CLK = sys_clk_s
 PORT DBG_CAPTURE = DBG_CAPTURE_s
 PORT DBG_CLK = DBG_CLK_s
 PORT DBG_REG_EN = DBG_REG_EN_s
 PORT DBG_TDI = DBG_TDI_s
 PORT DBG_TDO = DBG_TDO_s
 PORT DBG_UPDATE = DBG_UPDATE_s
END

BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_MB_DBG_PORTS = 1
 PARAMETER C_USE_UART = 1
```

```
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0x0000c000
 PARAMETER C_HIGHADDR = 0x0000c0ff
 BUS_INTERFACE SOPB = mb_opb
 PORT OPB_Clk = sys_clk_s
 PORT DBG_CAPTURE_0 = DBG_CAPTURE_s
 PORT DBG_CLK_0 = DBG_CLK_s
 PORT DBG_REG_EN_0 = DBG_REG_EN_s
 PORT DBG_TDI_0 = DBG_TDI_s
 PORT DBG_TDO_0 = DBG_TDO_s
 PORT DBG_UPDATE_0 = DBG_UPDATE_s
END
```

2. To use the UART functionality in the MDM target, users have to set the C_USE_UART parameter while instantiating the **opb_mdm** in a system. To print program STDOUT onto the XMD console, C_UART_WIDTH should be set as 8. UART input can also be provided from the host to the program running on MicroBlaze by using the **"xuart w** <**byte**>" command.
3. In order to perform fast download on MicroBlaze- target, the **opb_mdm** Master FSL Bus Interface (MSFL0) should be connected to MicroBlaze Slave FSL Bus Interface (SFSL0). The following MHS snippet demonstrates the debug port connection needed between the MDM and MicroBlaze.

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_i
 PARAMETER HW_VER = 3.00.a
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_DIV = 1
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 4
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 1
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 1
 PARAMETER C_FSL_LINKS = 1
 BUS_INTERFACE SFSL0 = download_link
 BUS_INTERFACE DLMB = d_lmb_v10
 BUS_INTERFACE ILMB = i_lmb_v10
 BUS_INTERFACE DOPB = d_opb_v20
 BUS_INTERFACE IOPB = d_opb_v20
 PORT CLK = sys_clk
 PORT INTERRUPT = interrupt
END

BEGIN opb_mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_MB_DBG_PORTS = 1
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER C_BASEADDR = 0xFFFFC000
 PARAMETER C_HIGHADDR = 0xFFFFC0FF
 PARAMETER C_WRITE_FSL_PORTS = 1
 BUS_INTERFACE MFSL0 = download_link
 BUS_INTERFACE SOPB = d_opb_v20
 PORT OPB_Clk = sys_clk
END


BEGIN fsl_v20
```

```
 PARAMETER INSTANCE = download_link
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT SYS_Rst = sys_rst
 PORT FSL_Clk = sys_clk
END
```

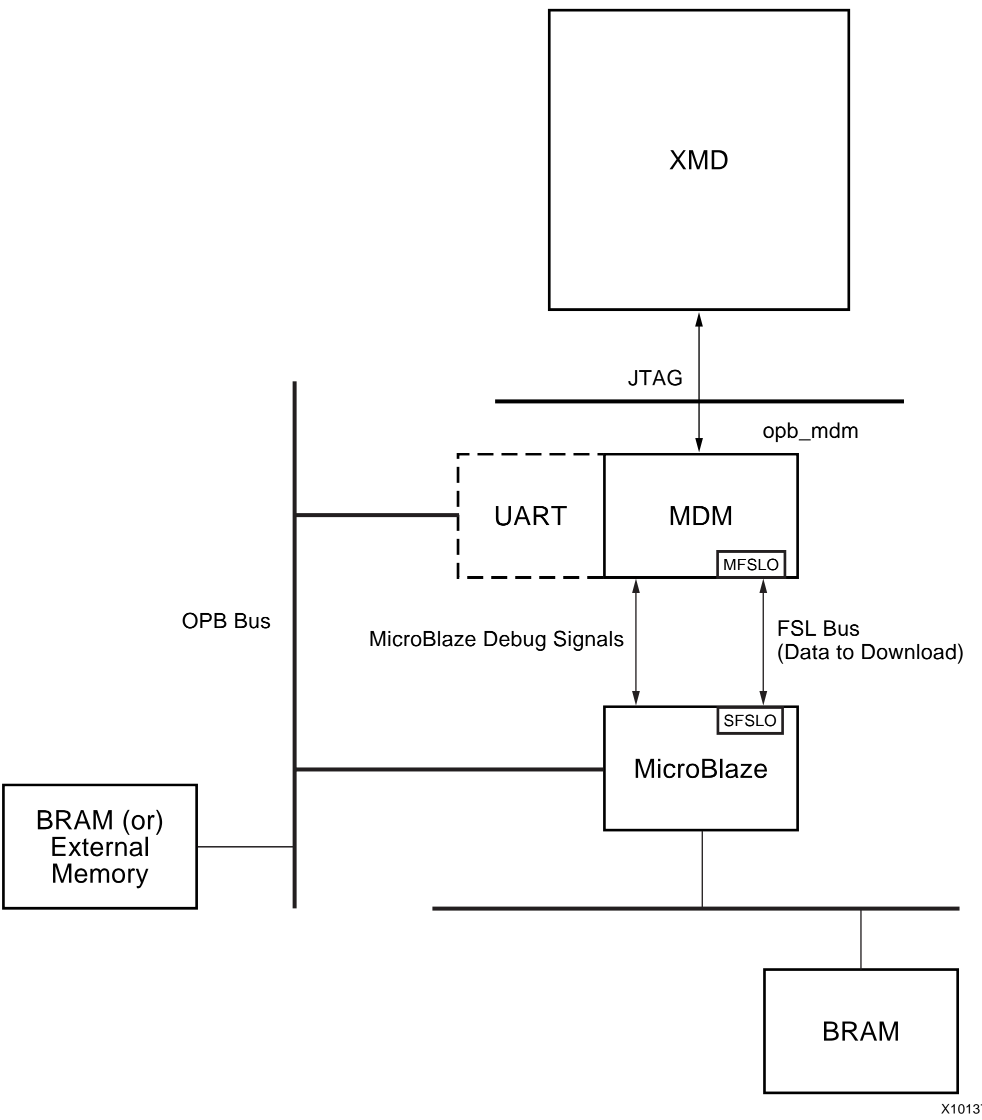


Figure 14-5: **MicroBlaze-MDM connection for Fast Download**

When the MHS file is loaded, XMD infers this connectivity automatically. When the size of program or data is greater than 256 bytes, fast download is used automatically. The section "Fast Download on a MicroBlaze System" in the *Platform Studio User Guide* describes fast download on MicroBlaze.

Note: Unlike the MicroBlaze stub target, programs should be compiled in executable mode and NOT in xmdstub mode while using the MDM target. Consequently, users need NOT specify a XMDSTUB_PERIPHERAL for compiling the xmdstub

Example Debug Sessions

Example Using a MicroBlaze MDM Target

This example demonstrates a simple debug session with a MicroBlaze MDM target. Basic **xmd**-based commands are used after connecting to the MDM target using the "connect mb mdm" command. At the end of the session, GDB (mb-gdb) is connected to **xmd** using the GDB remote target. Refer to the GDB section of the est_guide for more information about connecting GDB to **xmd**.

```
XMD% connect mb mdm

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
-------------------------------------
Version............................3.00.a
No of PC Breakpoints...............4
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support..........off
Data Cache Support.................off

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000510      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 00000140     r10: 00000000     r18: 00000000     r26: 00000000
    r3: a5a5a5a5     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 00000000     r13: 00000140     r21: 00000000     r29: 00000000
    r6: 00000000     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000064     r23: 00000000     r31: 00000000
    pc: 00000070     msr: 00000004
<--- Launching GDB from XMD% console --->
XMD% start mb-gdb microblaze_0/code/executable.elf
XMD%
<--- From GDB, a connection is made to XMD and debugging is done from
the GDB GUI --->
XMD: Accepted a new GDB connection from 127.0.0.1 on port 3791
XMD%
XMD: GDB Closed connection
XMD% stp
BREAKPOINT at
     114:   F1440003  sbi     r10, r4, 3
XMD% dis 0x114 10
     114:   F1440003  sbi     r10, r4, 3
     118:   E0E30004  lbui    r7, r3, 4
     11C:   E1030005  lbui    r8, r3, 5
     120:   F0E40004  sbi     r7, r4, 4
     124:   F1040005  sbi     r8, r4, 5
```

```
        128:    B800FFCC  bri     -52
        12C:    B6110000  rtsd    r17, 0
        130:    80000000  Or      r0, r0, r0
        134:    B62E0000  rtid    r14, 0
        138:    80000000  Or      r0, r0, r0
XMD% dow microblaze_0/code/executable.elf
XMD% con
Processor started. Type "stop" to stop processor
RUNNING> stop <--- From this "RUNNING>" prompt, the debugging commands
"stop", "xuart", "xrreg 0 32" and some other basic Tcl commands can be
executed.
XMD%
Processor stopped at PC: 0x0000010c
XXMD% con
Processor started. Type "stop" to stop processor
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x000000f4 <--- With the MDM, the current PC of MicroBlaze can be
                     read while the program is running
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x00000110 <--- Note: the PC is constantly changing, as the
                     program is running
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x00000118 <--- Note: "format" is a basic Tcl command like printf
RUNNING> format "PC = 0x%08x" [xrreg 0 32]
PC = 0x00000118
XMD% rrd
    r0: 00000000      r8: 00000065     r16: 00000000     r24: 00000000
    r1: 00000548      r9: 0000006c     r17: 00000000     r25: 00000000
    r2: 00000190     r10: 0000006c     r18: 00000000     r26: 00000000
    r3: 0000014c     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000500     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 24242424     r13: 00000190     r21: 00000000     r29: 00000000
    r6: 0000c204     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000068     r15: 0000005c     r23: 00000000     r31: 00000000
    pc: 0000010c     msr: 00000000
XMD% bps 0x100
Setting breakpoint at 0x00000100
XMD% bps 0x11c hw
Setting breakpoint at 0x0000011c
XMD% bpl
SW BP: addr = 0x00000100, instr = 0xe1230002 <-- Software Breakpoint
HW BP: BP_ID   0 : addr = 0x0000011c      <--- Hardware Breakpoint
XMD% con
Processor started. Type "stop" to stop processor
RUNNING>
Processor stopped at PC: 0x00000100
XMD% con
Processor started. Type "stop" to stop processor
RUNNING>
Processor stopped at PC: 0x0000011c
```

Example Using Two MicroBlaze Processors and a JTAG-based UART in MDM

```
XMD% connect mb mdm -debugdevice cpunr 1

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
```

```
 1       05026093           8        XC18V04
 2       0123e093          10        XC2VP4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 2

MicroBlaze Processor 1 Configuration :
-------------------------------------
Version...........................3.00.a
No of PC Breakpoints...............4
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support.........off
Data Cache Support................off
JTAG MDM Connected to Mircoblaze 1

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234

XMD% connect mb mdm -debugdevice cpunr 2

MicroBlaze Processor 2 Configuration :
-------------------------------------
Version...........................3.00.a
No of PC Breakpoints...............4
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support.........off
Data Cache Support................off
JTAG MDM Connected to Mircoblaze 2

Connected to MicroBlaze "mdm" target. id = 1
Starting GDB server for "mdm" target (id = 0) at TCP port no 1235
<--- Note: Two GDB servers are started at different TCP ports for
parallel debugging from GDB -->
XMD% targets
List of connected targets

Target ID          Target Type
------------------------------
0                  MicroBlaze MDM-based (hw) Target
1                 MicroBlaze MDM-based (hw) Target *
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000540      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 000001e8     r10: 00000000     r18: 00000000     r26: 00000000
    r3: 00000000     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 0000c000     r13: 000001e8     r21: 00000000     r29: 00000000
    r6: 00000000     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000130     r23: 00000000     r31: 00000000
   pc: 00000188    msr: 00000000
XMD% targets 0
Setting current target to target id 0
List of connected targets

Target ID          Target Type
------------------------------
0                  MicroBlaze MDM-based (hw) Target *
```

```
1                   MicroBlaze MDM-based (hw) Target
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000548      r9: 0000006c     r17: 00000000     r25: 00000000
    r2: 00000190     r10: 0000006c     r18: 00000000     r26: 00000000
    r3: 0000014c     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000500     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 02020202     r13: 00000190     r21: 00000000     r29: 00000000
    r6: 0000c200     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 0000006f     r15: 0000005c     r23: 00000000     r31: 00000000
   pc: 000000f8    msr: 00000000
XMD% mrd 0xC000 4     <--- Reading the MDM UART's registers from
                          MicroBlaze's point of view
    C000:    00000000
    C004:    00000000
    C008:    00000004 <--- Note: Status reg is 4, i.e UART is empty
    C00C:    00000000
XMD% xuart w 0x42 <--- Write a character onto the MDM UART from the host
XMD% mrd 0xC008   <--- Read the MDM UART status reg using MicroBlaze
    C008:    00000005 <--- Status is "valid data present"
XMD% mrd 0xC000 <--- Read the UART data i.e consume the char
    C000:    00000042
XMD% mrd 0xC008
    C008:    00000004 <--- Status is again "empty"
XMD% scan "Hello" "%c%c%c%c%c" ch1 ch2 ch3 ch4 ch5
5
XMD% xuart w $ch1
XMD% xuart w $ch2
XMD% xuart w $ch3
XMD% xuart w $ch4
XMD% xuart w $ch5
XMD% dow uart_test.elf
XMD% con
Processor started. Type "stop" to stop processor
RUNNING> Hello
```

Example Debug Session with Read Watchpoints

In this debug session, there is a program running on the board that is polling and waiting on MDM UART input; UART is at Baseaddress 0xC000. The program loops around waiting for the data valid bit to be set in the status register 0xC008. Using a read watchpoint, MicroBlaze is stopped as soon as there is load from address 0xC000. In the MicroBlaze configuration below, there are for PC hardware breakpoints, one Read Addr/Data watchpoint and one Write Addr/Data watchpoint.

```
XMD% connect mb mdm

JTAG chain configuration
--------------------------------------------------
Device    ID Code          IR Length     Part Name
 1        05026093           8           XC18V04
 2        0123e093          10           XC2VP4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
-------------------------------------
```

```
Version...........................3.00.a
No of PC Breakpoints..............4
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support.........off
Data Cache Support................off
JTAG MDM Connected to Mircoblaze 1

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD% mrd 0xC000 4
    C000:   00000000
    C004:   00000000
    C008:   00000004
    C00C:   00000000
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000540      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 000001e8     r10: 00000000     r18: 00000000     r26: 00000000
    r3: 00000000     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 0000c000     r13: 000001e8     r21: 00000000     r29: 00000000
    r6: 00000042     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000130     r23: 00000000     r31: 00000000
    pc: 00000190     msr: 00000000
XMD% dis 0x188 5
     188:   E8650008  lwi     r3, r5, 8
     18C:   A4630001  andi    r3, r3, 1
     190:   BC03FFF8  beqi    r3, -8
     194:   C8602800  lw      r3, r0, r5
     198:   B60F0008  rtsd    r15, 8
XMD% watch r xC000
Setting watchpoint at 0x0000c000
XMD% con
Processor started. Type "stop" to stop processor
RUNNING> xuart w 0x42

RUNNING>
Processor stopped at PC: 0x00000198
XMD% dis 0x194
     194:   C8602800  lw      r3, r0, r5
XMD% rrd
    r0: 00000000      r8: 00000000     r16: 00000000     r24: 00000000
    r1: 00000540      r9: 00000000     r17: 00000000     r25: 00000000
    r2: 000001e8     r10: 00000000     r18: 00000000     r26: 00000000
    r3: 00000042     r11: 00000000     r19: 00000000     r27: 00000000
    r4: 00000000     r12: 00000000     r20: 00000000     r28: 00000000
    r5: 0000c000     r13: 000001e8     r21: 00000000     r29: 00000000
    r6: 00000000     r14: 00000000     r22: 00000000     r30: 00000000
    r7: 00000000     r15: 00000130     r23: 00000000     r31: 00000000
    pc: 00000198     msr: 00000000
XMD%
```

Example with Special JTAG Chain Setup (Non-Xilinx Devices)

This example demonstrates the use of -configdevice option to specify the JTAG chain on the board, in case **xmd** is unable to autodetect the JTAG chain. The autodetect in **xmd** might fail for non-xilinx devices on the board for which the JTAG IRLengths are not known. The JTAG (Boundary Scan) IRLength information is usually available in BSDL files

provided by device vendors. For these "Unknown" devices, IRLength is the only critical information needed and the other fields like partname and idcode are optional.

Following is a description of the options use in the example below,

- Xilinx Parallel cable (III or IV) connection is done over the LPT1 parallel port.
- The two devices in the JTAG chain are explicitly specified
  - only the IRLength of the PROM is specified. Partname is inferred from the idcode since **xmd** knows about the XC18V04 PROM device
  - the IRLength, partname and idcode of the 2nd device is specified.
- The debugdevice option explicitly specifies to **xmd** that the FPGA device of interest is the 2nd device in the JTAG chain.

```
XMD% connect mb mdm \
> -configdevice devicenr 1 irlength 8 \
> -configdevice devicenr 2 irlength 10 idcode 0x0123e093 partname V2P4 \
> -debugdevice devicenr 2

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       05026093           8        XC18V04
 2       0123e093          10        V2P4
Assuming, Device No: 2 contains the MicroBlaze system
Connected to the JTAG MicroBlaze Debug Module (MDM)
No of processors = 1
MicroBlaze Processor 1 Configuration :
-------------------------------------
Version............................3.00.a
No of PC Breakpoints...............4
No of Read Addr/Data Watchpoints...1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support.........off
Data Cache Support................off
JTAG MDM Connected to Mircoblaze 1

Connected to MicroBlaze "mdm" target. id = 0
Starting GDB server for "mdm" target (id = 0) at TCP port no 1234
XMD%
```

MicroBlaze Stub Hardware Target

Connect to a MicroBlaze target using the xmdstub (a ROM monitor running on the target) as well as start a GDB server for the target. xmd connects to xmdstub through JTAG or Serial interface. The default option connects using JTAG interface.

MicroBlaze Stub-JTAG Target Options

Usage

connect mb **stub -comm jtag** *[-cable <JTAG Cable options>] {[-configdevice <JTAG chain options>]} [-debugdevice <MicroBlaze options>]*

JTAG Cable Options and JTAG Chain Options

Refer to PowerPC Hardware Target options.

MicroBlaze Options

devicenr *<PowerPC device position>*

Position of the FPGA device containing the MicroBlaze, in the JTAG chain

MicroBlaze Stub-Serial Target Options

Usage

connect mb **stub** -**comm serial** *<Serial Communication options>*

Serial Communication Options

-port *<serial port>*

Specify the serial port to which the remote hardware is connected, when xmd communication is over the serial cable. The default serial port is */dev/ttya* on Solaris, */dev/ttyS0* on Linux and *Com1* on Windows

-baud *<serial port baud rate>*

Specify the serial port baud rate in bps. The default value is *19200* bps.

-timeout *<timeout in secs>*

Timeout period while waiting for reply from xmdstub for xmd commands.

Note: User Program outputs. If the program has any I/O functions like print() or putnum(), that write output onto the UART or JTAG Uart, it will be printed on the console/terminal where the **xmd** was started. (Refer to the MicroBlaze Libraries chapter for libraries and I/O functions information).

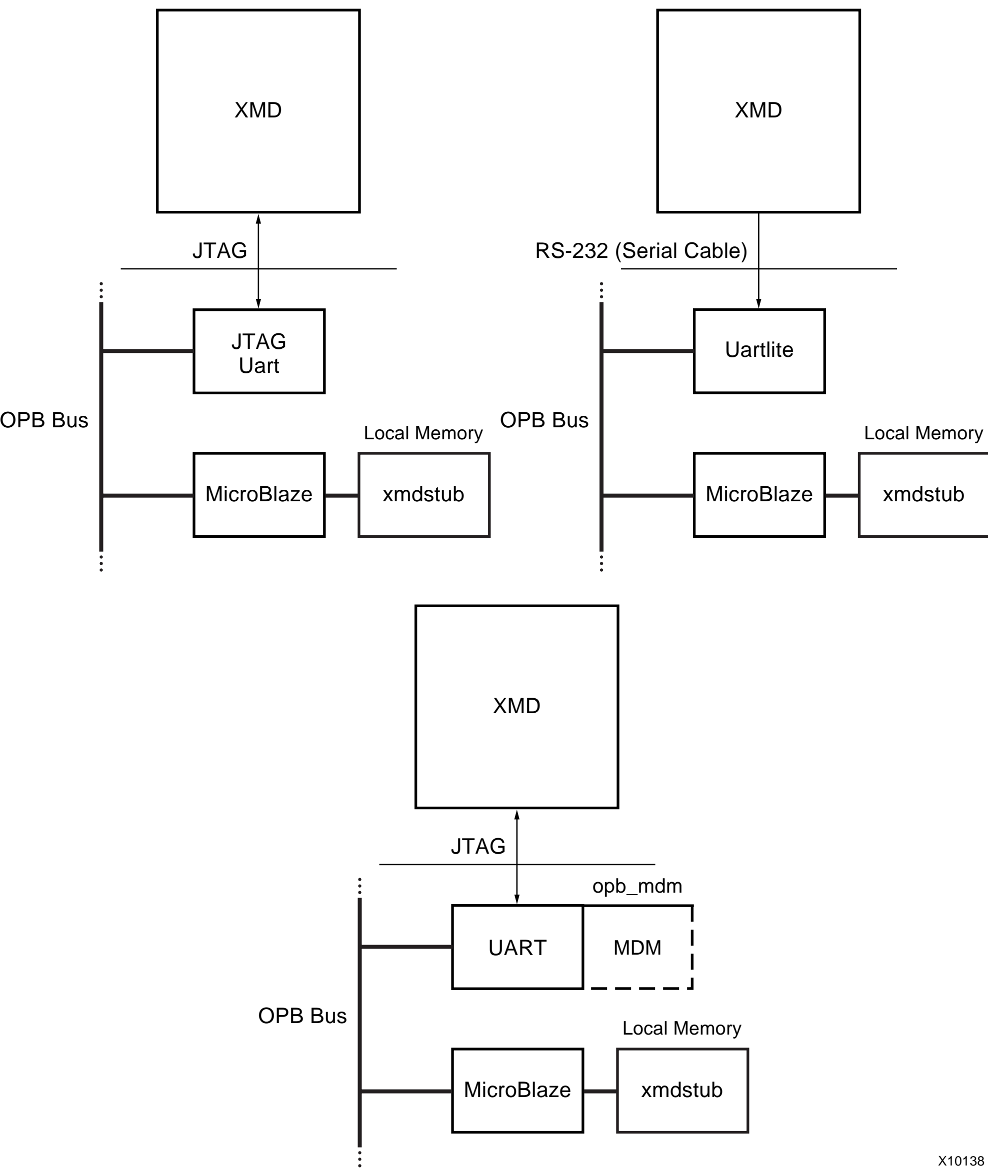


Figure 14-6: **MicroBlaze stub Target with JTAG UART and Uartlite**

Stub Target Requirements

To debug programs on the hardware board using XMD, the following requirements have to be met.

- **xmd** uses a JTAG or serial connection to communicate with **xmdstub** on the board. Hence a opb_mdm or a Uart designated as XMDSTUB_PERIPHERAL in the mss file is needed on the target MicroBlaze system.

  Platform Generator can create a system that includes a opb_mdm or a UART, if specified in the system's MHS file. For more information on creating a system with a UART or opb_mdm, refer to the "MicroBlaze Hardware Specification Format"

chapter of the *Platform Specification Format Reference Manual*. The cables supported with the xmdstub mode are: Xilinx Parallel Cable III and Parallel Cable IV.

- **`xmdstub`** on the board uses the opb_mdm or Uart to communicate with the host computer. Hence, it must be configured to use the opb_mdm or Uart in the MicroBlaze system.

  Library Generator can configure the **`xmdstub`** to use the XMDSTUB_PERIPHERAL in the system. **`libgen`** will generate a **`xmdstub`** configured for the XMDSTUB_PERIPHERAL and put it in **`code/xmdstub.elf`** as specified by the XMDSTUB attribute in the mss file. For more information, refer to the Library Generator chapter.

- **`xmdstub`** executable must be included in the MicroBlaze local memory at system startup.

  Data2MEM can populate the MicroBlaze memory with **`xmdstub`**. libgen generates a Data2MEM script file that can be used to populate the BRAM contents of a bitstream containing a MicroBlaze system. It uses the executable specified in the DEFAULT_INIT.

- Any user program that has to be downloaded on the board for debugging should have a program start address higher than 0x400 and the program should be linked with the startup code in crt1.o

  `mb-gcc` can compile programs satisfying the above two conditions when it is run with the option **`-xl-mode-xmdstub`**.

Note: For source level debugging, programs should also be compiled with the -g option. While initially verifying the functional correctness of a C program, it is advisable to not use any mb-gcc optimization option like -O2 or -O3 as mb-gcc does aggressive code motion optimizations which may make debugging difficult to follow.

MicroBlaze Simulator Target

Users can use **`mb-gdb`** and **`xmd`** to debug programs on the cycle-accurate simulator built in XMD.

Usage

connect mb **sim** *[-memsize <size>]*

memsize *<size>*

Size of the memory address bus allocated in the simulator. Programs can access the memory range from `0` to 2^{size-1} .Default memory size of 64Kbytes.

Simulator Target Requirements

To debug programs on the Cycle-Accurate Instruction Set Simulator using XMD, the following requirements have to be met.

- Programs should be compiled for debugging and should be linked with the startup code in crt0.o

  `mb-gcc` can compile programs with debugging information when it is run with the option **`-g`** and by default, mb-gcc links crt0.o with all programs. (Explicit option: **`-xl-mode-executable`**)

- Programs have a default memory size of 64Kbytes.

- Currently, XMD with simulator target does not support the simulation of OPB peripherals.

MDM Peripheral Target

User can connect to opb_mdm peripheral and use the UART interface for debugging and collecting information from the system.

Usage

connect **mdm** <-uart>

MDM Target Requirements

In order to use the UART functionality in the MDM target, users have to set the C_USE_UART parameter while instantiating the `opb_mdm` in a system. In order to print program STDOUT onto the xmd console, C_UART_WIDTH should be set as 8.

UART input can also be provided from the host to the program running on MicroBlaze by using the "**xuart** *w <byte>*" command. User can use "**terminal**" command to spawn a hyperteminal-like interface to read/write from UART interface. "**read_uart**" command provides interface to write to STDIO or to file.

Virtual Platform Microblaze Target

User can connect to MicroBlaze Virtual Platform target for debugging. VP is a Cycle-Accurate model of Microblaze system used for debugging, profiling and tracing programs accurately. XMD spawns VP if executable is present in ***<system>/virtualplatform/*** directory and communicates over TCP socket interface.

Usage

vpconnect mb

XMD Internal Tcl Commands

In the Tcl interface mode, xmd starts a Tcl shell augmented with xmd commands. All xmd Tcl commands start with 'x' and can be listed from xmd by typing "x?". It is recommended to use the Tcl wrappers for these internal commands as described in Figure 14-1. The Tcl wrappers would pretty print the output of most of these commands and also provide more options. While the Tcl wrappers will be backwards compatible, these x<name> commands may be deprecated in a future EDK release.

Program Initialization

xload_sysfile *<xmp|mhs|mss> <XMP|MHS|MSS filename>*

Load XMP/MHS/MSS file.

xrut *[Session ID]*

Authenticate....

xconnect *<target> <connect type> [options]*

Connect to Processor or Peripheral target. Valid Target types are : mb | ppc | mdm. Refer Connect command Section for information on options.

xvpconnect *mb*

Connect to MicroBlaze Virtual Platform Target.

xdisconnect <*target id*>

Disconnect from using the target.

xtargets [<*target id*>]

Print the target ID and target type of all current targets or a specific target.

Register/Memory

xrmem <*target id*> *addr* [*num*]

Read num bytes or 1 byte from memory address <addr>

xwmem <*target id*> *addr value*

Write a 8-bit byte *value* at the specified memory *addr*.

xrreg <*target id*> [*reg*]

Read all registers or only register number **`reg.`**

xwreg <*target id*> *reg value*

Write a 32-bit *value* into register number *reg*

xdownload <*target id*> [-data] *filename* [*addr*]

Download the given ELF or data file (with -data option) onto the current target's memory. If no address is provided along with ELF file, the download address is determined from the ELF file by reading its headers. If an address is provided with the ELF file, it is treated as PIC code (Position Independent Code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics. Note that NO Bounds checking is done by xmd, except preventing writes into xmdstub area (address 0x0 to 0x800).

xdisassemble *inst*

Disassemble and display one 32-bit instruction.

Program Control

xcontinue <*target id*> [*addr*] *[-quit]*

Continue execution from the current PC or from the optional address argument.

xstop <*target id*>

Stop the Program execution.

xcycle_step <*target id*> *[cycles]*

Cycle step through one clock cycle of PowerPC ISS. If *cycles* is specified, then step "*cycles*" number of clock cycles. **Note:** This command is only for Simulator targets.

xstep <*target id*>

Single step one MicroBlaze instruction. If the PC is at an IMM instruction the next instruction is executed as well. During a single step, interrupts are disabled by keeping the BIP flag set. Use xcontinue with breakpoints to enable interrupts while debugging.

xreset <target id> [reset type]

Reset target. Optionally provide target specific reset types like signals mentioned in Table 14-2.

xbreakpoint *<target id>* <addr | function name><sw|hw>

Set a breakpoint at the given address or start of function. **Note:** Breakpoints on instructions immediately following an `imm` instruction can lead to undefined results for an xmdstub target.

xwatch *<target id>* <r!w><address> [value]

Set read/write watchpoints at a given <address> and check for <value>. If <value> is not specified, watchpoints match any value. Address and value can be specified in hex or binary format.

xremove <target id> <addr | function name | bp id | all>

Remove breakpoint/watchpoint.

xlist *<target id>*

List all the breakpoint addresses.

Table 14-2: **XMD MicroBlaze Hardware Target Signals**

Signal Name (Value)	Description
Processor Break (0x20)	Raises the Brk signal on MicroBlaze using the JTAG UART Ext_Brk signal. It sets the Break-in-Progress (BIP) flag on MicroBlaze and jumps to addr 0x18
Non-maskable Break (0x10)	Similar to the Break signal but works even while the BIP flag is already set. Refer the MicroBlaze ISA documentation for more information about the BIP flag.
System Reset (0x40)	Resets the entire system by sending an OPB Rst using the JTAG UART Debug_SYS_Rst signal.
Processor Reset (0x80)	Resets MicroBlaze using the JTAG UART Debug_Rst signal.

Program Trace/Profile

xstats *<target id>* [*options*]

Display the simulation statistics for the current session.'reset' option can be provided to reset the simulation statistics.

xtraceopen *<target id> [filename]*

Open a trace file to collect trace information. If *filename* is not specified, *isstrace.out* is used as the default filename. **Note:** This command is only for PowerPC ISS target.

xtracestart *<target id>*

Start collecting trace information. Trace file should be opened before trace start. **Note:** This command is only for PowerPC ISS target.

xtracestop *<target id>*

Stop collecting trace information.

Note: This command is only for PowerPC ISS target.

xtraceclose *<target id>*

Close the trace file.

Note: This command is only for PowerPC ISS target.

xprofile *<target id>* [-o <GMON Output File>]

Generate Profile output that can be read by mb-gprof or powerpc-eabi-gprof.

Miscellaneous Commands

xuart *<r|w|s> [<data>]*

Perform one of 3 UART operations on the MDM's UART if it is enabled. This command is valid only for the MDM target.

xuart <r> - Read byte from the MDM UART

xuart <w> <data> - Write byte onto the MDM UART

xuart <s> - Read the status of MDM UART

xforce_use_fsl_dow *<target id>*

Force XMD to use FSL based fast download. This command should be used when the cable type is xilinx_svffile, when reading from target is not possible. Especially when SystemACE file is generated.

xverbose

Toggle ON/OFF verbose mode. Dumps debugging information from XMD.

xhelp

Lists the XMD commands.

XMD TCP Socket Interface

External tools can communicate with XMD through TCP socket interface. An XMD TCP server is started when XMD is invoked with -ipcport option. XMD internal commands can be executed through this interface. The Socket interface follows the following protocol:

Sending Commands to XMD

```
<xmd command> [command options] #
```

is the terminating string. The <xmd command> is evaluated.

Return Types

XMD returns Command Execution Status (Success, Error, Info, Warn, etc.) and related output strings in the following format.

1. **Success:** `X [Return Values] # [String to Print on stdo]`
2. **Info:** `I <Information printed on stdo>`
3. **Warn:** `W <Warning printed on stderr>`

4. **Error:**

   Usage: *E01 <Usage> [String to Print on stderr] # [String to Print on stdo]*

   Error: *E02 <Msg to stderr> [String to Print on stderr] # [String to Print on stdo]*