Embedded System Tools Reference Manual

Embedded Development Kit, EDK 8.2i

UG111 (v6.0) June 23, 2006

XILINX®

.

Embedded System Tools Reference Manual
UG111 (v6.0) June 23, 2006

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.
03/19/04	2.0	Updated for service pack release.
08/20/04	3.0	EDK 6.3i release.
02/15/05	4.0	EDK 7.1i release.
04/28/05	4.1	Updated for service pack release.
07/05/05	4.2	Updated for service pack release.
10/24/05	5.0	EDK 8.1i release.
06/23/06	6.0	EDK 8.2i release.

About This Guide

Welcome to the Embedded Development Kit (EDK). This product provides you with a full set of design tools and a wide selection of standard peripherals required to build embedded processor systems based on the MicroBlaze™ soft processor and PowerPC™ hard processor.

This guide contains information about the embedded system tools included in EDK. These tools, consisting of processor platform tailoring utilities, software application development tools, a full featured debug tool chain, and device drivers and libraries, allow you to fully exploit the power of MicroBlaze and PowerPC processors along with their corresponding peripherals.

Guide Contents

This guide contains the following chapters:

- Chapter 1, “Embedded System and Tools Architecture Overview”
- Chapter 2, “Platform Generator (Platgen)”
- Chapter 3, “Simulation Model Generator (Simgen)”
- Chapter 4, “Library Generator (Libgen)”
- Chapter 5, “Virtual Platform Generator (VPgen)”
- Chapter 6, “Platform Specification Utility (PsfUtility)”
- Chapter 7, “Version Management Tools”
- Chapter 8, “Bitstream Initializer (BitInit)”
- Chapter 9, “Flash Memory Programming”
- Chapter 10, “GNU Compiler Tools”
- Chapter 11, “GNU Debugger (GDB)”
- Chapter 12, “Xilinx Microprocessor Debugger (XMD)”
- Chapter 13, “System ACE File Generator (GenACE)”
- Chapter 14, “EDK Shell”
- Chapter 15, “Command Line (no window) Mode”
- Appendix A, “GNU Utilities”
- Appendix B, “Interrupt Management”
- Appendix C, “Glossary”

Additional Resources

Resource	Description/URL
EDK Home	Embedded Development Kit home page, FAQ and tips. http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm
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://www.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at: http://www.xilinx.com/support/mysupport.htm
Application Notes	Descriptions of device-specific design techniques and approaches http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp?iLanguageID=1&category=-1209827&sGlobalNavPick=&sSecondaryNavPick=
Data Sheets	Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://www.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment: http://www.xilinx.com/xlnx/xil_tt_home.jsp
GNU Manuals	The entire set of GNU manuals may be found at: http://www.gnu.org/manual
ISE Manuals	ISE software manuals are available at: http://www.xilinx.com/support/software_manuals.htm
Other Documentation	Additional documentation on the Xilinx website: http://www.xilinx.com/support/library.htm

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	Refer To 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	Refer to 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	Refer to 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 7
Typographical 7
Online Document 8

Chapter 1: Embedded System and Tools Architecture Overview

About EDK 19
Design Process Overview 20
An Introduction to EDK Tools and Utilities 21
Xilinx Platform Studio (XPS) 23
Xilinx Software Development Kit (SDK) 24
EDK Command Line or "no window" Mode 24
The Base System Builder (BSB) Wizard 24
The Create and Import IP Wizard 25
Configure Coprocessor Wizard 25
Platform Generator (Platgen) 25
Library Generator (Libgen) 27
GNU Compiler Tools (GCC) 28
Debug Configuration Wizard 28
Xilinx Microprocessor Debugger (XMD) 28
GNU Debugger (GDB) 29
Simulation Model Generator (Simgen) 29
Simulation Library Compiler (CompEDKLib) 29
Virtual Platform Generator (VPgen) 29
Bus Functional Model Compiler (BFM) 29
Bitstream Initializer (Bitinit) 29
System ACE File Generator (GenACE) 30
Flash Memory Programmer 30
Format Revision (revup) Tool and Version Management Wizard 30
LibXil Memory File System Generator (LibXil MFS) 30
Platform Specification Utility 31
Project Creation and Management 31
Controlling the EDK Flow (Advanced) 31

Chapter 2: Platform Generator (Platgen)

Tool Requirements 33
Tool Usage 33
Tool Options 34
Load Path 35

Output Files . . . 35
HDL Directory . . . 35
Implementation Directory . . . 36
Synthesis Directory . . . 36
About Memory Generation . . . 36
BMM Policy . . . 37
BMM Flow . . . 37
Reserved MHS Parameters . . . 38
Synthesis Netlist Cache . . . 39
Restrictions . . . 39

Chapter 3: Simulation Model Generator (Simgen)

Simgen Overview . . . 41
Simulation Libraries . . . 41
Xilinx Libraries . . . 42
UNISIM Library . . . 42
SIMPRIM Library . . . 42
XilinxCoreLib Library . . . 42
EDK Library . . . 42
CompXLib Utility . . . 43
CompEDKLib Utility . . . 43
Usage . . . 43
CompEDKLib Command Line Examples . . . 44
Use Case I: Launching the GUI to Compile the Xilinx and EDK Simulation Libraries . . . 44
Use Case II: Compiling HDL Sources in the Built-In Repositories in the EDK . . . 44
Use Case III: Compiling HDL Sources in Your Own Repository . . . 44
Other Details . . . 45
Simulation Models . . . 45
Behavioral Models . . . 45
Structural Models . . . 46
Timing Models . . . 46
Single and Mixed Language Models . . . 47
Creating Simulation Models Using XPS Batch . . . 47
Simgen Syntax . . . 48
Requirements . . . 48
Options . . . 48
Output Files . . . 50
Memory Initialization . . . 50
VHDL . . . 50
Verilog . . . 51
Simulating Your Design . . . 51
Restrictions . . . 51

Chapter 4: Library Generator (Libgen)

Overview . . . 53
Tool Usage . . . 53
Tool Options . . . 54

Load Paths . . . 55
Unix System Load Paths . . . 55
PC System Load Paths . . . 56
Additional Directories . . . 56
Search Priority Mechanism . . . 56
Output Files . . . 57
`include` Directory . . . 58
`lib` Directory . . . 58
`libsrc` Directory . . . 58
`code` Directory . . . 58
Libraries and Drivers Generation . . . 58
Basic Philosophy . . . 58
MDD, MLD, and Tcl . . . 59
MSS Parameters . . . 59
Drivers . . . 60
Libraries . . . 60
OS Block . . . 61
Interrupts and Interrupt Controllers . . . 61
Importance of Instantiation . . . 61
Interrupt Controller Driver Customization . . . 61
MicroBlaze . . . 61
PowerPC . . . 62
XMDStub Peripherals (MicroBlaze Specific) . . . 62
STDIN and STDOUT Peripherals . . . 62

Chapter 5: Virtual Platform Generator (VPgen)

Overview . . . 63
Tool Usage and Options . . . 63
Output Files . . . 64
Available Models . . . 64
Current Restrictions . . . 66

Chapter 6: Platform Specification Utility (PsfUtility)

Tool Options . . . 67
Overview of the MPD Creation Process . . . 68
Detailed Use Models for Automatic MPD Creation . . . 69
Peripherals with a Single Bus Interface . . . 69
Signal Naming Conventions . . . 69
Invoking the PsfUtility . . . 69
Peripherals with Multiple Bus Interfaces . . . 69
Non-Exclusive Bus Interfaces . . . 70
Exclusive Bus Interfaces . . . 70
Peripherals with TRANSPARENT Bus Interfaces . . . 70
BRAM PORTS . . . 71
Peripherals with Point-to-Point Connections . . . 71
DRC Checks in PsfUtility . . . 71
HDL Source Errors . . . 71
Bus Interface Checks . . . 71

Conventions for Defining HDL Peripherals 72
Naming Conventions for Bus Interfaces 72
Naming Conventions for VHDL Generics 73
Reserved Parameters 74
Naming Conventions for Bus Interface Signals 75
Global Ports 76
Slave DCR Ports 76
Slave FSL Ports 77
Master FSL Ports 78
Slave LMB Ports 79
Master OPB Ports 80
Slave OPB Ports 81
Master/Slave OPB Ports 82
Master PLB Ports 84
Slave PLB Ports 85

Chapter 7: Version Management Tools

Overview 87
Format Revision Tool Backup and Update Processes 87
Changes in 8.2i 87
Changes in 8.1i 88
Changes in 7.1i 88
Changes in 6.3i 88
Changes in 6.2i 88
Command Line Option for the Format Revision Tool 89
The Version Management Wizard 89

Chapter 8: Bitstream Initializer (BitInit)

Overview 91
Tool Usage 91
Tool Options 91

Chapter 9: Flash Memory Programming

Overview 93
Flash Programming from XPS 93
Supported Flash Hardware 94
Customizing Flash Programming 95
Manual Conversion of ELF Files to SREC for Bootloader Applications 97
Operational Characteristics and Workarounds 97
Handling Flash Devices with Conflicting Sector Layouts 97
Data Polling Algorithm for AMD/Fujitsu Command Set 97

Chapter 10: GNU Compiler Tools

Compiler Framework 100
Common Compiler Usage and Options 101
Usage 101
Input Files 101
Output Files 101

File Types and Extensions . . . 102
Libraries . . . 102
Language Dialect . . . 103
Quick Reference . . . 104
Compiler Options . . . 104
-g . . . 105
-gstabs . . . 105
-On . . . 105
-v . . . 105
-save-temps . . . 105
-o filename . . . 106
-Wp,*option* . . . 106
-Wa,*option* . . . 106
-Wl,*option* . . . 106
--help . . . 106
Library Search Options . . . 106
Header File Search Option . . . 107
Default Search Paths . . . 107
Linker Options . . . 108
-defsym _STACK_SIZE=value . . . 108
-defsym _HEAP_SIZE=value . . . 109
Memory Layout . . . 109
Reserved Memory . . . 109
I/O Memory . . . 110
User and Program Memory . . . 110
Object-File Sections . . . 111
.text . . . 111
.rodata . . . 111
.sdata2 . . . 111
.data . . . 112
.sdata . . . 112
.sbss . . . 112
.bss . . . 112
.heap . . . 112
.stack . . . 112
.init . . . 112
.fini . . . 112
.ctors . . . 112
.dtors . . . 113
.got2/.got . . . 113
.eh_frame . . . 113
Linker Scripts . . . 113
MicroBlaze Compiler Usage and Options . . . 114
MicroBlaze Compiler . . . 114
Compiler Options Quick Reference . . . 114
-mcpu=vX.YY.Z . . . 115
-mxl-soft-mul . . . 116
-mno-xl-soft-mul . . . 116
-mxl-soft-div . . . 116
-mno-xl-soft-div . . . 116
-mxl-stack-check . . . 116
-mxl-barrel-shift . . . 116
-mxl-gp-opt . . . 117

-mxl-pattern-compare . . . 117
-mhard-float . . . 117
-mno-clearbss . . . 117
Application Execution Modes . . . 118
-xl-mode-executable . . . 118
-xl-mode-xmdstub . . . 118
-xl-mode-bootstrap . . . 118
-xl-mode-novectors . . . 118
MicroBlaze Assembler . . . 119
MicroBlaze Linker . . . 120
-defsym _TEXT_START_ADDR=value . . . 120
-relax . . . 120
-N . . . 120
Startup Files . . . 120
First Stage Initialization Files . . . 121
Second Stage Initialization Files . . . 122
Other files . . . 123
Modifying Startup Files . . . 123
Reducing the Startup Code Size for C Programs . . . 123
Compiler Libraries . . . 124
Command Line Arguments . . . 124
Interrupt Handlers . . . 125
_interrupt_handler attribute . . . 125
_save_volatiles attribute . . . 125
PowerPC Compiler Usage and Options . . . 126
Compiler Options . . . 126
-mfpu={sp_lite, sp_full} . . . 126
-msoft-float . . . 126
-mppcperflib . . . 126
-mno-clearbss . . . 127
Linker Options . . . 127
-defsym _START_ADDR=value . . . 127
Startup Files . . . 127
xil-crt0.o . . . 128
xil-pgcrt0.o . . . 129
xil-sim-crt0.o . . . 129
xil-sim-pgcrt0.o . . . 129
Other files . . . 129
Modifying Startup Files . . . 129
Reducing the Startup Code Size for C Programs . . . 129
Modifying Startup Files for Bootstrapping an Application . . . 130
Compiler Libraries . . . 130
Other Notes . . . 131
C++ Code Size . . . 131
C++ Standard Library . . . 131
Position Independent Code (Relocatable Code) . . . 131
Other Switches and Features . . . 131

Chapter 11: GNU Debugger (GDB)

Overview . . . 133
- Tool Usage . . . 133
- Tool Options . . . 133
- Debug Flow using GDB . . . 134

MicroBlaze GDB Targets . . . 134
- Simulator Target . . . 134
- Hardware Target . . . 134
- Virtual Platform Target . . . 134
- Compiling for Debugging on MicroBlaze Targets . . . 134

PowerPC Targets . . . 135

Console Mode . . . 135

GDB Command Reference . . . 136

Chapter 12: Xilinx Microprocessor Debugger (XMD)

XMD Usage . . . 138

XMD Command Reference . . . 140
- XMD User Commands . . . 140
- Special Purpose Register Names . . . 144
  - MicroBlaze Special Purpose Register Names . . . 144
  - PowerPC Special Purpose Register Names . . . 144

Connect Command Options . . . 145
  - Usage . . . 145
- PowerPC Target . . . 145
  - PowerPC Hardware Connection . . . 146
  - PowerPC Target Requirements . . . 148
  - Example Debug Sessions . . . 149
  - PowerPC Simulator Target . . . 153
  - Running PowerPC ISS . . . 153
  - Example Debug Session for PowerPC ISS Target . . . 154
  - Advanced PowerPC Debugging Tips . . . 155
- MicroBlaze Processor Target . . . 155
  - MicroBlaze MDM Hardware Target . . . 156
  - MicroBlaze MDM Target Requirements . . . 158
  - Example Debug Sessions . . . 160
  - MicroBlaze Stub Hardware Target . . . 167
  - MicroBlaze Stub-JTAG Target Options . . . 167
  - MicroBlaze Stub-Serial Target Options . . . 167
  - Stub Target Requirements . . . 169
  - MicroBlaze Simulator Target . . . 169
  - Simulator Target Requirements . . . 170
  - MDM Peripheral Target . . . 170
- Virtual Platform MicroBlaze Target . . . 170
  - Configure Debug Session . . . 171

XMD Internal Tcl Commands . . . 173
- Program Initialization Options . . . 173
- Register/Memory Options . . . 174
- Program Control Options . . . 175
- Program Trace/Profile Options . . . 176
- Miscellaneous Commands . . . 176

Chapter 13: System ACE File Generator (GenACE)

Assumptions . . . 179
Tool Requirements . . . 179
GenACE Features . . . 179
GenACE Model . . . 180
The Genace.tcl Script . . . 181
Syntax . . . 181
Usage . . . 183
Supported Target Boards . . . 183
Generating ACE Files . . . 184
Single FPGA Device . . . 184
Hardware and Software Configuration . . . 184
Hardware and Software Partial Reconfiguration . . . 184
Hardware Only Configuration . . . 184
Hardware Only Partial Reconfiguration . . . 184
Software Only Configuration . . . 184
Software Only Configuration on MicroBlaze; Downloading Using Fast Download . . . 185
ACE Generation for a Single Processor in Multi-Processor System: . . . 185
Multi-Processor System Configuration: . . . 185
Multiple FPGA Devices . . . 186
Related Information . . . 186
Adding a New Device to the JTAG Chain . . . 186
CF Device Format . . . 187

Chapter 14: EDK Shell

Summary . . . 189
The EDK-Installed Cygwin Environment . . . 189
Requirements for Using an Existing Cygwin Environment . . . 189
EDK Shell . . . 189
Using xbash . . . 190
The -override and -undo Options . . . 190
Cygwin Commands . . . 190

Chapter 15: Command Line (no window) Mode

Creating a New Empty Project . . . 194
Creating a New Project With an Existing MHS . . . 194
Opening an Existing Project . . . 194
Reading an MSS File . . . 194
Saving Files and Your Project . . . 194
Setting Project Options . . . 195
Executing Flow Commands . . . 196
Reloading an MHS File . . . 197
Adding a Software Application . . . 197
Deleting a Software Application . . . 197
Adding a Program File to a Software Application . . . 197
Deleting a Program File from a Software Application . . . 197

Setting Options on a Software Application 198
Settings on Special Software Applications 199
Restrictions 199
MSS Changes 199
XMP Changes 199

Appendix A: GNU Utilities

General Purpose Utility for MicroBlaze and PowerPC 201
cpp 201
gcov 201
Utilities Specific to MicroBlaze and PowerPC 201
mb-addr2line 201
mb-ar 201
mb-as 201
mb-c++ 202
mb-c++filt 202
mb-g++ 202
mb-gasp 202
mb-gcc 202
mb-gdb 202
mb-gprof 202
mb-ld 202
mb-nm 202
mb-objcopy 202
mb-objdump 202
mb-ranlib 203
mb-readelf 203
mb-size 203
mb-strings 203
mb-strip 203
Other Programs and Files 203

Appendix B: Interrupt Management

Overview of Interrupt Management in EDK 205
Steps Involved in Interrupt Management 205
Interrupt Handling in MicroBlaze and PowerPC 205
Interrupt Ports 205
Enabling Interrupts 205
For Additional Information 206
Connecting Interrupts 206
Interrupt Service Routines (ISRs) 208
For Additional Information 208
Interrupt Vector Tables 208
MicroBlaze 208
PowerPC 209
Libgen Customization 209
Purpose of the Libgen Tool 209
Introducing xparameters.h 209

Example Systems for MicroBlaze 210
MicroBlaze System Without an Interrupt Controller (Single Interrupt Signal) 210
Procedure 210
Example MHS File Snippet (for an Internal Interrupt Signal). 210
Example MSS File Snippet 211
Example C Program 211
Example MHS File Snippet for an External Interrupt Signal 212
Example MSS File Snippet 212
Example C Program 212
MicroBlaze System With an Interrupt Controller (One or More Interrupt Signals) 213
Procedure 213
MHS File Snippet Showing an INTC for a Timer and UART 214
Example MSS File Snippet 214
Example C Program 215
Example Systems for PowerPC 217
PowerPC System Without Interrupt Controller (Single Interrupt Signal) 217
Procedure 217
Example MHS File Snippet (for an Internal Interrupt Signal). 217
Example MSS File Snippet 218
Example C Program 218
Example MHS File Snippet (For External Interrupt Signal) 220
Example MSS File Snippet 220
Example C Program 220
PowerPC System With an Interrupt Controller (One or More Interrupt Signals) . . 221
Procedure 221
Example MHS File Snippet 222
Example MSS File Snippet 223
Example C Program 223

Appendix C: Glossary

Chapter 1

Embedded System and Tools Architecture Overview

This chapter describes the architecture of the embedded system tools and flows provided in the Xilinx® Embedded Development Kit (EDK) for developing systems based on the Xilinx® PowerPC™ and MicroBlaze™ embedded processors. The chapter contains the following sections:

- About EDK
- Design Process Overview
- An Introduction to EDK Tools and Utilities
- Project Creation and Management

About EDK

Providing a suite of design tools based on a common framework, the Xilinx Embedded Development Kit (EDK) enables you to design a complete embedded processor system for implementation in a Xilinx FPGA device.

Note: Though the Xilinx Integrated Software Environment (ISE™) must be installed along with EDK, it is possible to create your entire design from start to finish in the EDK environment.

EDK includes:

- The Xilinx Platform Studio (XPS) GUI.
- The embedded system tool suite.
- Embedded processing IP cores, such as processors and peripherals.
- The Platform Studio SDK (Software Development Kit), based on the Eclipse open-source framework, which you can use (optionally) to develop your embedded software application.

When you install EDK, you must also install the Integrated Software Environment (ISE), a Xilinx development system product required to implement designs into Xilinx programmable logic devices. EDK depends on ISE components to synthesize the microprocessor hardware design, to map it to an FPGA target, and to generate and download the bitstream.

For information about ISE, refer to the ISE software documentation. For links to ISE documentation and other useful information see "Additional Resources" in the "Preface" section of this book.

Design Process Overview

The tools provided with EDK are designed to assist in all phases of the embedded design process, as illustrated in Figure 1-1.

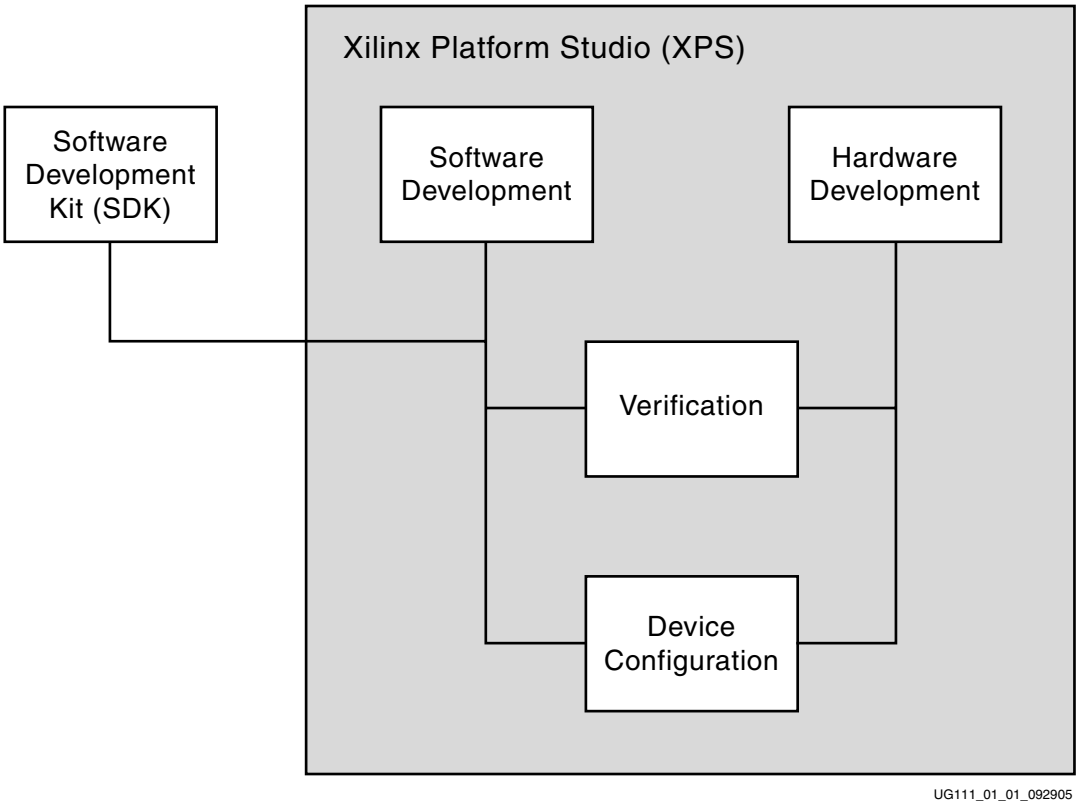

Figure 1-1: **Basic Embedded Design Process Flow**

Hardware Platform Creation

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

The hardware platform consists of one or more processors and peripherals connected to the processor buses. EDK captures the hardware platform in the MHS file (Microprocessor Hardware Specification).

Software Platform Creation

A software platform is a collection of software drivers and, optionally, the operating system on which to build your application. The software image created consists only of the portions of the Xilinx library you actually use in your embedded design. EDK captures the software platform in the MSS file (Microprocessor Software Specification). You can create multiple applications to run on the software platform.

Hardware Platform Verification Using Simulation

To verify the correct functionality of your hardware platform, you can create a simulation model and run it on an HDL simulator. When simulating your system, the processor(s) run your actual software executables. You can choose to create a behavioral, structural, or timing-accurate simulation model.

Software Verification Using Debugging

The primary technique for debugging embedded software is to load your design on a supported development board and use a debugging tool to control the target processor. Alternatively, you can use an instruction set simulator or simplified system simulation model ("virtual platform") running on the host computer to debug your code.

You can also gauge the performance of your system by profiling the execution of your code.

Device Configuration

Once your hardware and software platforms are completed, you create a configuration bitstream for the target FPGA device. For prototyping, you can download the bitstream along with any software you wish to run on your embedded platform while connected to your host computer. For production, you store your configuration bitstream and software in a non-volatile memory connected to the FPGA.

An Introduction to EDK Tools and Utilities

Table 1-1describes the tools and utilities supported in EDK, and Figure 1-2, page 27 shows how the tools operate together to create an embedded system. The remainder of this section provides an overview of each tool, with references for obtaining additional information.

Table 1-1: **EDK Tools and Utilities**

Design Environments	
Xilinx Platform Studio (XPS)	An integrated design environment (GUI) in which you can create your complete embedded design.
Xilinx Software Development Kit (SDK)	An integrated design environment (GUI), complementary to XPS, that helps you with the development of software application projects.
EDK Command Line or "no window" Mode	Allows you to run embedded design flows or change tool options from a command line.
Hardware Development	
The Base System Builder (BSB) Wizard	Allows you to create a working embedded design quickly, using any features of a supported development board or using basic functionality common to most embedded systems. Xilinx recommends using BSB for initial project creation.

Table 1-1: **EDK Tools and Utilities *(Continued)***

The Create and Import IP Wizard	Assists you in adding your own peripheral(s) to a design. Creates associated directories and data files, ensuring that the peripheral is recognized by the EDK tools.
Configure Coprocessor Wizard	Helps you add a coprocessor to a CPU. Applies to MicroBlaze and Virtex™-4 based PowerPC designs only.
Platform Generator (Platgen)	Constructs the programmable system on a chip in the form of HDL and implementation netlist files.
Software Development	
Library Generator (Libgen)	Constructs a software platform comprising a customized collection of software libraries, drivers, and OS.
GNU Compiler Tools (GCC)	Builds a software application based on the platforms created by the Library Generator.
Verification	
Debug Configuration Wizard	Automates hardware and software platform debug configuration tasks common to most designs.
Xilinx Microprocessor Debugger (XMD)	Opens a shell for software download and debugging. Also provides a channel through which the GNU debugger accesses the device.
GNU Debugger (GDB)	GUI for debugging software on either a simulation model or target device.
Simulation Model Generator (Simgen)	Generates the hardware simulation model and the compilation script file for simulating the complete system.
Simulation Library Compiler (CompEDKLib)	Compiles the EDK Simulation Libraries for the target simulator, as required, before starting behavioral simulation of the design.
Virtual Platform Generator (VPgen)	Generates a cycle-accurate model of an embedded system in the form of a binary executable for software debugging and profiling. You can use VPgen when actual hardware is not available.
Bus Functional Model Compiler (BFM)	Helps simplify the verification of a custom peripheral by creating a model of the bus environment to use in place of the actual embedded system.
Device Configuration	
Bitstream Initializer (Bitinit)	Updates an FPGA configuration bitstream to initialize the on-chip instruction memory with the software executable.

Table 1-1: **EDK Tools and Utilities *(Continued)***

System ACE File Generator (GenACE)	Generates a Xilinx System ACE™ configuration file based on the FPGA configuration bitstream and software executable to be stored in a non-volatile device in a production system.
Flash Memory Programmer	Allows you to use your target processor to program on-board Compact Flash Interface (CFI)-compliant parallel flash devices with software and data.
Miscellaneous	
Format Revision (revup) Tool and Version Management Wizard	The revup tool updates the design files (the MHS, for example) to their current format. The Version Management Wizard helps migrate IPs and drivers created with an earlier EDK release to the latest version.
LibXil Memory File System Generator (LibXil MFS)	Creates an MFS memory image on a host system that can subsequently be downloaded to the embedded system memory.
Platform Specification Utility	Automates the generation of Microprocessor Peripheral Definition (MPD) data files required to create EDK-compliant custom peripherals.

Xilinx Platform Studio (XPS)

XPS provides an integrated environment for creating software and hardware specification flows for embedded processor systems based on MicroBlaze and PowerPC processors. It also provides an editor and a project management interface to create and edit source code. XPS offers customization of tool flow configuration options and provides a graphical system editor for connection of processors, peripherals, and buses. XPS is available on Windows®, Solaris®, and Linux platforms. There is also a batch mode invocation of XPS available.

From XPS, you can run all embedded system tools needed to process hardware and software system components. You can also perform system verification within the XPS environment.

XPS offers the following features:

- Ability to add cores, edit core parameters, and make bus and signal connections to generate an MHS file
- Ability to generate and modify the MSS file
- Support for all tools described in Table 1-1.
- Ability to generate and view a system block diagram and/or design report
- Multiple-user software applications support
- Project management
- Process and tool flow dependency management

Refer to the Xilinx Platform Studio online help for details on using the XPS GUI.

Xilinx Software Development Kit (SDK)

The Xilinx Platform Studio SDK is a complementary GUI to XPS (Xilinx Platform Studio) and provides a development environment for software application projects. SDK is based on the Eclipse open source standard. Platform Studio SDK features include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic MAKE file generation.Error navigation
- Well integrated environment for seamless debugging and profiling of embedded targets
- Source code version control

For more information about SDK, see the SDK online help.

EDK Command Line or “no window” Mode

For information on running EDK from a command line, see Chapter 15, “Command Line (no window) Mode.”

The Base System Builder (BSB) Wizard

The Base System Builder (BSB) wizard helps you quickly build a working system. Some embedded design projects can be completed using the BSB Wizard alone. For more complex projects, the BSB Wizard provides a baseline system, which you can then customize to complete your embedded design. For efficiency in project creation, Xilinx recommends using the BSB Wizard in either case.

Based on the board you have selected, BSB allows you to select and configure basic system elements such as processor type, debug interface, cache configuration, memory type and size, and peripheral. For each option, functional default values are preselected in the wizard.

If your target development is not available or not currently supported by the BSB, you can select the Custom board option instead of selecting a target board. Using this option, you must specify the individual hardware devices that you expect to have on your custom board. To run the generated system on a custom board, you must enter the FPGA pin location constraints into the User Constraints File (UCF). If a supported target board is selected, BSB automatically inserts these constraints into the UCF.

When you exit the BSB, it creates a Microprocessor Hardware Specification (MHS) file and a Microprocessor Software Specification (MSS) file, and loads them into your XPS project. You can then further enhance the design in Xilinx Platform Studio (XPS).

The BSB also optionally generates one or more software projects. Each project contains a sample application and linker script that can be compiled and run on the hardware for the target development board. Each application is designed to illustrate system aliveness and perform simple and basic testing of some of the hardware. The contents of each test application might vary depending on the components in your system. XPS supports multiple software projects for every hardware system, each of which contains its own set of source files and linker script.

For detailed information on using the features provided in the BSB Wizard, see the Xilinx Platform Studio online help.

The Create and Import IP Wizard

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

In the *Create* mode, the Create and Import Peripheral Wizard creates a number of files. Some are templates that help you implement your peripheral without your 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 the template file and using various auxiliary design support files that are output by the wizard, you can quickly get started designing your custom logic.

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

When you create or import a peripheral, MPD (Microprocessor Peripheral Definition) and PAO (Peripheral Analyze Order) files are automatically generated. The MPD file defines the interface for the peripheral, and the PAO file tells other tools (Platgen, Simgen, for example) which HDL files are required for compilation (synthesis or simulation) for the peripheral and in what order. For more information about MPD and PAO files, see the *Platform Specification Format Reference Manual.*

For detailed information on using the features provided in the Create and Import Peripheral Wizard, see the Xilinx Platform Studio online help.

Configure Coprocessor Wizard

This tool is available only if you have included a MicroBlaze™ device in your design, or if you are using a Virtex™-4 device in combination with a PowerPC™405 processor.

The Coprocessor Wizard helps add and connect a coprocessor to a CPU. A coprocessor is a hardware module that implements a user-defined function in the FPGA fabric and connects to the processor through the Fast Simplex Link (FSL) interface.

FSL channels are dedicated 32-bit, point-to-point communication interfaces implemented using FIFOs.

For More Information

For details on the Fast Simplex Link (FSL), refer to the FSL bus documentation, the *MicroBlaze Processor Reference Guide,* and the FCB to FSL Bridge document.

For details on the APU, refer to the "PowerPC 405 APU Controller" chapter of the *PowerPC 405 Block Reference Manual*.

For instructions on using the Coprocessor Wizard, refer to the Xilinx Platform Studio online help.

Platform Generator (Platgen)

Platgen compiles the high-level description of your embedded processor system into an HDL netlist that can be implemented in a target FPGA device.

The embedded hardware platform typically consists of one or more processors and a variety of peripherals and memory blocks, interconnected via processor buses. It also has port connections to the outside world. Each of the peripheral cores (or *processor IP)* has a number of parameters that you can adjust to customize its behavior. These parameters also

define the address map of your peripherals and memories. Because EDK lets you select from various optional features, the FPGA needs only to implement the subset of functionality required by your application.

The hardware platform description is maintained in a file known as the Microprocessor Hardware Specification (MHS). The MHS is the principal source file representing the hardware component of your embedded system. It is stored as ASCII text.

As shown in Figure 1-2, page 27, Platgen reads the MHS file as its primary design input. Platgen also reads various processor core (pcore) hardware description files (MPD, PAO) from the EDK library and any user IP repository. Platgen produces the top-level HDL design file for the embedded system that stitches together all the instances of parameterized pcores contained in the system. In the process, it resolves all the high-level bus connections in the MHS into the actual signals required to interconnect the processors, peripherals and on-chip memories. It also invokes the XST (Xilinx Synthesis Technology) compiler to synthesize each of the instantiated pcores. (The system-level HDL netlist produced by Platgen is used as part of the FPGA implementation process.) In addition, Platgen generates the BMM (BRAM Memory Map) file which contains addresses of various on-chip BRAM memories used. This file is used later for initializing the BRAMs with software.

Refer to Chapter 2, "Platform Generator (Platgen)," for more information.

For more information on the MHS files, see the "Microprocessor Hardware Specification (MHS)" chapter of the *Platform Format Specification Reference Manual.*

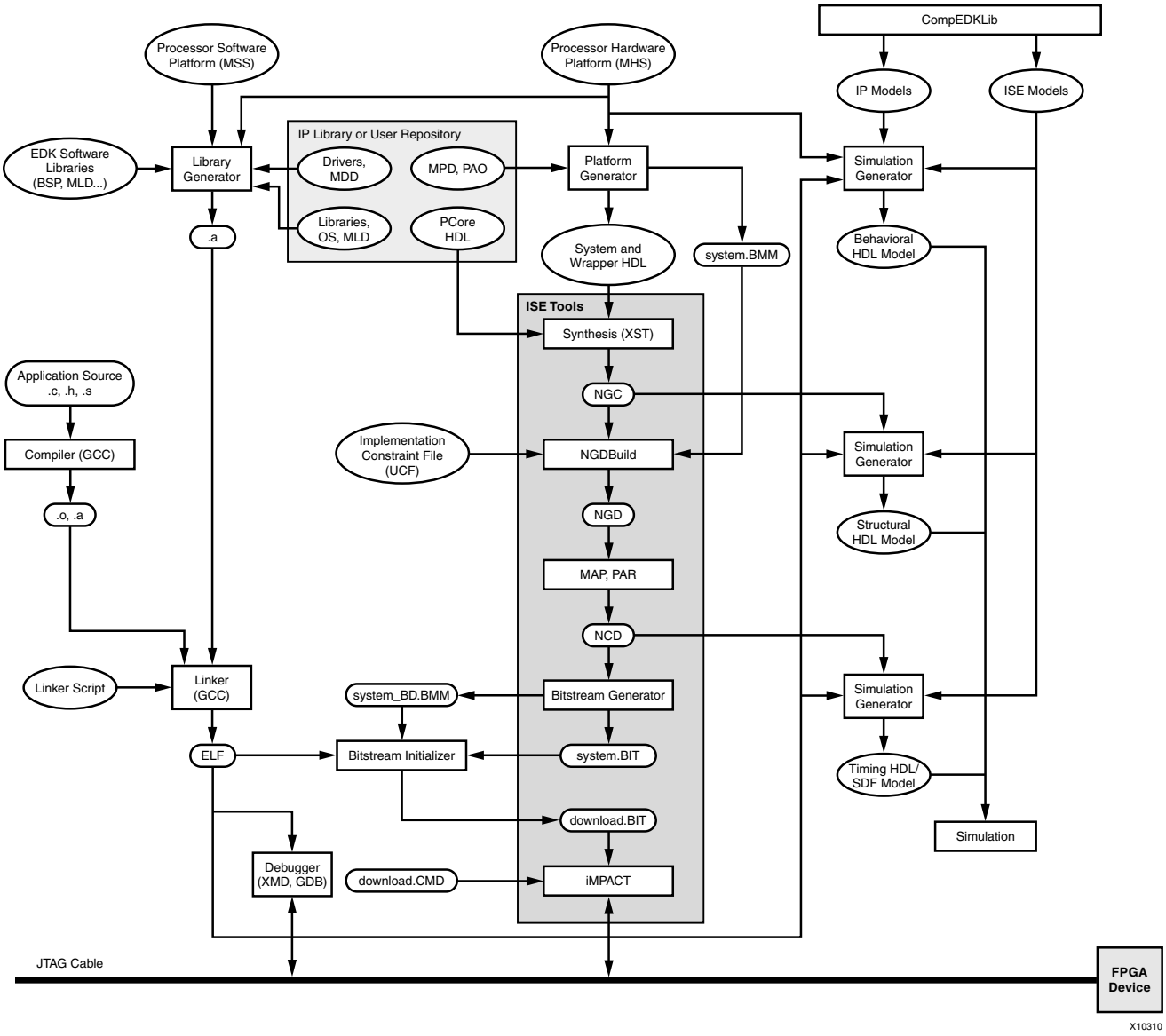

Figure 1-2: **Embedded Development Kit (EDK) Tools Architecture**

Library Generator (Libgen)

Libgen configures libraries, device drivers, file systems and interrupt handlers for the embedded processor system, creating what is called a "software platform." The embedded software platform defines, for each processor, the drivers associated with the peripherals you include in your hardware platform (the board support package), selected libraries, standard input/output devices, interrupt handler routines, and other related software features. Your XPS project further defines software applications to run on each processor, which are based on the software platform.

XPS maintains your software platform description in the Microprocessor Software Specification (MSS) file. The MSS file, which is an editable text file, together with your software applications, are the principal source files representing the software component

of your embedded system. Together with libraries and drivers from the EDK installation, along with any custom libraries and drivers for custom peripherals you provide, EDK is able to compile your applications, including software components specified in the MSS, into executable ELF files ready to run on your processor hardware platform.

As shown in Figure 1-2, page 27, Libgen reads both the MSS and MHS as its primary design inputs. Libgen also reads selected EDK libraries and various processor core (pcore) software description files (MDD and driver code) from the EDK library and any user IP repository.

Refer to Chapter 4, "Library Generator (Libgen)" and the Xilinx Platform Studio online help for more information. For more information on Libraries and Device Drivers, refer to the "Xilinx Microkernel (XMK)" section of the *OS and Libraries Document Collection*.

GNU Compiler Tools (GCC)

XPS calls GNU compiler tools for compiling and linking application executables for each processor in the system. For applications targeting a MicroBlaze processor, XPS runs the mb-gcc compiler. For applications targeting a PowerPC processor, XPS runs the powerpc-eabi-gcc compiler. As shown in Figure 1-2, page 27, the compiler can read a set of C source and header files or assembler source files for the targeted processor. The linker combines the compiled applications with selected libraries and produces the executable file in ELF format. The linker also reads a linker script, which is either the default linker script generated by the tools or one provided by the user. Refer to Chapter 10, "GNU Compiler Tools," for more information. Also refer to Appendix A, "GNU Utilities" for other useful tools.

Debug Configuration Wizard

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

You can instantiate a ChipScope™ core to monitor OPB (on-chip peripheral bus), PLB (processor local bus), or any other system-level signals. In addition, you can configure the parameters of an existing ChipScope core for hardware debugging. You can also provide JTAG based virtual input and output. To configure the software for debugging you can set the processor debug parameters. When co-debugging is enabled for a ChipScope core, you can set up mutual triggering between the software debugger and the hardware signals. The JTAG interface can be configured to transport UART signals to the Xilinx Microprocessor Debugger (XMD).

For detailed information on using the features provided in the Debug Configuration Wizard, see the Xilinx Platform Studio online help.

Xilinx Microprocessor Debugger (XMD)

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. As shown in Figure 1-2, page 27 the debugger utility (XMD) reads the application executable (ELF). For debugging on a physical FPGA, XMD communicates over the same download cable as used to configure the FPGA with a bitstream.

Refer to Chapter 12, "Xilinx Microprocessor Debugger (XMD)," for more information.

GNU Debugger (GDB)

The GNU Debugger (GDB) is a powerful yet flexible tool that provides a unified interface for debugging and verifying MicroBlaze and PowerPC systems during various development phases. It uses Xilinx® Microprocessor Debugger (XMD) as the underlying engine to communicate to processor targets.

Refer to Chapter 11, "GNU Debugger (GDB)" for more information.

Simulation Model Generator (Simgen)

The Simulation Platform Generation tool (Simgen) generates and configures various simulation models for the hardware. As shown in Figure 1-2, page 27, for generating a behavioral model, Simgen takes an MHS file as its primary design input. For generating structural or timing models, Simgen takes its primary design input from the post-synthesis or post-place-and-route design database, respectively. Simgen also reads the embedded application executable (ELF) for each processor to initialize on-chip memory, thus allowing the modeled processor(s) to execute their software code during simulation.

Refer to Chapter 3, "Simulation Model Generator (Simgen)" for more information.

Simulation Library Compiler (CompEDKLib)

CompEDKLib compiles the EDK HDL-based simulation libraries using the tools provided by various simulator vendors. This utility can operate in both the GUI and batch modes. In the GUI mode. it allows you to compile the Xilinx libraries (in your ISE installation) using CompXLib and the libraries available in EDK.

For more information about CompEDKLib, see "CompEDKLib Utility" in Chapter 3. For instructions on compiling simulation libraries, refer to the Xilinx Platform Studio online help.

Virtual Platform Generator (VPgen)

The virtual platform is a cycle-level simulation model of the hardware system. The virtual platform can be used to debug and profile software application code on the host machines, eliminating the need to get the actual hardware working on a prototyping board.

For more information, see Chapter 5, "Virtual Platform Generator (VPgen)."

Bus Functional Model Compiler (BFM)

Bus Functional Simulation simplifies the verification of hardware components that attach to a bus. For more information on bus functional models, see the document "BFM Simulation in Platform Studio."

Bitstream Initializer (Bitinit)

The Bitstream Initializer (Bitinit) tool initializes the on-chip BRAM memory connected to a processor with its software information. This utility reads hardware-only bitstream produced by the ISE tools (`system.bit`), and outputs a new bitstream (`download.bit`) which includes the embedded application executable (ELF) for each processor. The utility uses the BMM file, originally generated by Platgen and updated by the ISE tools with physical placement information on each BRAM block in the FPGA. Internally, the Bitstream Initializer tool uses the Data2MEM utility provided in ISE to update the

bitstream file. See Figure 1-2, page 27, to see how the Bitinit tool fits into the overall system architecture. Refer to Chapter 8, "Bitstream Initializer (BitInit)," for more information on this tool.

System ACE File Generator (GenACE)

Generate Xilinx® System ACE™ configuration files from an FPGA bitstream and ELF/data files. The ACE file generated can be used to configure the FPGA, initialize BRAM, initialize external memory with valid program or data, and bootup the processor in a production system. EDK provides a Tool Command Language (Tcl) script, `genace.tcl`, which uses XMD commands to generate ACE files. ACE files can be generated for PowerPC and MicroBlaze with MDM systems.

For more information see Chapter 13, "System ACE File Generator (GenACE)."

Flash Memory Programmer

The programming solution is designed to be generic and targets a wide variety of flash hardware and layouts.

See Chapter 9, "Flash Memory Programming."

Format Revision (revup) Tool and Version Management Wizard

The Format Revision Tool (revup) updates an existing EDK project to the current version. The revup tool performs format changes only. It does not update your design. Backups of existing files, such as XMP, MHS, and MSS, are performed before the format changes are applied. Refer to Chapter 7, "Version Management Tools," for more information.

The Version Management Wizard appears automatically when an older project is opened in a newer version of EDK (for example, when a project created in EDK 7.1.1 is opened in version 8.1).

The Version management wizard is invoked after format revision has been performed. The wizard provides information about any changes in Xilinx Processor IPs (pcores) used in the design. If a new compatible version of an IP is available, then the wizard also prompts you to update to the new version.

For More Information

For instructions on using the Version Management Wizard, see Chapter 7, "Version Management Tools," and the "Version Management Wizard" topic in the Xilinx Platform Studio online help.

LibXil Memory File System Generator (LibXil MFS)

Provides the capability to manage program memory in the form of file handles. You can create directories and have files within each directory. The file system can be accessed from high-level C language through function calls specific to the file system.

For more information about XilMFS, refer to the "LibXil Memory File System (MFS)" section of the *OS and Libraries Document Collection*.

Platform Specification Utility

Enables automatic generation of Microprocessor Peripheral Definition (MPD) files required to create an IP core compliant with the Embedded Development Kit (EDK). Features provided by this tool can be used with the help of the Create and Import Peripheral Wizard in Xilinx Platform Studio (XPS).

For more information, see Chapter 6, "Platform Specification Utility (PsfUtility)."

Project Creation and Management

Project information is saved in a Xilinx Microprocessor Project (XMP) file. An XMP file consists of the location of the MHS (Microprocessor Hardware Specification) file, the MSS (Microprocessor Software Specification) file, and the C source and header files that must 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 must be run.

Controlling the EDK Flow (Advanced)

This section contains advanced details for controlling how EDK processes your embedded design. This information is common to both XPS and the command line "no windows" batch processing mode.

MAKE Files

You can specify your own MAKE file. Currently, the MAKE file is split into two parts:

- The main MAKE file: `<projname>.make`
- The include MAKE file: `<projname>_incl.make`.

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

```
include system_incl.make
```

This causes the macros defined in `<projname>_incl.make` to be visible in `<projname>.make`. XPS always writes out both MAKE files. However, you can choose not to use the `<projname>.make` file for your flow. Instead, you can specify your own MAKE file. The MAKE file specified must be named differently from the two MAKE files generated by XPS. You are expected to include the `<projname>_incl.make` in your own MAKE file as well. This way, changes you apply to any options and settings in XPS are reflected in your own MAKE file. Typically, you would generate the `<projname>.make` file once and then copy and modify it for your own purposes.

You must update your MAKE file whenever you make a significant change in your design.

The following affect MAKE file structure:

- Adding, deleting, or renaming a processor.
- Adding, deleting, or renaming a software application.
- Changing the choice of implementation tool between ISE™ (Project Navigator), Xplorer (in XPS), and Xflow (in XPS).

The ACE file generation command might be changed if you change the number of processors in your design or if you add or 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.

Implementation and Download Control Files

If you choose to have EDK run the implementation tools using Xflow from ISE, EDK expects a certain directory structure in the project directory. For each project, you must provide the User Constraints File (UCF). This file resides in the `data` directory in the project directory and has the name `<project_name>.ucf`. You must also provide an iMPACT script file. This file resides in the `etc` directory and is called `download.cmd`. If these files do not exist, XPS prompts you 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 the `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, you can modify the two files. When a new project is created, if the `data` and `etc` directories do not exist, XPS creates these empty directories in the project directory.

XILINX®

Chapter 2

Platform Generator (Platgen)

Hardware generation is accomplished with the Platform Generator (Platgen) tool. Platgen constructs the programmable system on a chip in the form of hardware netlists (HDL and implementation netlist files). Platgen creates a hardware platform using the Microprocessor Hardware Specification (MHS) file as input. In addition to netlist files in various formats such as NGC and EDIF, Platgen creates support files for downstream tools and top-level HDL wrappers to allow you to add other components to the automatically generated hardware platform.

Note: After running Platgen, FPGA implementation tools (in ISE™) are run to complete the implementation of the hardware. Typically, XPS spawns the Project Navigator front end for the implementation tools, allowing full control over the implementation. At the end of the ISE flow, a bitstream is generated to configure the FPGA. This bitstream includes initialization information for block RAM memories on the FPGA chip. If your code or data must be placed on these memories at startup, the Data2MEM tool in the ISE tool set is used to update the bitstream with code/data information obtained from your executable files, which are generated at the end of the Software application creation and verification flow.

This chapter contains the following sections:

- Tool Usage
- Tool Options
- Load Path
- Output Files
- About Memory Generation
- Reserved MHS Parameters
- Synthesis Netlist Cache
- Restrictions

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 for more information.

Tool Usage

Run Platgen as follows:

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

Tool Options

The following options are supported in the current version.

Table 2-1: **Platgen Syntax Options**

Option	Command	Description
Help	**-h, -help**	Displays the usage menu and then exits without running the Platgen flow.
Version	**-v**	Displays the version number of Platgen and then exits without running the Platgen flow.
Filename	**-f <*filename*>**	Reads command line arguments and options from file.
Integration Style	**-intstyle ise|xflow|silent**	Indicate contextual information when invoking Xilinx applications within a flow or project environment.
Language	**-lang verilog|vhdl**	HDL language output. Default: `vhdl`.
Log output	**-log <*logfile*[.log]>**	Specify the log file. Default: `platgen.log`. This option is currently not implemented.
Library Path for user peripherals and driver repositories	**-lp <*library_path*>**	Add <*library_path*> to the list of IP search directories. A library is a collection of repository areas.
Output directory	**-od <*output_dir*>**	Output directory path. Default: The current directory.
Part name	**-p <*partname*>**	Use the specified part type to implement the design.
Synthesis project files	**-st xst|none**	Generate synthesis project files. Default: `xst` Platgen produces a synthesis vendor-specific project file. Deprecated.
Instance name	**-ti <*instname*>**	Top-level instance name.
Top-level module	**-tm <*top_module*>**	Name the top-level module as desired.
Top level	**-toplevel yes|no**	Input design represents a whole design or a level of hierarchy. Default: `yes`. Deprecated.

Load Path

Refer to Figure 2-1 for a depiction of the peripheral directory structure.

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

- Use the current directory (from which Platgen was launched).
- Set the EDK tool **-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`.

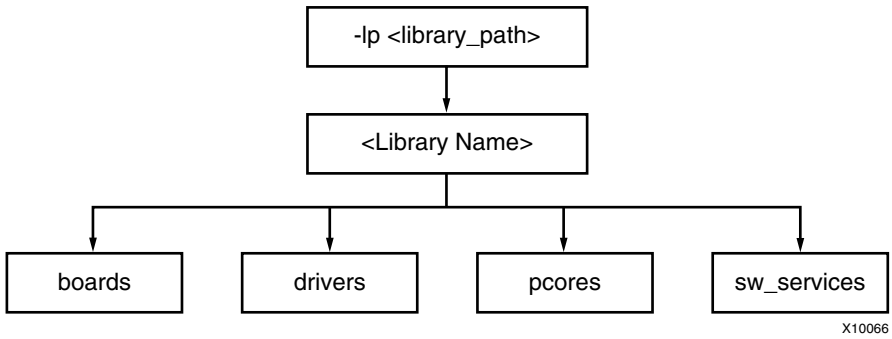

Figure 2-1: **Peripheral Directory Structure**

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.

system`_stub.[vhd|v]`

This is the toplevel template HDL file of the instantiation of the system. Use this file as a starting point for your own toplevel HDL file. 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 implementation netlist file of the peripheral:

`peripheral_wrapper.ngc`

Synthesis Directory

The synthesis directory contains the following synthesis project file:

`system.[prj|scr]`

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; the associated options of the BRAM controller define the resulting BRAM block.

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 data width and address width to the same BRAM block instance. For example, you can connect an OPB BRAM controller and LMB BRAM controller to the same BRAM block. However, you cannot connect an OPB BRAM controller and a PLB BRAM controller to the same BRAM block instance. You can connect an 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, define the different depth sizes of memory. Refer to the "Microprocessor Hardware Specification (MHS)" chapter in the *Platform Specification Format Reference Manual* for more information.

Only predefined memory sizes are allowed. A predefined memory size is limited to the number of Select BlockRAM available when the memory size is of powers of two. The smallest memory size must support byte-write access.

Table 2-2: **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
Spartan-3e	8, 16, 32, 64	16, 32, 64, 128
Virtex™	2, 4, 8, 16	4, 8, 16, 32
Virtex-E	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
Virtex-5	4, 8, 16, 32, 64, 128, 256	8, 16, 32, 64,128, 256, 512

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

For example, for a memory size of 4 kBytes on a Virtex device, Platgen uses eight Select BlockRAMs to support byte-write access.

BMM Policy

A BlockRAM Memory Map (BMM) 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 is as follows:

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

Bitgen outputs `<system>_bd.bmm`, which 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 about the platform that is generated. The following table lists the reserved parameter names:

Table 2-3: **Automatically Expanded Reserved Parameters**

Parameter	Description
C_FAMILY	FPGA device family
C_PACKAGE	Device package
C_SPEEDGRADE	Device speed grade
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_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
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.

Restrictions

In the Platgen flow, vector slicing is not supported.

Chapter 3

Simulation Model Generator (Simgen)

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

- Simgen Overview
- Simulation Libraries
- CompXLib Utility
- CompEDKLib Utility
- Simulation Models
- Simgen Syntax
- Output Files
- Memory Initialization
- Simulating Your Design
- Restrictions

Simgen Overview

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 MHS file. Refer to the "Microprocessor Hardware Specification (MHS)" chapter in the *Platform Specification Format Reference Manual* for more information.

For more information about simulation basics and for discussions of behavioral, structural, and timing simulation methods, refer to the *Platform Studio Online Help*.

Simulation Libraries

EDK simulation netlists use low-level hardware primitives available in Xilinx® FPGAs. Xilinx provides simulation models for these primitives in the libraries listed in this section.

The libraries described in the following sections 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

Xilinx Libraries can be compiled using the CompXLib utility. Refer to the "Simulating Your Design" chapter of the *Synthesis and Verification Design Guide* in your ISE™ distribution to learn more about compiling and using Xilinx simulation libraries. Xilinx ISE provides the following libraries for simulation:

UNISIM Library

The UNISIM Library 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 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

The SIMPRIM Library is used for timing simulation. It includes all the Xilinx Primitives Library components used by Xilinx implementation tools.

Timing simulation models generated by Simgen instantiate SIMPRIM library components.

XilinxCoreLib Library

The Xilinx CORE Generator™ is a graphical Intellectual Property (IP) design tool for creating high-level modules like FIR Filters, FIFOs, CAMs, and 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

The EDK Library is 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 overall compile time. The EDK IP components library is provided for VHDL only and may be encrypted.

The Xilinx CompEDKLib utility deploys compiled models for EDK IP components into a common location. Unencrypted EDK IP components can be compiled using CompEDKLib. Precompiled libraries are provided for encrypted components.

CompXLib Utility

Xilinx provides the CompXLib utility to compile the HDL libraries for all Xilinx-supported simulators. CompXLib compiles the UNISIM, SIMPRIM and XilinxCoreLib libraries for all supported device architectures using the tools provided by the simulator vendor.

To compile your HDL libraries using CompXLib, follow these steps. You must have an installation of the Xilinx implementation tools:

Run CompXLib with the **-help** option if you need to display a brief description for the available options:

```
compxlib -help
```

The CompXLib tool uses the following syntax:

```
compxlib -s <simulator> -f <family[:lib],<family[:lib],...|all>
[-l <language>]
[-o <compxlib_output_directory>]
[-w]
[-p <simulator_path>]
```

Note: Each simulator uses certain environment variables that you must set before invoking CompXLib. Consult your simulator documentation to ensure that the environment is properly set up to run your simulator.

Note: Make sure you use the **-p <simulator_path>** option to point to the directory where the ModelSim executable is, if it is not in your path.

The following is an example of a command for compiling Xilinx libraries for MTI_SE:

```
> compxlib -s mti_se -f all -l vhdl -w -o .
```

This command compiles all the necessary Xilinx libraries into the current working directory.

For More Information

Refer to the "Verifying Your Design" chapter of the *Synthesis and Verification Design Guide* included in your ISE distribution for additional information on using and compiling Xilinx simulation libraries.

CompEDKLib Utility

Before starting behavioral simulation of your design, you much compile the EDK Simulation Libraries for the target simulator. For this purpose, Xilinx provides the CompEDKLib utility.

CompEDKLib compiles the EDK HDL-based simulation libraries using the tools provided by various simulator vendors. This utility can operate in both the GUI and batch modes. In the GUI mode it allows you to compile the Xilinx libraries (in your ISE installation) using CompXLib and the libraries available in EDK. Refer to the *Platform Studio Online Help* for more instructions on compiling simulation libraries.

Usage

```
compedklib [ -h ] [ -o output-dir-name ] [ -lp repository-dir-name ]
     [ -E compedklib-output-dir-name ] [ -lib core-name ]
     [ -compile_sublibs ] [ -exclude deprecated ]
     -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 simulators are MTI PE/SE and NCSIM.

CompEDKLib Command Line Examples

Use Case I: Launching the GUI to Compile the Xilinx and EDK Simulation Libraries

```
compedklib
```

No options are required. This launches the same GUI as when selecting **Options** → **Compile Simulation Libraries** in XPS when no project is open.

Note: This is the only mode of **`compedklib`** that also compiles the Xilinx Libraries. All other modes compile only the EDK libraries.

Use Case II: 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> -exclude deprecated
```

In this case the pcores available in the EDK install are compiled and stored in `<compedklib-output-dir-name>`. The value of 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, or obsolete state. Adding **-exclude deprecated** has the effect of not compiling deprecated cores. If you have deprecated cores in your design, do not use the **`-exclude deprecated`** option.

Use Case III: Compiling HDL Sources in Your Own Repository

If you have your own repository of EDK-style pcores, you can 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, **`-E`** accounts for the possibility that some of the pcores in your repository might 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 is read but only the pcores indicated by the **`-lib`** options are compiled.

You can add **-compile_sublibs** to the above to compile the pcores that the indicated pcore depends on.

Other Details

- If the simulator is not indicated, then MTI is assumed.
- You can supply multiple **-X** and **-E** arguments. The order of arguments 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 into `<compedklib-output-dir-name>`.
- If your pcores are in your XPS project, then Use Case 2 does not apply. Simgen creates the scripts to compile them.
- Only VHDL is supported.
- The execution log is available in `compedklib.log`.
- Beginning in EDK 7.1, the file indicated by your `MODELSIM` environment variable is not modified by CompEDKLib. However, the simulation scripts generated by Simgen will modify the file pointed to by the `MODELSIM` variable.

Simulation Models

This section describes how and when each of three FPGA simulation models are implemented. At the certain points in the design process, Simgen creates an appropriate simulation model, as shown in Figure 3-1. Instructions for creating simulation models using XPS Batch Mode are also included in this section.

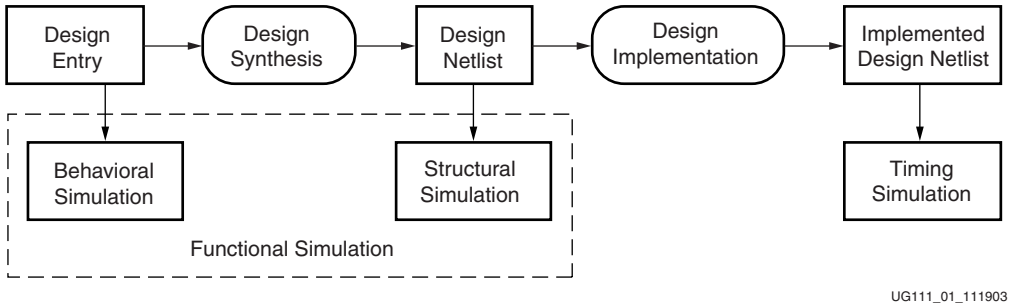

Figure 3-1: **FPGA Design Simulation Stages**

Behavioral Models

To create a behavioral simulation model as displayed in Figure 3-2, Simgen requires an MHS file as input. Simgen creates a set of HDL files that model the functionality of the design. Optionally, Simgen can generate a compile script for a specified vendor simulator. If specified, Simgen can generate HDL files with data to initialize BRAMS associated with any processor that exists in the design. This data is obtained from an existing Executable and Link Format (ELF) file.

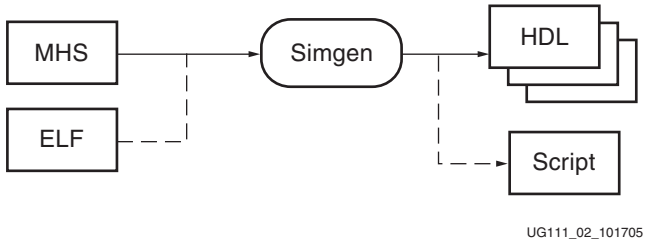

Figure 3-2: **Behavioral Simulation Model Generation**

Structural Models

To create a structural simulation model as displayed in Figure 3-3, Simgen requires an MHS file as input and associated synthesized netlist files. From these netlist files, Simgen creates 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. If specified, Simgen can generate HDL files with data to initialize BRAMS associated with any processor that exists in the design. This data is obtained from an existing ELF file.

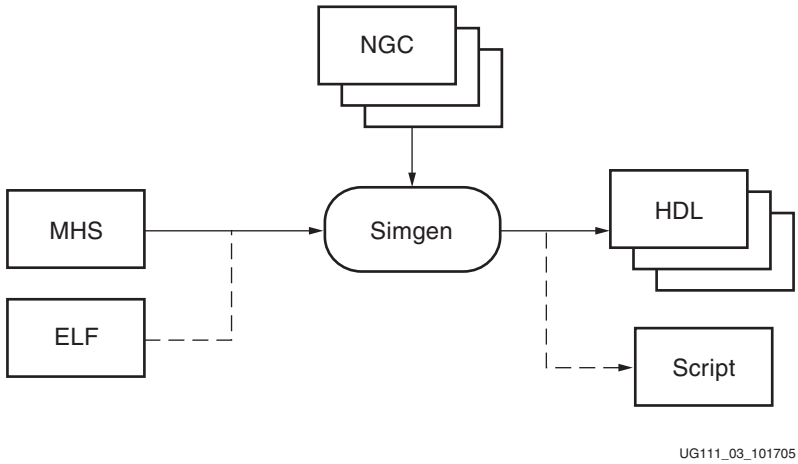

Figure 3-3: **Structural Simulation Model Generation**

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 as displayed in Figure 3-4, Simgen requires an MHS file as input and an associated implemented netlist file. From this netlist file, Simgen creates 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. If specified, Simgen can generate HDL files with data to initialize BRAMS associated with any processor that exists in the design. This data is obtained from an existing ELF file.

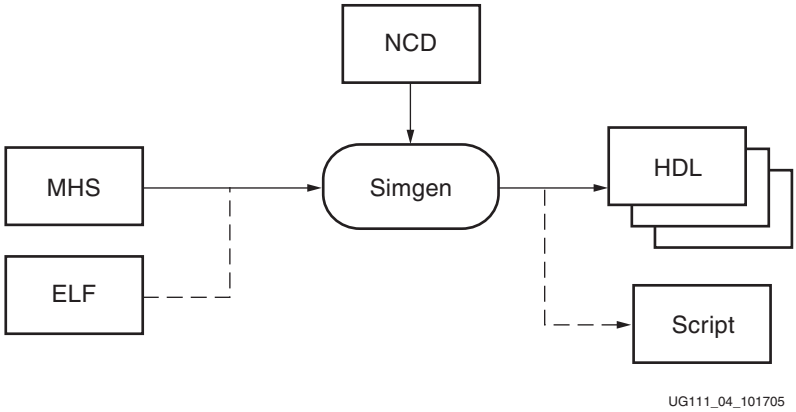

Figure 3-4: **Timing Simulation Model Generation**

Single and Mixed Language Models

Simgen allows the use of mixed language components in behavioral files for simulation. By default, Simgen takes the native language in which each component is written. Individual components cannot 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 can generate single language models by translating the HDL files that are not in the desired language. The resulting translated HDL files are structural files.

All Structural and Timing simulation models are always single language.

Creating Simulation Models Using XPS Batch

1. Open your project by loading your XMP file:

   ```
   XPS% load xmp <filename>.xmp
   ```

2. Set the following simulation values at the XPS prompt.
   a. Select the simulator of your choice using the following command:

      ```
      XPS% xset simulator [ mti | ncs | none ]
      ```

   b. Specify the path to the Xilinx and EDK precompiled libraries using the following commands:

      ```
      XPS% xset sim_x_lib <path>
      XPS% xset sim_edk_lib <path>
      ```

   c. Select the Simulation Model using the following command:

      ```
      XPS% xset sim_model [ behavioral | structural | timing ]
      ```

3. To generate the simulation model, type the following:

   ```
   XPS% run simmodel
   ```

   When the process finishes, HDL models are saved in the simulation directory.

4. To open the simulator, type the following:

   ```
   XPS% run sim
   ```

Simgen Syntax

At the prompt, run Simgen with the MHS file and appropriate options as inputs.

For example, **simgen** *system_name.mhs* [*options*]

Requirements

Verify that your system is properly configured to run the Xilinx ISE tools. 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.

Table 3-1: **Simgen Syntax Options**

Option	Command	Description
Help	**-h, -help**	Displays the usage menu and then quits.
Version	**-v**	Displays the version and then quits.
Options File	**-f *filename***	Reads command line arguments and options from file.
HDL Language	**-lang vhdl|verilog**	Specifies the HDL language: VHDL or Verilog. Default: **vhdl**
Log Output	**-log *logfile[.log]***	Specifies the log file. Default: **simgen.log**
Library Directories	**-lp *library_path***	Allows you to specify library directory paths. This option can be specified more than once for multiple library directories.
Simulation Model Type	**-m beh|str|tim**	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**	Allows or disallows 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. ***Note:*** Only valid when **-m beh** is used Default: **yes**
Output Directory	**-od *output_dir***	Specifies the project directory path. The default is the current directory.

Table 3-1: **Simgen Syntax Options *(Continued)***

Option	Command	Description
Target Part or Family	**-p *partname***	Allows you to target a specific part or family. This option must be specified.
Processor ELF Files	**-pe *proc_instance elf_file* {*elf_file*}**	Specifies a list of ELF files to be associated with the processor with instance name as defined in the MHS.
Simulator	**-s mti|ncs**	Generates compile script and helper scripts for vendor simulator: mti or ncs. **mti** - ModelSim **ncs** - NcSim
Source Directory	**-sd *source_dir***	Specifies the source directory to search for netlist files.
Testbench Template	**-tb**	Creates a testbench template file. Use **-ti** and **-tm** to define the design under test name and the testbench name, respectively.
Top-Level Instance	**-ti *top_instance***	When a testbench template is requested, use *top_instance* to define the instance name of the design under test. When design represents a submodule, use *top_instance* for the top-level instance name.
Top-Level Module	**-tm *top_module***	When a testbench template is requested, use *top_module* to define the name of the testbench. When the design represents a submodule, use *top_module* for the top-level entity/module name.
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***	Specifies the path to the EDK simulation libraries directory. This is the output directory of the CompEDKLib tool.
Xilinx Library Directory	**-X *xlib_dir***	Path to the 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, which is located inside the `\output_directory`. In the `\simulation` directory, there is a subdirectory for each simulation model.

For example:

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

Where `<sim_model>` is one of: `behavioral, structural, or timing`

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 (SDF) file with the appropriate block and net delays from the place and route process used only for timing simulation.
- *system_name*_setup.`[do|sh]`
  Script to compile the HDL files and load the compiled simulation models in the simulator.
- *test_harness*_setup.`[do|sh]`
  Helper script to set up the simulator and specify signals to display in a waveform window or tabular list window (ModelSim only).
- *test_harness*_wave.`[do|sv]`
  Helper script to set up simulation waveform display.
- *test_harness*_list.`do`
  Helper script to set up simulation tabular list display (ModelSim only).
- *instance*_wave.`[do|sv]`
  Helper script to set up simulation waveform display for the specified instance.
- *instance*_list.`do`
  Helper script to set up simulation tabular list display for the specified instance (ModelSim Only).

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 GNU Compiler Collection (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, run Simgen with the **`-pe`** option to generate a VHDL file. This file contains 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`** contain the data in `executable.elf`.

Verilog

For Verilog simulation models, run Simgen with the **`-pe`** option to generate a Verilog file. This file contains 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 contains the data in `executable.elf`.

Simulating Your Design

When simulating your design, there are some special considerations 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. Refer to the "Simulating Your Design" chapter in the ISE *Synthesis and Verification Design Guide* for more information. A PDF version of this document is located at `/doc/usenglish/books/docs/sim/sim.pdf` in your install area, or online at:
http://www.xilinx.com/support/sw_manuals/xilinx8/index.htm.

Helper scripts generated at the test harness (or testbench) level are simulator *setup* scripts. When run, the *setup* script performs initialization functions and displays usage instructions for creating waveform and list (ModelSim only) windows using the *waveform* and *list* scripts. The top-level scripts invoke instance-specific scripts. You may need to edit hierarchical path names in the helper scripts for test harnesses not created by Simgen.

Commands in the scripts are commented or not commented to define the set of signals displayed. Editing the top-level *waveform* or *list* scripts allows you to include or exclude all signals for an instance; editing the instance level scripts allows you to include or exclude individual port signals. For timing simulations, only top-level ports are displayed.

Restrictions

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

XILINX®

Chapter 4

Library Generator (Libgen)

This chapter describes the Library Generator utility, Libgen, needed for the generation of libraries and drivers for embedded soft processors. It also describes how you can customize peripherals and associated drivers. This chapter contains the following sections:

- Overview
- Tool Usage
- Tool Options
- Load Paths
- Output Files
- Libraries and Drivers Generation
- MSS Parameters
- Drivers
- Libraries
- OS Block
- Interrupts and Interrupt Controllers
- XMDStub Peripherals (MicroBlaze Specific)
- STDIN and STDOUT Peripherals

Overview

Libgen is generally the first tool that you run to configure libraries and device drivers. Libgen takes a Microprocessor Software Specification (MSS) file that you create. 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 the "Microprocessor Software Specification (MSS)" chapter in the *Platform Specification Format Reference Manual*.

Note: EDK includes a Format Revision tool to convert older MSS file formats to a new MSS format. Refer to Chapter 7, "Version Management Tools," for more information.

Tool Usage

To run Libgen, type the following:

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

Tool Options

The following options are supported in this version:.

Table 4-1: **Libgen Syntax Options**

Option	Command	Description
Help	**-h, -help**	Displays the usage menu and then quits.
Version	**-v**	Displays the version number of Libgen and then quits.
Log output	**-log *logfile[.log]***	Specifies the log file. Default: *libgen.log*
Architecture family	**-p *partname***	Defines the target device defined either as architecture family or partname. Use **-h** to view a list of values for the target family.
Output directory	**-od *output_dir***	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 *YOUR_PROJECT f*or convenience in the documentation.
Source directory	**-sd *source_dir***	Specifies the source directory *source_dir* for searching the input files. The default is the current working directory.
Library Path for user peripherals and driver repositories	**-lp *library_path***	Specifies a library containing repositories of user peripherals, drivers, OSs, and libraries. Libgen looks for the following: • Drivers in the directory library_path/*sub_dir*/drivers/ • Libraries in the directory library_path/*sub_dir*/sw_services/ • OSs in the directory library_path/*sub_dir*/bsp/ Here *sub_dir* is a subdirectory under library_path.

Table 4-1: **Libgen Syntax Options *(Continued)***

Option	Command	Description
MHS file	**-mhs *mhsfile.mhs***	Specifies the Microprocessor Hardware Specification (MHS) file to be used for Libgen. The following is the order Libgen uses to search and locate `mhsfile.mhs`: 1. Current working directory (`YOUR_PROJECT/`). 2. If no **-mhs** option is used, look in the MSS file for the parameter `HW_SPEC_FILE` to get the `mhsfilename`. 3. 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`.
Libraries	**-lib**	Use this option to copy libraries and drivers but not to compile them.

Load Paths

Refer to Figure 4-1 and Figure 4-2 for diagrams of the directory structure for drivers, libraries, and Operating Systems (OSs).

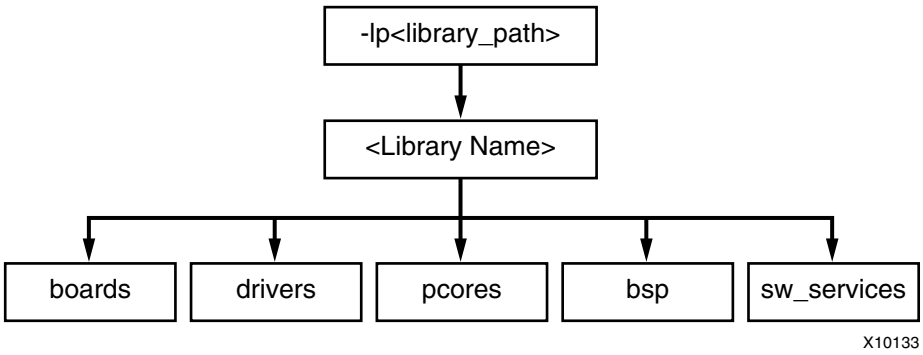

Figure 4-1: **Directory Structure of Peripherals, Drivers, Libraries, and OSs**

Unix System Load Paths

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

- Drivers: `$XILINX_EDK/sw/`*`Library Name`*`/drivers`
- Libraries: `$XILINX_EDK/sw/`*`Library Name`*`/sw_services`
- OSs: `$XILINX_EDK/sw/`*`BSP Name`*`/bsp`

PC System Load Paths

On a PC, the drivers and libraries reside in the following locations:

- Drivers: `%XILINX_EDK%\sw\`*`Library Name`*`\drivers`
- Libraries: `%XILINX_EDK%\sw\`*`Library Name`*`\sw_services`
- OSs: `%XILINX_EDK%\sw\`*`BSP Name`*`\bsp`

Additional Directories

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

- Use the current working directory from which Libgen was launched.
- Set the EDK tool option **-lp**. Libgen looks for drivers, OSs, and libraries under each of the subdirectories of the path specified in the **-lp** option.

Search Priority Mechanism

Libgen uses a search priority mechanism to locate drivers and libraries, as follows:

1. Search the current working directory:
   a. Drivers: Search for drivers inside the `drivers` or `pcores` directory in the current working directory in which you run Libgen.
   b. Libraries: Search for libraries inside the `sw_services` directory in the current working directory in which you run Libgen.
   c. OS: Search for OSs inside the `bsp` directory in the current working directory from which you run Libgen.
2. Search the repositories under the library path directory specified using the **`-lp`** option:
   a. Drivers: Search one of the following, as specified by the **`-lp`** option:
      - *`library_path`*`/`*`Library Name`*`/drivers` and *`library_path`*`/`*`Library Name`*`/pcores` (UNIX)
      - *`library_path`*`\`*`Library Name`*`\drivers` and *`library_path`*`\`*`Library Name`*`\pcores` (PC)
   b. Libraries: Search one of the following 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*:
      - *`library_path`*`/`*`Library Name`*`/sw_services` (UNIX)
      - *`library_path`*`\`*`Library Name`*`\sw_services` (PC)
   c. OSs: Search one of the following as specified by the **`-lp`** option. In this case, *library_path* is the directory argument to the **`-lp`** option and *OS Name* is a subdirectory under *library_path*:
      - *`library_path`*`/`*`OS Name`*`/bsp` (UNIX)
      - *`library_path`*`\`*`OS Name`*`\bsp` (PC)

3. Search the EDK install area:
   a. Drivers: Search one of the following:
      - `$XILINX_EDK/sw/`*`Library Name`*`/drivers` (UNIX)
      - `%XILINX_EDK%\sw\`*`Library Name`*`\drivers` (PC)
   b. Libraries: Search `$XILINX_EDK/sw/`*`Library Name`*`/sw_services` (UNIX) and `%XILINX_EDK%\sw\`*`Library Name`*`\sw_services`
   c. OSs: Search `$XILINX_EDK/sw/`*`Library Name`*`/bsp` (UNIX) and `%XILINX_EDK%\sw\`*`Library Name`*`\bsp`

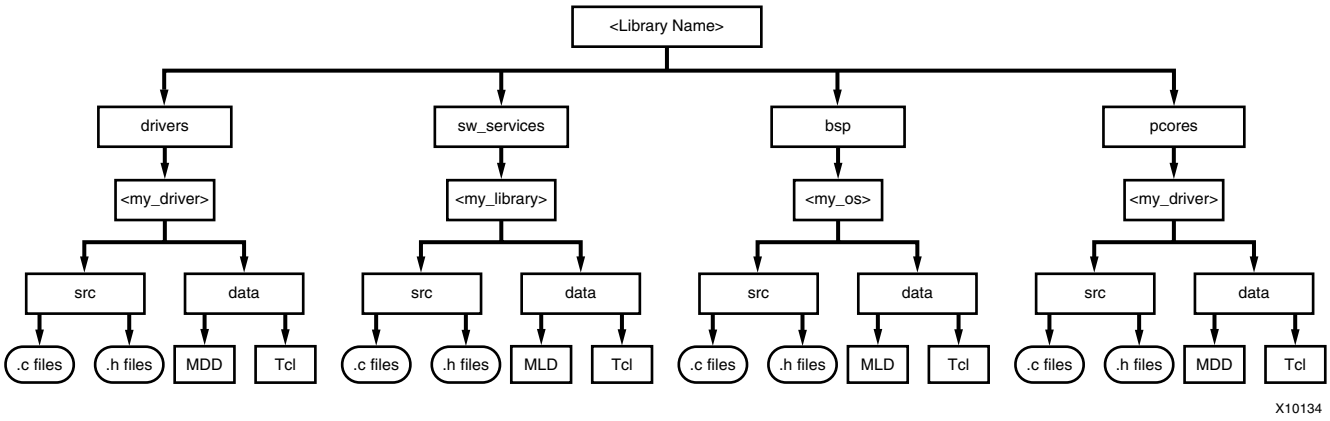

Figure 4-2: **Directory Structure of Drivers, OSs, and Libraries**

Output Files

Libgen generates directories and files in the `YOUR_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, which are described in the following subsections:

- `include` Directory
- `lib` Directory
- `libsrc` Directory
- `code` Directory

include Directory

The include directory contains C header files 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, OSs, libraries and user programs, as well as function prototypes. The Microprocessor Driver Definition (MDD) file for each driver specifies the definitions that must be customized for each peripheral that uses the driver. Refer to the "Microprocessor Driver Definition (MDD)" chapter in the *Platform Specification Format Reference Manual* for more information. The Microprocessor Library Definition (MLD) file for each OS and library specifies the definitions that you must customize. Refer to the "Microprocessor Library Definition (MLD)" chapter 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. For more information about the libraries, refer to the "Xilinx Microkernel (XMK)" section of the *OS and Libraries Reference Manual*, (available in the /doc directory of your EDK installation).

libsrc Directory

The libsrc directory contains intermediate files and MAKE files needed to compile the OSs, 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 your driver, OS, and library directories. Refer to the "Drivers", "OS Block", 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. You must put your sources, executables, and any other files in an area that you create.

Libraries and Drivers Generation

Basic Philosophy

This section describes the basic philosophy for generation of libraries and drivers.

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 does the following for each processor:

- Builds the directory structure as defined in the "Output Files" section.
- Copies the necessary source files for the drivers, OSs, and 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 or MLD file) procedure for each of the drivers, OSs, and libraries visible to the processor.

- Calls the `generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor. This generates the necessary configuration files for each of the drivers, OSs, and libraries in the `include` directory of the processor.
- Calls the `post_generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor.
- Runs `make` (with targets `include` and `libs`) for the OSs, drivers, and libraries specific to the processor. On Unix platforms (Linux and Solaris), the `gmake` utility is used, while on NT platforms, `make` is used for compilation.
- Calls the `execs_generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor.

MDD, MLD, and Tcl

A driver or library has two data files associated with it:

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

  The Tcl file includes procedures that Libgen calls at various stages of its execution. Various procedures in a Tcl file include:

  - `DRC`
    The name of DRC given in the MDD or MLD file
  - `generate`
    A Libgen-defined procedure that is called after files are copied
  - `post_generate`
    A Libgen-defined procedure that is called after `generate` has been called on all drivers, OSs, and libraries
  - `execs_generate`
    A Libgen-defined procedure that is called after the BSPs, libraries, and drivers have been generated

Note: The data generation (Tcl) file is not necessary for a driver, OS, or library.

For more information about the Tcl procedures and MDD/MLD related parameters, refer to the "Microprocessor Driver Definition (MDD)" chapter in the *Platform Specification Format Reference Manual* and the "Microprocessor Library Definition (MLD)" chapter in the *Platform Specification Format Reference Manual* (available in the `/doc` directory of your EDK installation).

MSS Parameters

For a complete description of the MSS format and all the parameters that MSS supports, refer to the "Microprocessor Software Specification (MSS)" chapter in the *Platform Specification Format Reference Manual* (available in the `/doc` directory of your EDK installation).

Drivers

Most peripherals require software drivers. The EDK peripherals are shipped with associated drivers, libraries and BSPs. Refer to the *Device Driver Programmer Guide* for more information on driver functions. This guide is located in the `/doc` folder of your EDK installation, file name: `xilinx_drivers_guide.pdf`.

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 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 the "Microprocessor Driver Definition (MDD)" and "Microprocessor Software Specification (MSS)" chapters in the *Platform Specification Format Reference Manual* (available in the `/doc` directory of your EDK installation) for more information.

You can write your own drivers. These drivers must be in a specific directory under `YOUR_PROJECT/drivers` or `library_name/drivers`, as shown in Figure 4-1 on page 55. The `DRIVER_NAME` attribute allows you to specify any name for your drivers, which is also the name of the driver directory. The source files and MAKE file for the driver must be in the `src/` subdirectory under the `driver_name` directory. The MAKE file 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 the "Microprocessor Driver Definition (MDD)" chapter 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. Each library is associated with a processor instance specified using the `PROCESSOR_INSTANCE` parameter. The library directory contains C source and header files and a MAKE file for the library.

The MLD file for each library specifies all configurable options for the libraries. Each MLD file has a corresponding Tcl file associated with it. Refer to the "Microprocessor Library Definition (MLD)" and "Microprocessor Software Specification (MSS)" chapters in the *Platform Specification Format Reference Manual* (available in the `/doc` directory of your EDK installation) for more information.

You can write your own libraries. These libraries must be in a specific directory under `YOUR_PROJECT/sw_services` or `library_name/sw_services` as shown in Figure 4-1 on page 55. The `LIBRARY_NAME` attribute allows you to specify any name for your libraries, which is also the name of the library directory. The source files and MAKE file for the library must be in the `src` subdirectory under the `library_name` directory. The MAKE file 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 for more information about the structure of the libraries. Refer to the "Microprocessor Library Definition (MLD)" chapter in the *Platform Specification Format Reference Manual* for details on how to write an MLD and its corresponding Tcl file.

OS Block

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 MAKE file 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 the "Microprocessor Library Definition (MLD)" and "Microprocessor Software Specification (MSS)" chapters in the *Platform Specification Format Reference Manual* (available in the /doc directory of your EDK installation) for more information.

You can write your own OSs. These OSs must be in a specific directory under YOUR_PROJECT/bsp or library_name/bsp as shown in Figure 4-1 on page 55. The OS_NAME attribute allows you to specify any name for your OS, which is also the name of the OS directory. The source files and MAKE file for the OS must be in the src subdirectory under the os_name directory. The MAKE file 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 OSs to get an understanding of the structure of the OSs. Refer to the "Microprocessor Library Definition (MLD)" chapter in the *Platform Specification Format Reference Manual* for details on how to write an MLD and its corresponding Tcl file.

Interrupts and Interrupt Controllers

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. Alternatively, you can dynamically register interrupt handlers in your code. Interrupts for the peripherals need to be enabled in your 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 for you to implement. If the INT_HANDLER parameter is not specified, a default dummy handler routine for the peripheral is used.

MicroBlaze

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.

PowerPC

For the PowerPC™ processor, you are responsible for setting up the exception table. Refer to Appendix B, "Interrupt Management," for more information.

XMDStub Peripherals (MicroBlaze Specific)

These peripherals are used specifically for debug with the XMDStub program. For more information about the debug program XMDStub, refer to Chapter 12, "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.

Chapter 5

Virtual Platform Generator (VPgen)

This chapter introduces the virtual platform generation utility (VPgen) in XPS. It contains the following sections:

- Overview
- Tool Usage and Options
- Output Files
- Available Models
- Current Restrictions

Overview

The virtual platform is a cycle-level simulation model of the hardware system. The virtual platform can be used to debug and profile software application code on the host machines, eliminating the need to get the actual hardware working on a prototyping board. EDK supports virtual platforms for NT and Linux only. These models are functionally correct only on clock edges and not between edges. Therefore, they provide a faster solution than would performing a complete simulation in event-driven simulators such as ModelSim and NCSim.

VPgen takes an Microprocessor Hardware Specification (MHS) file as input and generates a binary executable for the hardware system. VPgen substitutes every component (pcore) in the system and replaces it with the corresponding C-model of the component. VPgen can also generate a C-model from a synthesizable HDL code source. VPgen generates a top-level kernel, which integrates various models and provides a mechanism of communication between them. The kernel also has a static scheduling of models of various components on clock edges.

The generated model has an interface used by XMD to "control" the virtual platform while executing, debugging, or profiling the software application.

Tool Usage and Options

In XPS, you can generate a virtual platform model by selecting **Tools →Generate Virtual Platform**. Once a virtual platform model is successfully generated, you can use XMD to connect to this model and work on your software application. Refer to "Virtual Platform MicroBlaze Target" on page 170.

You can run VPgen from the command line as follows:

```
% vpgen [options] system.mhs
```

The following options are supported by VPgen:

Table 5-1: **VPgen Syntax Options**

Option	Command	Description
Help	**-h, -help**	Displays the usage menu and then exits.
Display version information	**-v**	Displays the version number of VPgen.
Log	**-log logfile[.log]**	Specifies the log file. The default is `vpgen.log`.
Architecture family	**-p part_name**	Defines the target device defined either as architecture family or partname. Use the **-h** option to view a list of values for the target family.
Specify library path for your peripherals	**-lp library_path**	Specifies a library containing repositories of user peripherals. VPgen looks for peripherals in the `<library_path>/<sub_dir>/pcores` directory.

Output Files

VPgen generates its output in the `virtualplatform` directory within the directory containing the MHS file. The main output file of interest is `virtualplatform/vpexec[.exe]`, which is the compiled binary executable of the hardware system and the kernel.

To generate `vpexec`, VPgen also produces intermediate files such as `<system>.[c|h]`. For each peripheral that does not have a predefined model available, a `<peripheral_instance>_wrapper.[c|h]` file generates. There is also a MAKE file called `vpgen.make` that compiles all the C files and produces `vpexec`.

Available Models

For any IP that requires interaction with logic outside of the FPGA, that is, which has ports going out of the MHS, a modeling of the external logic is required within the IP model itself. Additionally, hand-written C models of IPs are more optimized than auto-generated models. Thus, for MicroBlaze™, EDK provides a hand-written Instruction Set Simulator (ISS) model that is used by VPgen.

From EDK 8.1i onward, VPgen also supports generating of models for blocks which do not have any I/O ports (no ports connecting to top-level ports in the MHS). Example of these blocks are processor accelerator blocks or DMA-type blocks. The tool generates a cycle-accurate model of such IPs by reading the HDL files associated with those IPs and generating the model on the fly. The generated models are compiled into the `vpexec` executable file.

Models for the following IPs are provided:

Table 5-2: **IP Models Provided in EDK**

IP	Description/Notes
bram_block	Supports version v1.00.a
fsl_v20	Supports versions v1.00.b and v2.00.a
lmb_bram_if_cntlr	Supports versions v1.00.a and v2.00.a. However, the functionality for the lmb_bram_if_cntlr and the connected bram_block are incorporated within the MicroBlaze ISS model itself. It is therefore required that both the ILMB and DLMB of the MicroBlaze device, if used, connect to the same bram_block in the MHS file.
mch_opb_ddr	Supports versions v1.00.a and v1.00.b
mch_opb_emc	Supports version 1.00.a
mch_opb_sdram	Supports version v1.00.a
microblaze	Supports versions v3.00.a, v4.00.a, and v5.00.a. MicroBlaze is modeled using a cycle accurate Instruction Set Simulator (ISS).
opb_bram_if_cntlr	Supports version v1.00.a
opb_ddr	Supports versions v1_00_b and v1_10_a. opb_ddr models only a simple volatile storage with fixed read and write latencies of 8 and 4 OPB clock cycles respectively.
opb_emc	Supports versions v1.10.b, v2.00.a opb_emc (and other memory models) models only a simple volatile storage on the memory interface. It does not model specific memory chips that recognize command sequences or have control registers.
opb_gpio	Supports version v3.01.b
opb_intc	Supports version v1.00.c
opb_sdram	Supports versions v1.00.c, v1.00.d and v1.00.e opb_sdram models a volatile storage with fixed read and write latencies of 8 and 4 OPB clock cycles respectively.
opb_uartlite	Supports version v1.00.b
opb_timer	Supports version v1.00.b
opb_v20	Supports version v1.10.c. The model does not support dynamic priority arbitration.
util_bus_split	Supports version v1.00.a
util_flipflop	Supports version v1.00.a

Table 5-2: **IP Models Provided in EDK *(Continued)***

IP	Description/Notes
util_reduced_logic	Supports version v1.00.a
util_vector_logic	Supports version v1.00.a

VPgen generates a real model for `util_*` IPs provided with EDK. For any other IP, VPgen generates a dummy model that conforms to the interface required by the kernel but does not perform any functionality of that IP.

Current Restrictions

In the current release, VPgen supports most designs generated by the Base System Builder Wizard (BSB). This includes any design that contains a combination of peripherals in the available models list. VPgen also has the ability to create cycle-accurate models for user IP cores that do not have IO ports (a port connection to a top-level port in the MHS file).

- VPgen creates a dummy model for any core for which there is no available model and for which no cycle-accurate model could be generated. A dummy model does not respond to stimulation on its ports. If access to a core occurs through the C program for a MicroBlaze device, then the model for MicroBlaze times out.

  Note: Do not stimulate a core for which there is no model available.
- Only designs containing a single MicroBlaze device are supported.
- PowerPC™ devices and PLB buses are not supported.

Chapter 6

Platform Specification Utility (PsfUtility)

This chapter describes the various features and the usage of the Platform Specification Utility (PsfUtility) tool that enables automatic generation of Microprocessor Peripheral Definition (MPD) files. MPD files are required to create IP cores that are compliant with the Embedded Development Kit (EDK). The Create and Import Peripheral Wizard in the Xilinx® Platform Studio (XPS) GUI supports features provided by the PsfUtility for MPD file creation (recommended).

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
- Conventions for Defining HDL Peripherals

Tool Options

Table 6-1: **PsfUtility Syntax Options**

Option	Command	Description
Help	`-h, -help`	Displays the usage menu and then exits.
Display version information	`-v`	Displays the version number.

Table 6-1: **PsfUtility Syntax Options *(Continued)***

Option	Command	Description
HDL file to MPD	**-hdl2mpd <hdlfile>**	Generate MPD from VHDL/Ver src/prj file. Suboptions: **-lang <ver|vhdl|>** — Specify language **-top <design>** — Specify top-level entity or module name. **{-bus <opb|plb|dcr|lmb|fsl> <m|s|ms>}** — Specify one or more bus interfaces for the core. **{-tbus <transparent_bus_name> bram_port}** — Specify one or more transparent bus interfaces for the core. **{-p2pbus <busif_name> <bus_std> <target|initiator>}** — Specify one or more point-to-point connections for the core. **-o <outfile>** — Specify output filename; default is stdout.
PAO file to MPD	**-pao2mpd <paofile>**	Generate MPD from Peripheral Analyze Order (PAO) file. Suboptions: **-lang <ver|vhdl>** — Specify language. **-top <design>** — Specify top-level entity or module name. **{-bus <opb|plb|dcr|lmb|fsl> <m|s|ms>}** — Specify one or more bus interfaces for the core. **{-tbus <transparent_bus_name> bram_port}** — Specify one or more transparent bus interfaces for the core. **{-p2pbus <busif_name> <bus_std> <target|initiator>}** — Specify one or more point-to-point connections of the core. **-o <outfile>** — Specify output filename; default is stdout.
Single core MHS template	**-deploy_core <corename> <coreversion>**	Generate MHS Template that instantiates a single core. Suboptions: **{-lp <library path>}**— Add one or more additional IP library search paths. **-o <outfile>**— Specify output filename; default is stdout.

Overview of the MPD Creation Process

You can use the PsfUtility to create MPD specifications from the VHDL specification of the core automatically. The steps involved in creating a core and delivering it through EDK are:

1. 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 "Conventions for Defining HDL Peripherals" on page 72.

   Note: Following these naming conventions enables the PsfUtility to create a correct and complete MPD file.

2. Create an XST (Xilinx Synthesis Technology) project file or a PAO file that lists all the HDL sources required to implement the IP. Invoke the PsfUtility by providing the XST project file or the PAO file with additional options. For more information on invoking

the PsfUtility with different options, see the following section, "Detailed Use Models for Automatic MPD Creation."

Detailed Use Models for Automatic MPD Creation

You can run the PsfUtility in a variety of ways, depending on the bus standard and bus interface types used with the peripheral and the number of bus interfaces a peripheral contains. Bus standards and types can be one of the following:

- OPB (on-chip peripheral bus) SLAVE
- OPB MASTER
- OPB MASTER_SLAVE
- PLB (processor local bus) SLAVE
- PLB MASTER
- PLB MASTER_SLAVE
- DCR (design control register) SLAVE
- LMB (local memory bus) SLAVE
- FSL (fast simplex link) SLAVE
- FSL MASTER
- TRANSPARENT BUS (special case)
- POINT TO POINT BUS (special case)

Peripherals with a Single Bus Interface

Most processor peripherals have a single bus interface. This is the simplest model for the PsfUtility. For most such peripherals, complete MPD specifications can be obtained without any additional attributes added to the source code.

Signal Naming Conventions

The signal names must follow the conventions specified in "Conventions for Defining HDL Peripherals" on page 72. When there is only one bus interface, no bus identifier has to be specified for the bus signals.

Invoking the 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 such as UART, the command is:

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

Peripherals with Multiple Bus Interfaces

Some peripherals might have multiple bus interfaces associated with them. These interfaces can be exclusive bus interfaces, non-exclusive bus interfaces, or a combination of both. All bus interfaces on the peripheral that can be connected to the peripheral

simultaneously are exclusive interfaces. For example, an OPB Slave bus interface and a DCR Slave bus interface are exclusive because they can be connected simultaneously.

Note: On a peripheral containing exclusive bus interfaces: a port can be connected to only one of the exclusive bus interfaces.

Non-exclusive bus interfaces cannot be connected simultaneously.

Note: Peripherals with non-exclusive bus interfaces have ports that can be connected to more than one of the non-exclusive interfaces. Further, non-exclusive interfaces have the same bus interface standard. For example, an OPB Slave interface and an OPB Master/Slave interface are non-exclusive if they are connected to the same slave ports on the peripheral.

Non-Exclusive Bus Interfaces

Signal Naming Conventions

Signal names must adhere to the conventions specified in "Conventions for Defining HDL Peripherals" on page 72. For non-exclusive bus interfaces, bus identifiers need not be specified.

Invoking the PsfUtility With Buses Specified in the Command Line

You can specify buses on the command line when the bus signals do not have bus identifier prefixes. The command line for invoking the 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 a peripheral with a PLB slave interface and a PLB Master/Slave interface such as gemac, the command is:

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

Exclusive Bus Interfaces

Signal Naming Conventions

Signal names must adhere to the conventions specified in "Conventions for Defining HDL Peripherals" on page 72. Bus identifiers must be specified only when the peripheral has more than one bus interface of the same bus standard and type.

Invoking the PsfUtility With Buses Specified in the Command Line

You can specify buses on the command line when the bus signals are not prefixed with bus identifiers. The command line for invoking the 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 a peripheral with a PLB slave interface and a DCR Slave interface, the command is:

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

Peripherals with TRANSPARENT Bus Interfaces

Some peripherals such as 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
```

The BRAM ports must adhere to the signal naming conventions specified in "Conventions for Defining HDL Peripherals" on page 72.

Peripherals with Point-to-Point Connections

Some peripherals, such as multi-channel memory controllers, might have point-to-point connections (**BUS_STD = XIL_MEMORY_CHANNEL, BUS_TYPE = TARGET**).

Signal Naming Conventions

The signal names must follow conventions such that all signals belonging to the point-to-point connection start with the same bus interface name prefix, such as `MCH0_*`.

Invoking the PsfUtility with Point-to-Point Connections Specified in the Command Line

You can specify point-to-point connections in the command line using the bus interface name as a prefix to the bus signals. The command line for invoking PsfUtil is as follows:

```
psfutil -hdl2mpd <hdlfile> -lang <vhdl|ver> -top <top_entity>
{-p2pbus <busif_name> <bus_std> <target|initiator>} -o <mpdfile>
```

For example, to create an MPD specification for a peripheral with an `MCH0` connection, the command is:

```
psfutil -hdl2mpd mch_mem.prj -lang vhdl -top mch_mem -p2pbus MCH0
XIL_MEMORY_CHANNEL TARGET -o mch_mem.mpd
```

DRC Checks in PsfUtility

To enable generation of correct and complete MPD files from HDL sources, the PsfUtility reports the DRC errors listed below. The DRC checks are listed in the order in which they are performed.

HDL Source Errors

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

Bus Interface Checks

Depending on what bus interface is associated with which cores, the PsfUtility does the following for every bus interface specified:

- Checks and reports any missing bus signals.
- Checks and reports any repeated bus signals.

The PsfUtility does not generate an MPD file unless all bus interface checks are completed.

Conventions for Defining HDL Peripherals

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

- Lists ports and default connectivity for bus interfaces.
- Lists parameters (generics) and default values.
- Parameters defined in the MHS overwrite corresponding HDL source parameters.

Individual peripheral documentation contains information on all source file options.

Naming Conventions for Bus Interfaces

A bus interface is a grouping of related interface signals. For the automation tools to function properly, it is important to adhere to the signal naming conventions and parameters associated with a bus interface. When the signal naming conventions are correctly implemented, the following interface types are automatically recognized, and the MPD file contains the bus interface label shown in Table 6-2.

Table 6-2: **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
Slave FSL interface	SFSL
Master FSL interface	MFSL

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

Naming Conventions for VHDL Generics

For cores that contain more than one of the same bus interface, a *bus identifier* must be used. The bus identifier must be attached to all associated signals and generics.

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, the generics associated with that port can optionally use <*BI*>. If no <*BI*> string is used in the name, 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 is typically not included.
- All generics that specify a base address must end with _BASEADDR, and all generics that specify a high address must end with _HIGHADDR. Further, to tie these addresses with buses, they must also follow the conventions for parameters, as listed above. For peripherals with more than one bus interface type, the parameters must have the bus standard type specified in the name. For example, parameters for an address on the PLB bus must be specified as C_PLB_BASEADDR and C_PLB_HIGHADDR.

The Platform Generator (Platgen) automatically expands and populates certain reserved generics. For correct operation, a bus tag must be associated with these parameters. To have the PsfUtility automatically infer this information, all conventions specified above must be followed for 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.

Table 6-3: **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

Table 6-3: **Automatically Expanded Reserved Generics *(Continued)***

Parameter	Description
C_<*BI*>DCR_DWIDTH	DCR data width
C_<*BI*>DCR_NUM_SLAVES	Number of DCR slaves
C_<*BI*>FSL_DWIDTH	FSL data width
C_<*BI*>LMB_AWIDTH	LMB address width
C_<*BI*>LMB_DWIDTH	LMB data width
C_<*BI*>LMB_NUM_SLAVES	Number of LMB slaves
C_<*BI*>OPB_AWIDTH	OPB address width
C_<*BI*>OPB_DWIDTH	OPB data width
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

Platgen automatically populates all parameters shown in Table 6-4.

Table 6-4: **Reserved Parameters**

Parameter	Description
C_BUS_CONFIG	Defines the bus configuration of the MicroBlaze processor.
C_FAMILY	Defines the FPGA device family.
C_INSTANCE	Defines the instance name of the component.
C_DCR_AWIDTH	Defines the DCR address width.
C_DCR_DWIDTH	Defines the DCR data width.
C_DCR_NUM_SLAVES	Defines the number of DCR slaves on the bus.
C_LMB_AWIDTH	Defines the LMB address width.
C_LMB_DWIDTH	Defines the LMB data width.
C_LMB_NUM_SLAVES	Defines the number of LMB slaves on the bus.
C_OPB_AWIDTH	Defines the OPB address width.
C_OPB_DWIDTH	Defines the OPB data width.
C_OPB_NUM_MASTERS	Defines the number of OPB masters on the bus.
C_OPB_NUM_SLAVES	Defines the number of OPB slaves on the bus.

Table 6-4: **Reserved Parameters *(Continued)***

Parameter	Description
C_PLB_AWIDTH	Defines the PLB address width.
C_PLB_DWIDTH	Defines the PLB data width
C_PLB_MID_WIDTH	Defines the PLB master ID width. This is set to log2(S).
C_PLB_NUM_MASTERS	Defines the number of PLB masters on the bus.
C_PLB_NUM_SLAVES	Defines the number of PLB slaves on the bus.

Naming Conventions for Bus Interface Signals

This section provides naming conventions for bus interface signal names. The conventions are flexible to accommodate embedded processor systems that have more than one bus interface and more than one bus interface port per component. When cores with more than one bus interface port are included in a design, it is important to understand how to use a bus identifier. (As explained previously, a bus identifier must be used for cores that contain more than one of the same bus interface. The bus identifier must be attached to all associated signals and generics.)

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 bus interface type.
- If more than one instance of a particular bus interface type is used on a core, the bus identifier <*BI*> must be included 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. <*BI*> must be included in the port 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 slave FSL port.
  - The core has more than one master FSL 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 requires a <*BI*> string to differentiate the ports because 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 is typically not 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 the naming conventions shown in the table below:

Table 6-5: **Slave DCR Port Naming Conventions**

<*Sln*>	A meaningful name or acronym for the slave output. <*Sln*> must *not* contain the string DCR (upper, lower, or mixed case), so that slave outputs are not confused with bus outputs.
<*nDCR*>	A meaningful name or acronym for the slave input. The last three characters of <*nDCR*> must contain the string DCR (upper, lower, or mixed case).
<*BI*>	A bus identifier. 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, lower, or mixed case). For peripherals with multiple slave DCR ports, the <*BI*> strings must be unique for each bus interface.

Note: If <*BI*> is present, <*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 FSL Ports

Slave FSL ports must follow the naming conventions shown in the table below:

Table 6-6: **Slave FSL Port Naming Conventions**

<*nFSL*> or <*nFSL_S*>	A meaningful name or acronym for the slave I/O. The last five characters of <*nFSL_S*> must contain the string `FSL_S` (upper, lower, or mixed case).
<*BI*>	A bus identifier. Optional for peripherals with a single slave FSL port and required for peripherals with multiple slave FSL ports. <*BI*> must *not* contain the string `FSL_S` (upper, lower, or mixed case). For peripherals with multiple slave FSL ports, the <*BI*> strings must be unique for each bus interface.

FSL Slave Outputs

For interconnection to the FSL, all slaves must provide the following outputs:

```
<BI><nFSL_S>_Data: out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
<BI><nFSL_S>_Control: out std_logic;
<BI><nFSL_S>_Exists: out std_logic;
```

Examples:

```
FSL_S_Control:out std_logic;
Memcon_FSL_S_Control : out std_logic;
Bus1_timer_FSL_S_Control : out std_logic;
Bus1_timer_FSL_S_Data : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
Bus2_timer_FSL_S_Control : out std_logic;
Bus2_timer_FSL_S_Data : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
```

FSL Slave Inputs

For interconnection to the FSL, all slaves must provide the following inputs:

```
<BI><nFSL>_Clk : in  std_logic;
<BI><nFSL>_Rst : in  std_logic;
<BI><nFSL_S>_Clk : in  std_logic;
<BI><nFSL_S>_Read : in  std_logic;
```

Examples:

```
FSL_S_Read : in  std_logic;
Bus1_FSL_S_Read : in  std_logic;
```

Master FSL Ports

Master FSL ports must follow the naming conventions shown in the table below:

Table 6-7: **Master FSL Port Naming Conventions**

<*nFSL*> or <*nFSL_M*>	A meaningful name or acronym for the master I/O. The last five characters of <*nFSL_M*> must contain the string `FSL_M` (upper, lower, or mixed case).
<*BI*>	A bus identifier. Optional for peripherals with a single master FSL port, and required for peripherals with multiple master FSL ports. <*BI*> must *not* contain the string `FSL_M` (upper, lower, or mixed case). For peripherals with multiple master FSL ports, the <*BI*> strings must be unique for each bus interface.

FSL Master Outputs

For interconnection to the FSL, all masters must provide the following outputs:

```
<BI><nFSL_M>_Full: out std_logic;
```

Examples:

```
FSL_M_Full:out std_logic;
Memcon_FSL_M_Full : out std_logic;
```

FSL Master Inputs

For interconnection to the FSL, all masters must provide the following inputs:

```
<BI><nFSL>_Clk : in  std_logic;
<BI><nFSL>_Rst : in  std_logic;
<BI><nFSL_M>_Clk : in  std_logic;
<BI><nFSL_M>_Data : in  std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
<BI><nFSL_M>_Control : in  std_logic;
<BI><nFSL_M>_Write : in  std_logic;
```

Examples:

```
FSL_M_Write : in  std_logic;
Bus1_FSL_M_Write : in  std_logic;
Bus1_timer_FSL_M_Control : out std_logic;
Bus1_timer_FSL_M_Data : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
Bus2_timer_FSL_M_Control : out std_logic;
Bus2_timer_FSL_M_Data : out std_logic_vector(0 to C_<BI>FSL_DWIDTH-1);
```

Slave LMB Ports

Slave LMB ports must follow the naming conventions shown in the table below:

Table 6-8: **Slave LMB Port Naming Conventions**

<*Sln*>	A meaningful name or acronym for the slave output. <*Sln*> must *not* contain the string `LMB` (upper, lower, or mixed case), so that slave outputs will not be confused with bus outputs.
<*nLMB*>	A meaningful name or acronym for the slave input. The last three characters of <*nLMB*> must contain the string `LMB` (upper, lower, or mixed case).
<*BI*>	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, lower, or mixed case). For peripherals with multiple slave LMB ports, the <*BI*> strings must be unique for each bus interface.

Note: If <*BI*> is present, <*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.

Master OPB ports must follow the naming conventions shown in the table below:

Table 6-9: **Master OPB Port Naming Conventions**

<*Mn*>	A meaningful name or acronym for the master output. <*Mn*> must *not* contain the string `OPB` (upper, lower, or mixed case), so that master outputs will not be confused with bus outputs.
<*nOPB*>	A meaningful name or acronym for the master input. The last three characters of <*nOPB*> must contain the string `OPB` (upper, lower, or mixed case).
<*BI*>	A bus identifier. 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, lower, or mixed case). For peripherals with multiple OPB ports, the <*BI*> strings must be unique for each bus interface.

Note: If <*BI*> is present, <*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 slave OPB ports that are independent of master OPB ports. For the signal list for cores that use a combined master/slave bus interface, refer to "Master/Slave OPB Ports" on page 82.

Slave OPB ports must follow the naming conventions shown in the table below:

Table 6-10: **Slave OPB Port Naming Conventions**

<*Sln*>	A meaningful name or acronym for the slave output. <*Sln*> must *not* contain the string `OPB` (upper, lower, or mixed case), so that slave outputs are not confused with bus outputs.
<*nOPB*>	A meaningful name or acronym for the slave input. The last three characters of <*nOPB*> must contain the string `OPB` (upper, lower, or mixed case).
<*BI*>	A Bus Identifier. 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, lower, 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.

Note: If <*BI*> is present, <*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 bus interface type is typically used when a peripheral has both master and slave functionality and when DMA is included with the peripheral. It is useful for the master and slave to share the input and output data buses.

Master/slave OPB ports must follow the naming conventions shown in the table below:

Table 6-11: **Master/Slave OPB Port Naming Conventions**

<Mn>	A meaningful name or acronym for the master output. *<Mn>* must *not* contain the string `OPB` (upper, lower, or mixed case), so that master outputs are not confused with bus outputs.
<Sln>	A meaningful name or acronym for the slave output. To avoid confusion between slave and bus outputs, *<Sln>* must *not* contain the string `OPB` (upper, lower, or mixed case).
<nOPB>	A meaningful name or acronym for the slave input. The last three characters of *<nOPB>* must contain the string `OPB` (upper, lower, or mixed case).
<BI>	A bus identifier. 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, lower, 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.

Note: If *<BI>* is present, *<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 the naming conventions shown in the table below:

Table 6-12: **Master PLB Port Naming Conventions**

<*Mn*>	A meaningful name or acronym for the master output. <*Mn*> must *not* contain the string `PLB` (upper, lower, or mixed case), so that master outputs are not confused with bus outputs.
<*nPLB*>	A meaningful name or acronym for the master input. The last three characters of <*nPLB*> must contain the string `PLB` (upper, lower, or mixed case).
<*BI*>	A bus identifier. 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, lower, or mixed case). For peripherals with multiple master PLB ports, the <*BI*> strings must be unique for each bus interface.

Note: If <*BI*> is present, <*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 the naming conventions shown in the table below:

Table 6-13: **Slave PLB Port Naming Conventions**

<*Sln*>	A meaningful name or acronym for the slave output. <*Sln*> must *not* contain the string PLB (upper, lower, or mixed case), so that slave outputs are not confused with bus outputs.
<*nPLB*>	A meaningful name or acronym for the slave input. The last three characters of <*nPLB*> must contain the string PLB (upper, lower, or mixed case).
<*BI*>	A bus identifier. 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, lower, or mixed case). For peripherals with multiple PLB ports, the <*BI*> strings must be unique for each bus interface.

Note: If <*BI*> is present, <*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 7

Version Management Tools

This chapter introduces the version management tools in XPS. It contains the following sections:

- Overview
- Format Revision Tool Backup and Update Processes
- Command Line Option for the Format Revision Tool
- The Version Management Wizard

Overview

When you open an older project with the current version of EDK, the Format Revision Tool automatically performs format changes to an existing EDK project and makes it compatible with the current version. Backups of existing files, such as XMP (Xilinx® Microprocessor Project), MHS (Microprocessor Hardware Specification), and MSS (Microprocessor Software Specification), are performed before the format changes are applied. Updates to IP and drivers, if any, are handled by the Version Management Wizard, which launches after the Format Revision Tool runs. The format revision tool does not modify the IPs used in the MHS design. It only updates the syntax, so the project can be opened with the new tools.

Note: For projects created in EDK 3.2 or earlier releases, automated updates are not possible; for these, you must use the `revup32to61` batch utility provided in EDK 6.1, 6.2, and 6.3.

Format Revision Tool Backup and Update Processes

The Format Revision Tool creates a backup of your files and a file name extension that specifies the EDK release number. For example, EDK 8.1i files are saved with a `.81` extension and then modified for EDK 8.2i tools.

Changes in 8.2i

- For submodule designs, the Format Revision Tool expands any I/O ports into individual _I, _O, and _T ports. This aligns with changes to Platgen; any buffers in the generated stub HDL are not instantiated, and the interface of the generated HDL stays the same as that in the MHS file.
- The Format Revision Tool changes the value of SIGIS for top-level ports from DCMCLK to CLK. The value DCMCLK has been deprecated.
- The preprocessor, assembler, and linker specific options for a software application are moved and included among the Advanced Compiler Options settings; individual options have been eliminated.
- The synthesis tool setting is removed.

Changes in 8.1i

- The PROCINST PARAMETER is added to LIBRARY blocks, which ensures that a given library can be configured differently across different processor instances in the system.
- MicroBlaze™-based application linker script updates are provided to allow the addition of new vector sections that support CRT changes.
- MicroBlaze-based application linker script updates are provided to allow the addition of new sections that support C++.
- PowerPC™-based application linker script updates are provided to allow the addition of new sections that support C++.
- For MicroBlaze applications, the program start address is changed from 0x0 to 0x50 to accommodate the change in size of `xmdstub.elf`.
- For projects that use the Spartan™-3 FPGA architecture, there is a change to `bitgen.ut`.

Changes in 7.1i

PowerPC-based application linker script updates are provided to allow for the addition of new sections that support GCC 3.4.1 changes.

Changes in 6.3i

The EDGE and LEVEL subproperties on top-level interrupt ports are consolidated into the SENSITIVITY subproperty in the MHS file.

Changes in 6.2i

- The `mb-gcc` compiler option related to the hard multiplier is removed. This is based only on FPGA architecture.
- In the MSS file, the PROCESSOR block is split into two blocks, PROCESSOR and OS. In conjunction with this change:
  - The Linux and VxWorks LIBRARY blocks are renamed to reflect their new status as OS blocks.
  - With the introduction of the OS block, all peripherals used with Linux and VxWorks operating systems are specified using a CONNECTED_PERIPHS parameter, which replaces the CONNECT_TO parameter used in earlier versions. When the Format Revision Tool runs, it collects old CONNECT_TO driver parameter peripherals and collates them in the CONNECTED_PERIPHS parameter of the OS block.
  - In the MSS file PROCESSOR block, the following parameters are removed: LEVEL, EXECUTABLE, SHIFTER, and DEFAULT_INIT.
  - In the PROCESSOR block, the DEBUG_PERIPHERAL is renamed XMDSTUB_PERIPHERAL.

Command Line Option for the Format Revision Tool

Run the Format Revision tool as follows from the command line:

```
revup system.xmp
```

The following option is supported:

-h (Help) – Displays the usage menu and then quits.

The Version Management Wizard

When an older project is opened for the first time with the new version of EDK, the Format Revision Tool runs, and the Version Management Wizard opens. Some IP cores may have been obsoleted or updated in the repository since the project was last processed, so the wizard outlines the modifications, provides the option to automatically upgrade to the latest backward-compatible revision or provides more information on how to upgrade to the latest version of the core. The wizard also gives you the option to make similar updates for drivers, if required. Backup copies of the MHS and MSS files are created before the project is modified. You may choose to cancel the wizard at any time without modifying the files, but, as a result, it may not be possible to run the project with the current version of XPS.

XILINX®

Chapter 8

Bitstream Initializer (BitInit)

This chapter describes the Bitstream Initializer (BitInit) utility. The chapter contains the following sections.

- Overview
- Tool Usage
- Tool Options

Overview

BitInit initializes the instruction memory of processors on the FPGA, which is 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

To invoke the BitInit tool, type the following:

```
% bitinit <mhsfile> [options]
```

Note: You must specify *<mhsfile>* before specifying other tool options.

Tool Options

The following options are supported in the current version of BitInit.

Table 8-1: **BitInit Syntax Options**

Option	Command	Description
Display Help	**-h**	Displays the usage menu and then quits.
Display version	**-v**	Displays the version and then quits.
Input BMM file	**-bm**	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`
Bitstream file	**-bt**	Specifies the input bitstream file that does not have its memory initialized. Default: `implementation/<sysname>.bit`

Table 8-1: **BitInit Syntax Options *(Continued)***

Option	Command	Description
Output bitstream file	**-o**	Specifies the name of the output file to generate the bitstream with initialized memory. Default: `implementation/download.bit`
Specify the Processor Instance name and list of ELF files	**-pe**	Specifies the name of the processor instance in the MHS and its associate list of ELF files that form its instruction memory. This option can be repeated several times based on the number of processor instances in the design.
Libraries path	**-lp**	Specifies the path to repository libraries. This option can be repeated to specify multiple libraries.
Log file name	**-log**	Specifies the name of the log file to capture the log. Default: `bitinit.log`
Quiet mode	**-quiet**	Runs the tool in quiet mode. In this mode, it does not print status, warning, or informational messages while running. It prints only error messages on the console.

Note: BitInit also produces a file named `data2mem.dmr`, which is the log file generated during invocation of the Data2MEM utility.

Chapter 9

Flash Memory Programming

This chapter describes the flash memory programming tools in EDK and includes following sections:

- Overview
- Supported Flash Hardware
- Customizing Flash Programming

Overview

Typically, you can program the following in flash:

- Executable or bootable images of applications
- Hardware bitstreams for your FPGA
- File system images, data files such as sample data and algorithmic tables

The first use case is most common. When the processor in your design comes out of reset, it starts executing code stored in BRAM at the processor reset location. Typically, BRAM is too small (by a few kilobytes or so) to accommodate your entire software application image. You can, therefore, store your software application image (typically, a few megabytes-worth of data) in flash memory. 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.

The software application you build from your project is in Executable Linked Format (ELF). When bootloading a software application from flash, ELF images should be converted to one of the common bootloadable image formats, such as SREC. This keeps the bootloader simpler and smaller. EDK provides GUI and command line options for creating bootloaders in SREC format. See the online help for instructions on creating a flash bootloader and on converting ELF images to SREC.

Flash Programming from XPS

The Xilinx® Platform Studio (XPS) GUI includes a Program Flash Memory dialog box, which allows you to program external Common Flash Interface (CFI) compliant parallel flash devices on your board, connected through the `opb_emc/plb_emc` external memory controller IP cores. The programming solution is designed to be generic and targets a wide variety of flash hardware and layouts.

The programming is achieved through the debugger connection to a processor in your design. XPS downloads and executes a small in-system flash programming stub on the target processor. A host Tcl script drives the in-system flash programming stub with

commands and data and completes the flash programming. The flash programming tools do not process or interpret the image file to be programmed and routinely program the file as-is onto flash memory. Your software and hardware application setup must infer the contents of the file being programmed, as desired.

Supported Flash Hardware

The flash programmer uses the Common Flash Interface (CFI) to query the flash devices, so it requires that the flash device be CFI compliant. The layout of the flash devices to form the total memory interface width is also important. Table 9-1 lists the flash layouts/configurations that are supported. If your flash layout does not match a configuration in the table, then you will have to customize the flash programming session. Refer to "Customizing Flash Programming" on page 95.

Table 9-1: **Supported Flash Configurations**

x8 only capable device forming an 8-bit data bus
x16/x8 capable device in x8 mode forming an 8-bit data bus
x32/x8 capable device in x8 mode forming an 8-bit data bus
x16/x8 capable device in x16 mode forming a 16-bit data bus
Paired x8 only capable devices forming a 16-bit data bus
Quad x8 only capable devices forming a 32-bit data bus
Paired x16 only capable devices in x16 mode, forming a 32-bit data bus
x32 /x8 capable device in x32 mode, forming a 32-bit data bus
x32 only capable device forming a 32-bit data bus

The above physical layout, geometry information, and other logical information, such as command sets understood, are determined using the CFI. Currently, the flash programmer can handle flash devices that can understand only the CFI-defined command sets listed in Table 9-2.

Table 9-2: **CFI Defined Command Sets**

CFI Vendor ID	OEM Sponsor	Interface Name
1	Intel/Sharp	Intel/Sharp Extended Command Set
2	AMD/Fujitsu	AMD/Fujitsu Standard Command Set
3	Intel	Intel Standard Command Set
4	AMD/Fujitsu	AMD/Fujitsu Extended Command Set

By default, the flash programmer supports only flash devices that have a sector map which is exactly as stored in the CFI table. Some flash vendors store one sector map in the CFI table and another (based on the boot topology of the flash device) in hardware. Refer to "Customizing Flash Programming" on page 95 for more information about how to work around this.

The flash hardware is assumed to be in a reset state when programming is attempted by the flash programming stub. All the flash sectors are assumed to be in an unprotected state. The flash programming stub will not attempt to unlock or initialize the flash in any other way and will report an error if the flash hardware is not in a ready and unlocked state.

Customizing Flash Programming

The flash programming setup described above might not fit your requirements exactly. There could be, for example, hardware incompatibilities, flash command set incompatibilities, or memory size constraints. This section briefly describes the internal workings of the flash programming algorithm. Knowing this, you can plug in and replace elements of the flow to customize it for your particular setup.

When you click on the Program Flash button in XCS, the following sequence of events occurs:

1. A `flash_params.tcl` file is written out to the `etc/` folder. This contains parameters that describe the flash programming session and is used by the flash programmer Tcl file.
2. XPS launches XMD with the flash programmer Tcl script, executing it with a command such as `xmd -tcl flashwriter.tcl`. This flash programmer host Tcl comes from the installation. If you wish your own driver Tcl to run when you click the Program Flash button, place a copy of the `flashwriter.tcl` file in your project root directory. XMD searches for the specified file in your project directory before looking for it in the repository.
3. The flash programmer Tcl script copies the flash programmer application source files from the installation to the `etc/flashwriter` folder. It compiles the application locally to execute from the scratch memory address you specified in the dialog box. Here again, if you wish to compile your own flash writer sources, you can modify your local copy of the `flashwriter.tcl` script to compile your own sources instead of those from the installation.
4. The script downloads the flash programmer to the processor and communicates with the flash programmer through mailboxes in memory. In other words, it writes parameters to the memory locations corresponding to variables in the flash programmer address space and lets the flash programmer execute.
5. The script waits for the flash programmer to invoke a callback function at the end of each operation and stops the application at the callback function by setting a breakpoint at beginning of the function. Once the flash programmer stops, the host Tcl processes the results and continues with more commands as required.
6. While running, the flash programmer erases only as many flash blocks as required in which to store the image.
7. The flashwriter allocates a streaming buffer (based on the amount of scratch pad memory available) and iteratively stream programs the image file. The stream buffer is allocated within the flashwriter. If there is enough scratch memory to hold the entire image, the programming can be completed very quickly, in one shot.
8. Once the programming is done, the flash programmer Tcl sends an exit command to the flash programmer and terminates the XMD session.

Here is an example set of steps to perform for a custom flow:

1. Copy `flashwriter.tcl` from `<edk_install>/data/xmd/flashwriter.tcl` to your EDK project folder.

2. Create a directory within your EDK project, called *sw_services* (if it does not exist already).
3. Copy the entire `<edk_install>/data/xmd/flashwriter` directory to the `sw_services` directory.
4. Edit the following line in the `flashwriter.tcl` that you copied, and change:

   ```
   set flashwriter_src [file join $xilinx_edk "data" "xmd" "flashwriter" "src"]
   ```

   to

   ```
   set flashwriter_src [file join "." "sw_services" "flashwriter" "src"]
   ```

   Now every time you use the Program Flash Memory dialog box, the flash programming tools will use the script and the sources you copied to the `sw_services` directory. You are free to customize these as you wish.

   If you prefer to not have the GUI overwrite the `etc/flash_params.tcl` file, you must run the command ***`xmd -tcl flashwriter.tcl`*** on the command line to use only the values that you specify in the `etc/flash_params.tcl` file.

The various parameters in the `etc/flash_params.tcl` file are listed in Table 9-3.

Table 9-3: **Flash Programming Parameters**

Variable	Function
FLASH_FILE	This a string containing the full path of the file to be programmed.
FLASH_BASEADDR	The base address of the flash memory bank.
FLASH_PROG_OFFSET	The offset within the flash memory bank at which the programming should be done.
SCRATCH_BASEADDR	The base address of the scratch memory used during programming.
SCRATCH_LEN	The length of the scratch memory in bytes.
XMD_CONNECT	The connect command used in XMD to connect to the processor.
PROC_INSTANCE	The instance name of the processor used for programming.
TARGET_TYPE	The type of the processor instance used for programming: MicroBlaze™ or PowerPC™ 405.
FLASH_BOOT_CONFIG	Refer to "Handling Flash Devices with Conflicting Sector Layouts" on page 97.
EXTRA_COMPILER_FLAGS	For MicroBlaze, specify any compiler flags required to turn on support for hardware features. For example, if you have the hardware multiplier enabled, then add -mno-xl-soft-mul here. Do not set this variable for PowerPC.

Manual Conversion of ELF Files to SREC for Bootloader Applications

If you want to create some SREC images of your ELF file manually instead of using the auto-convert feature in XPS, you can use the command line tools. For example, to create a final software application image named `myexecutable.elf`, navigate in the console of your operating system (Cygwin on Windows platforms) to the folder containing this ELF file and type the following:

```
<platform>-objcopy -O srec myexecutable.elf myexecutable.srec
```

where platform is powerpc-eabi if your processor is PPC405, or mb if your processor is MicroBlaze.

This creates an SREC file that you can then use as appropriate. The utilities `mb-objcopy` and `powerpc-eabi-objcopy` are GNU binary utilities that ship with EDK.

For information about creating a bootloader from within the XPS GUI, see the XPS online help.

Operational Characteristics and Workarounds

Handling Flash Devices with Conflicting Sector Layouts

As mentioned earlier, some flash vendors store a different sector map in the CFI table and another (based on the boot topology of the flash device) in hardware. Because the boot topology information is not standardized in CFI, the flash programmer has no way of determining what kind of layout your particular flash device has.

If your flash hardware has a sector layout that is different from the one specified in the CFI table for the device, then you must create a custom flash programming flow. You must determine whether yours is a top-boot flash or a bottom-boot flash device. In a top-boot flash device, the smallest sectors are the last sectors in the flash. In a bottom-boot flash device, the smallest sectors are the first sectors in the flash layout.

Once you determine the flash device type, you must copy over the files as described earlier, to create a custom programming flow.

- If you have a bottom-boot flash, add the following line in your `etc/flash_params.tcl` file:

  ```
  set FLASH_BOOT_CONFIG BOTTOM_BOOT_FLASH
  ```

- If you have a top-boot flash, add the following line in your `etc/flash_params.tcl` file:

  ```
  set FLASH_BOOT_CONFIG TOP_BOOT_FLASH
  ```

Next, run the flash programming from the command line with the following command:

```
xmd -tcl flashwriter.tcl
```

Internally, these variables cause the flash programmer to rearrange the sector map according to the boot topology.

Data Polling Algorithm for AMD/Fujitsu Command Set

The DQ7 data polling algorithm is used during erasure and programming operations on flash hardware that supports the AMD/Fujitsu command set. Certain flash devices are known to use a configuration register to control the behavior of the data polling DQ7 bit. It is required that DQ7 outputs `0` during an erase operation and `1` at the end of the operation. Similarly, DQ7 must output inverted data during programming and the actual data after

programming is done. If your flash hardware has a different configuration when using the Program Flash Memory dialog box, then the programming could fail in obscure ways. Refer to your flash hardware datasheet for information about how to reset the configuration so that DQ7 behaves in this manner. Some known flash devices that offer this configuration feature are AT49BV322A(T), AT49BV162A(T) and AT49BV163A(T).

XILINX®

Chapter 10

GNU Compiler Tools

EDK includes the GNU compiler collection (GCC) for both PowerPC™ and MicroBlaze™ processors. The EDK GNU tools support both the C and C++ languages. The MicroBlaze GNU tools include *mb-gcc* and *mb-g++* compilers, *mb-as* assembler and *mb-ld* linker. The PowerPC tools include *powerpc-eabi-gcc* and *powerpc-eabi-g++* compilers, *powerpc-eabi-as* assembler and the *powerpc-eabi-ld* linker. The toolchains also include the C, Math, GCC, and C++ standard libraries. The GCC tools are built out of the open source GCC 3.4 version sources. For a complete reference of all the features of this release of the GCC tools, refer to http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/.

The compiler also uses the common binary utilities (referred to as binutils), such as an assembler, a linker, and object dump. The MicroBlaze compiler tools use the GNU binutils based on GNU version 2.10.1. The PowerPC compiler tools use the GNU binutils based on GNU version 2.15 of the sources.

The concepts, options, usage, and exceptions to language and library support are described in further sections. The rest of this chapter is organized as follows:

- Compiler Framework
- Common Compiler Usage and Options
- MicroBlaze Compiler Usage and Options
- PowerPC Compiler Usage and Options
- Other Notes

Compiler Framework

This section discusses the common features of both the MicroBlaze and PowerPC compilers. Figure 10-1 displays the GNU tool flow.

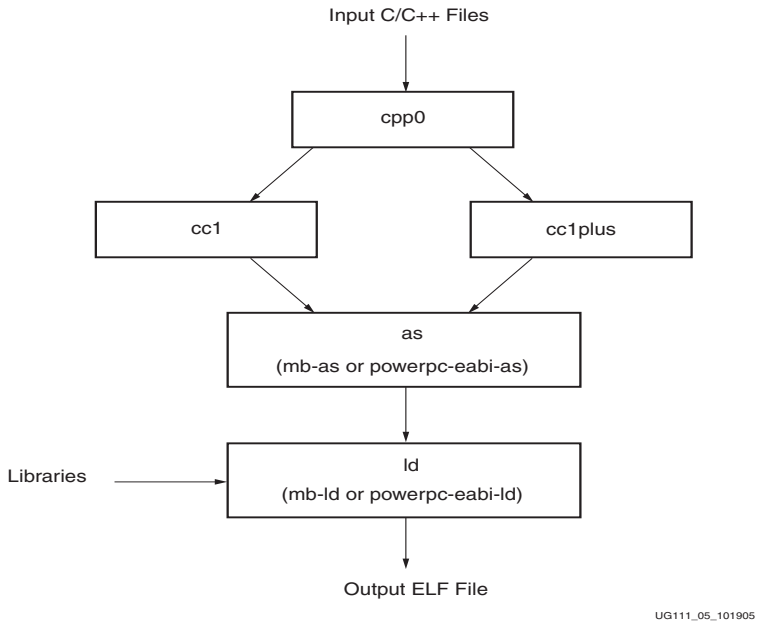

Figure 10-1: **GNU Tool Flow**

The GNU compiler is named *mb-gcc* for MicroBlaze and *powerpc-eabi-gcc* for PowerPC. The GNU compiler is a wrapper that calls the following executables:

- 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.
- Machine and language specific compiler
  This compiler works on the pre-processed code, which is the output of the first stage. The language-specific compiler is one of the following:
  - C Compiler (cc1)
    The compiler responsible for most of the optimizations done on the input C code and for generating assembly code.
  - C++ Compiler (cc1plus)
    The compiler responsible for most of the optimizations done on the input C++ code and for generating assembly code.
- 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. It creates an object file, which is passed on to the linker.
- Linker (mb-ld for MicroBlaze and powerpc-eabi-ld for PowerPC)
  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 the executables listed above are described later in this chapter.

Note: All future references to *gcc* in this chapter refer to both the MicroBlaze compiler, `mb-gcc`, and the PowerPC compiler, `powerpc-eabi-gcc`. All future references to *g++* refer to the MicroBlaze C++ compiler, mb-g++, and the PowerPC C++ compiler, powerpc-eabi-g++.

Common Compiler Usage and Options

Usage

To use the GNU Compiler, type:

```
Compiler_Name options files...
```

where *Compiler_Name* is `powerpc-eabi-gcc` or `mb-gcc`. To compile C++ programs, you can use either the `powerpc-eabi-g++` or the `mb-g++` command.

Input Files

The compilers take one or more of the following files as input:

- C source files
- C++ source files
- Assembly files
- Object files
- Linker scripts

Note: These files are optional. If they are not specified, the default linker script embedded in the linker (mb-ld or powerpc-eabi-ld) is used.

The default extensions for each of these types are listed in Table 10-1. In addition to the files mentioned above, the compiler implicitly refers to the libraries files `libc.a`, `libgcc.a`, `libm.a`, and `libxil.a`. The default location for these files is the EDK installation directory. When using the g++ compiler, `libsupc++.a` and `libstdc++.a` are also referenced. These are the C++ language support and C++ platform libraries, respectively.

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

File Types and Extensions

The GNU compiler determines the type of your file from the file extension. Table 10-1 illustrates the valid extensions and the corresponding file types. The gcc wrapper calls the appropriate lower level tools by recognizing these file types.

Table 10-1: **File Extensions**

Extension	File type (Dialect)
.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

Libraries

Table 10-2 lists the libraries necessary for the *powerpc_eabi_gcc* and *mb_gcc* compilers, as follows.

Table 10-2: **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` and `strlen`.
`libgcc.a`	GCC low-level library containing emulation routines for floating point and 64-bit arithmetic.
`libm.a`	Math Library, containing functions like `cos` and `sine`.
`libsupc++.a`	C++ support library with routines for exception handling, RTTI , and others.
`libstdc++.a`	C++ standard platform library. Contains standard language classes, such as those for stream I/O, file I/O, string manipulation, and others.

All the libraries are automatically linked in by both compilers. If the standard libraries are overridden, the search path for these libraries must be given to the compiler. The `libxil.a` is modified by the Library Generator tool, Libgen, to add driver and library routines.

Language Dialect

The GCC compiler recognizes both C and C++ dialects and generates code accordingly. By GCC convention, it is possible to use either the gcc or the g++ compilers equivalently on a source file. The compiler that you use and the extension of your source file determines the dialect used on the input and output files.

When using the gcc compiler, the dialect of a program is always determined by the file extension, as listed in Table 10-1. If a file extension shows that it is a C++ source file, the language is set to C++. This means that if you have compile C code contained in a CC file, even if you use the gcc compiler, it automatically mangles function names.

The primary difference between gcc and g++ is that g++ automatically sets the default language dialect to C++ (irrespective of the file extension), and if linking, automatically pulls in the C++ support libraries. This means that even if you compile C code in a .c file with the g++ compiler, it will mangle names.

Name mangling is a concept unique to C++ and other languages that support overloading of symbols. A function is said to be overloaded if the same function can perform different actions based on the arguments passed in, and can return different return values. To support this, C++ compilers encode the type of the function to be invoked in the function name, avoiding multiple definitions of a function with the same name.

Be careful about name mangling if you decide to follow a mixed compilation mode, with some source files containing C code and some others containing C++ code (or using gcc for compiling certain files and g++ for compiling others). To prevent name mangling of a C symbol, you can use the following construct in the symbol declaration.

```
#ifdef __cplusplus
extern "C" {
£endif

int foo();
int morefoo();

#ifdef __cplusplus
}
£endif
```

Make these declarations available in a header file and use them in all source files. This causes the compiler to use the C dialect when compiling definitions or references to these symbols.

Note: All the EDK drivers and libraries follow the conventions listed above in all the header files they provide. You must include the necessary headers, as documented in each driver and library, when you compile with *g++*. This ensures that the compiler recognizes library symbols as belonging to "C" type.

When compiling with either variant of the compiler, to force a file to a particular dialect, use the `-x <lang>` switch. Refer to the GCC manual on the GNU website for more information on this switch.

When using the gcc compiler, libstdc++.a and libsupc++.a are NOT automatically linked in. When compiling C++ programs, use the g++ variant of the compiler to make sure all the required support libraries are linked in automatically. Adding *-lstdc++* and *-lsupc++* to the gcc command are also possible options.

For more information about how to invoke the compiler for different languages, refer to http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Invoking-G_002b_002b.html#Invoking-G_002b_002b.

Quick Reference

Table 10-3 lists the commonly used compiler options. These options are common to the compilers for MicroBlaze and PowerPC.

Note: The compiler options are case sensitive.

Compiler Options

Table 10-3 provides a list of commonly used compiler options with a brief description of each. Links to more detailed information about each option are provided where appropriate.

Table 10-3: **Commonly Used Compiler Options**

Options	Explanation
-E	Preprocess only; do not compile, assemble and link. The preprocessed output displays on the standard out device.
-S	Compile only; do not assemble and link. Generates a `.s` file.
-c	Compile and Assemble only; do not link. Generates a `.o` file.
-g	Add debugging information, which is used by the 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.
-O *n*	Specify Optimization level; *n = 0,1,2,3,S*.
-v	Verbose. Display the programs invoked by the compiler.
-save-temps	Store the intermediate files produced at the end of each pass.
-o filename	Place the output in the *filename*.
-Wp,option	Pass comma-separated *option* to the preprocessor.
-Wa,option	Pass comma-separated *option* to the assembler.
-Wl,option	Pass comma-separated *option* to the linker.
--help	Display a short listing of options.
-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.

a. The compiler prefixes “lib” to the library name indicated in this command line switch.

-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 and 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 assembly (`.S`) files and assembly file symbols at the source level. This assembler option and is 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,`**.

-On

The GNU compiler provides optimizations at different levels. These optimization levels apply only to the C and C++ source files.

Table 10-4: **Optimizations for Values of n**

n	Optimization
0	No Optimization.
1	Medium Optimization.
2	Full optimization.
3	Full optimization. Attempt automatic inlining of small subprograms.
S	Optimize for size.

Note: Optimization levels 1 and above cause code re-arrangement. While debugging your code, use of no optimization level is recommended. 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 discovering 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 compiler saves the default output of the entire compilation as `a.out`.

-o *filename*

The compiler stores the default output of the compilation process in an ELF file named `a.out`. You can change the default name using **`-o output_file_name.`** The output file is created in ELF format.

-Wp,*option*

-Wa,*option*

-Wl,*option*

The compiler, mb-gcc or powerpc-eabi-gcc, is a wrapper around other executables such as the preprocessor, compiler (cc1), assembler, and the linker. You can run these components of the compiler individually or through the top level compiler.

There are certain options that are required by tools, but might not be necessary for the top-level compiler. To run these commands, use the options listed in Table 10-5.

Table 10-5: **Tool-Specific Options Passed to the Top-Level GCC Compiler**

Option	Tool	Example
`-Wp,option`	Preprocessor	mb-gcc -Wp,-D -Wp,MYDEFINE ... Tell the pre-processor to define the symbol MYDEFINE with the -D MYDEFINE option.
`-Wa,option`	Assembler	powerpc-eabi-gcc -Wa,-m405 ... Tell the assembler to target the PPC405 processor with the -m405 option.
`-Wl,option`	Linker	mb-gcc -Wl,-M ... Tell the linker to produce a map file with the -M option.

--help

Use this option with any GNU compiler to get more information about the available options.

You can also consult the GCC manual, available online at http://www.gnu.org/manual/manual.html.

Library Search Options

Library search options are as follows.

`-l libraryname`

By default, the compiler searches only the standard libraries, such as libc, libm, and libxil. You can also create your own libraries. You can specify the name of the library and where the compiler can find the definition of these functions. The compiler prefixes `lib` to the library name that you provide.

The compiler is sensitive to the order in which you provide options, particularly the **`-l`** command line switch. Provide this switch only after all of the sources in the command line.

For example, if you create your own library called `libproject.a.`, you can include functions from this library using the following command:

```
Compiler Source_Files -L${LIBDIR} -lproject
```

Caution! If you supply the library flag **`-l library name`** before the source files, the compiler does not find the functions called from any of the sources. This is because the compiler search is only done in one direction and it does not keep a list of libraries available.

`-L Lib Directory`

This option indicates the directories in which to search for the libraries. The compiler has a default library search path, where it looks for the standard library. Using the **`-L`** option, you can include some additional directories in the compiler search path.

Header File Search Option

`-I Directory Name`

This option searches for header files in the *Directory Name* directory before searching the header files in the standard path.

Default Search Paths

The compilers, mb-gcc and powerpc-eabi-gcc, search certain paths for libraries and header files. The search paths on the various platforms are described below.

Solaris Search Paths

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the **`-L dir name`** option.
2. Directories are passed to the compiler with the **`-B dir name`** option.
3. The compilers search the following libraries:
   a. **`${XILINX_EDK}`**`/gnu/`*processor*`/sol/microblaze/lib`
   b. **`${XILINX_EDK}`**`/lib/`*processor*

   Note: *Processor* indicates *powerpc-eabi* for PowerPC and *microblaze* for MicroBlaze.

Header files are searched in the following order:

1. Directories are passed to the compiler with the **`-I dir name`** option.
2. The compilers search the following header files:
   a. **`${XILINX_EDK}`**`/gnu/`*processor*`/sol/lib/gcc/`*processor*`/3.4.1/include`
   b. **`${XILINX_EDK}`**`/gnu/`*processor*`/sol/lib/`*processor*`/sys-include`

Initialization files are searched in the following order:

1. Directories are passed to the compiler with the **`-B dir name`** option.
2. The compilers search **`${XILINX_EDK}`**`/gnu/`*processor*`/sol/`*processor*`/lib`.

On Windows Cygwin Shell

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the **-L *dir name*** option.
2. Directories are passed to the compiler with the **-B *dir name*** option.
3. The compilers search the following libraries:
   a. **%XILINX_EDK%**/gnu/*processor*/nt/*processor*/lib
   b. **%XILINX_EDK%**/lib/*processor*

The compilers search header files in the following order:

1. Directories are passed to the compiler with the **-I *dir name*** option.
2. The compilers search the following header files:
   a. **%XILINX_EDK%**/gnu/*processor*/nt/lib/gcc/*processor*/3.4.1/include
   b. **%XILINX_EDK%**/gnu/*processor*/nt/*processor*/sys-include

The compilers search initialization files in the following order:

1. Directories are passed to the compiler with the **-B *dir name*** option.
2. The compilers search **%XILINX_EDK%**/gnu/*processor*/nt/*processor*/lib.

On Linux

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the **-L *dir name*** option.
2. Directories are passed to the compiler with the **-B *dir name*** option.
3. The compilers search the following libraries:
   a. **$XILINX_EDK%**/gnu/*processor*/lin/*processor*/lib
   b. **$XILINX_EDK%**/lib/*processor*

The compilers search header files in the following order:

1. Directories are passed to the compiler with the **-I *dir name*** option.
2. The compilers search the following header files:
   a. **$XILINX_EDK%**/gnu/*processor*/lin/lib/gcc/*processor*/3.4.1/include
   b. **$XILINX_EDK%**/gnu/*processor*/lin/*processor*/sys-include

The compilers search initialization files in the following order:

1. Directories are passed to the compiler with the **-B *dir name*** option.
2. The compilers search **$XILINX_EDK%**/gnu/*processor*/lin/*processor*/lib.

Linker Options

Linker options are as follows.

-defsym _STACK_SIZE=value

The total memory allocated for the stack can be modified using this linker option. The variable *STACK_SIZE* is the total space allocated for the stack. The variable *STACK_SIZE* is given the default value of 100 words, or 400 bytes. If your program is expected to need

more than 400 bytes for stack and heap combined, it is recommended that you increase the value of *STACK_SIZE* using this option. The value is 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 tries to write in other, incorrect, sections of the program, leading to incorrect execution of the code.

Note: A minimum stack size of 16 bytes (0x0010) is required for programs linked with the Xilinx®-provided C runtime (CRT) files.

-defsym _HEAP_SIZE=value

The total memory allocated for the heap can be controlled by the value given to the variable_HEAP_SIZE. The default value of _HEAP_SIZE is 0.

Dynamic memory allocation routines use the heap. If your program uses the heap in this fashion, then you must provide a reasonable value for _HEAP_SIZE.

Memory Layout

The MicroBlaze and PowerPC processors use 32-bit logical addresses and can address any memory in the system in the range 0x0 to 0xFFFFFFFF. This address range can be categorized into the following types:

- Reserved memory
- I/O memory
- User and program memory

Reserved Memory

Reserved memory has been defined by the hardware and software programming environment for privileged use. This is typically true for memory containing interrupt vector locations and operating system level routines. Table 10-6 lists the reserved memory locations for MicroBlaze and PowerPC as defined by the processor hardware. For more information on these memory locations, refer to the corresponding processor reference manuals.

Note: In addition to these memories that are reserved for hardware use, your software environment can reserve other memories. Refer to the manual of the particular software platform that you are using to find out if any memory locations are deemed reserved.

Table 10-6: **Hardware Reserved Memory Locations**

Processor Family	Reserved Memories	Reserved Purpose	Default Text Start Address
MicroBlaze	0x0 - 0x4F	Reset, interrupt, exception and other reserved vector locations	0x50
PowerPC	0xFFFFFFFC - 0xFFFFFFFF	Reset vector location	0xFFFF0000

I/O Memory

I/O memory refers to addresses used by your program to communicate with memory-mapped peripherals on the processor buses. These addresses are defined as a part of your hardware platform specification.

User and Program Memory

User and Program memory refers to all the memory that is required for your compiled executable to run. By convention, this includes memories for storing instructions, read-only data, read-write data, program stack, and program heap. These sections can be stored in any addressable memory in your system. By default the compiler generates code and data starting from the address listed in Table 10-6 and occupying contiguous memory locations. This is the most common memory layout for programs. You can modify the starting location of your program by defining (in the linker) the symbol `_TEXT_START_ADDR` for MicroBlaze and `_START_ADDR` for PowerPC.

In special cases, you might want to partition the various sections of your ELF file across different memories. This is done using the linker command language (refer to the "Linker Scripts" section of this chapter for details). The following are some situations in which you might want to change the memory map of your executable:

- When partitioning large code segments across multiple smaller memories
- Remapping often executed sections to fast memories
- Mapping read-only segments to non-volatile flash memories

No restrictions apply to how you can partition your executable. The partitioning is done at the output section level, or even at the individual function and data level. The resulting ELF file can have "holes," sections of memory that are not accessed by the program. The instructions and data in such an ELF are non-contiguous. Take care to not use the reserved locations documented. Alternatively, if you are an advanced user who wants to modify the default binary data provided by the tools for the reserved memory locations, you can always do so. You must replace the default startup files and the memory mappings provided by the linker.

Object-File Sections

An executable file is created by concatenating input sections from the object files (.o files) being linked together. The compiler, by default, creates code across standard and well-defined sections. Each section is given a well-known name based on its associated meaning and purpose. The various standard sections of the object file are displayed in Figure 10-2. In addition to these sections, you can also create your own custom sections and assign it to memories of your choice.

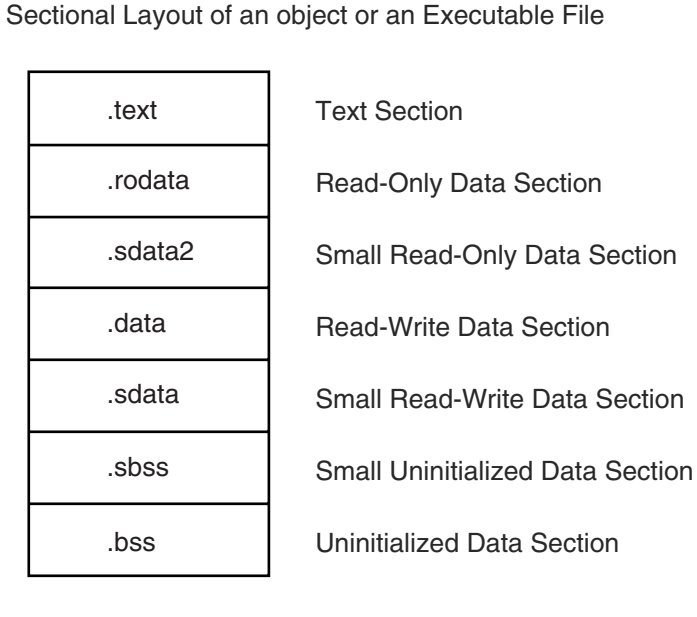

Figure 10-2: **Sectional Layout of an Object or Executable File**

In addition to these sections, you can also create custom sections and assign them to memories of your choice, as follows.

.text

This section of the object file contains executable program instructions. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags. This means that this section can be assigned to an initialized read-only memory (ROM) that is addressable from the processor instruction bus.

.rodata

This section contains read-only data. This section has the `r` (read-only) and the `i` (initialized) flags. Like the `.text` section, this section can also be assigned to an initialized, read-only memory that is addressable from the processor data bus.

.sdata2

This section is similar to the .rodata section. It contains small read-only data of size less than 8 bytes. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all the contents of this section are accessed using a single instruction. You can change the size of the data going into this section with the **`-G`** option to the compiler. This section has the `r` (read-only) and the `i` (initialized) flags.

.data

This section contains read-write data and has the `w` (read-write) and the `i` (initialized) flags. It must be mapped to initialized random access memory (RAM). It cannot be mapped to a ROM.

.sdata

This section contains small read-write data of a size less than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **`-G`** option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all contents of the section can be accessed using a single instruction. This section has the `w` (read-write) and the `i` (initialized) flags and must be mapped to initialized RAM.

.sbss

This section contains small un-initialized data of a size less than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **`-G`** option. This section has the `w` (read-write) flag and must be mapped to RAM.

.bss

This section contains un-initialized data. The program stack and the heap are also allocated to this section. This section has the `w` (read-write) flag and must be mapped to RAM.

.heap

This section contains uninitialized data that is used as the global program heap. Dynamic memory allocation routines allocate memory from this section. This section must be mapped to RAM.

.stack

This section contains uninitialized data that is used as the program stack. This section must be mapped to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack` and `.heap` sections might appear merged together into a section named `.bss_stack`.

.init

This section contains language initialization code and has the same flags as `.text`. It must be mapped to initialized ROM.

.fini

This section contains language cleanup code and has the same flags as `.text`. It must be mapped to initialized ROM.

.ctors

This section contains a list of functions that must be invoked at program startup and the same flags as `.data` and must be mapped to initialized RAM.

.dtors

This section contains a list of functions that must be invoked at program end, the same flags as `.data`, and it must be mapped to initialized RAM.

.got2/.got

This section contains pointers to program data, the same flags as `.data`, and it must be mapped to initialized RAM.

.eh_frame

This section contains frame unwind information for exception handling. It contains the same flags as `.rodata` and can be mapped to initialized ROM

Linker Scripts

The linker utility uses commands specified in linker scripts to divide your program on different blocks of memories. It describes the mapping between all of the sections in all of the input object files to output sections in the executable file. The output sections are mapped to memories in the system.

You do not need a linker script if you do not want to change the default contiguous assignment of program contents to memory. There is a default linker script provided with the linker that knows how to place section contents contiguously. You can always selectively modify only the starting address of your program by defining the linker symbol _TEXT_START_ADDR on MicroBlaze, or _START_ADDR on PowerPC, as displayed in this example:

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100

powerpc-eabi-gcc <input files and flags> -Wl,-defsym -
Wl,_TEXT_START_ADDR=0x2000

mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

The choices of the default script that will be used by the linker from the *$XILINX_EDK*`/gnu/<processor_name>/<platform>/<processor_name>/lib/ldscripts` area are described as follows:

- `elf32<`*procname*`>.x` is used by default when none of the following cases apply.
- `elf32<`*procname*`>.xn` is used when the linker is invoked with the `-n` option.
- `elf32<`*procname*`>.xbn` is used when the linker is invoked with the `-N` option.
- `elf32<`*procname*`>.xr` is used when the linker is invoked with the `-r` option.
- `elf32<`*procname*`>.xu` is used when the linker is invoked with the `-Ur` option.

  where `<`*procname*`>` = ppc or microblaze, `<`*processor_name*`>` = powerpc-eabi or microblaze, and `<`*platform*`>` = `lin`, `nt` or `sol`.

To use a linker script, provide it on the `gcc` command line. Use the command line option `-Wl,-T -Wl,<script>` for the compiler, as described below:

compiler -Wl,-T -Wl,linker_script <Other Options and Input Files>

If the linker is executed on its own, include the linker script as follows:

linker -T linker_script <Other Options and Input Files>

This tells GCC to use your linker script in the place of the default built-in linker script. Linker scripts are generated for your program from within XPS. In XPS, select **Tools** → **Generate Linker Script**. This opens up the linker script generator utility. Mapping sections to memory is done here. Stack and heap size can be set along with their memory mapping. Once the linker script is generated, it is automatically given as input to GCC when the application is compiled within XPS. The linker script defines the layout and the start address of each of the sections for the output executable file.

For more details on the linker scripts, refer to the GNU loader documentation in the binutil online manual, located at http://www.gnu.org/manual.

MicroBlaze Compiler Usage and Options

The MicroBlaze GNU compiler is derived from the standard GNU sources as Xilinx's port of the compiler. The features and options that are unique to the MicroBlaze compiler are described in the sections that follow. When compiling with the MicroBlaze compiler, the pre-processor automatically provides the definition *__MICROBLAZE__*. You can use this definition in any conditional code that you have.

MicroBlaze Compiler

The mb-gcc compiler for the Xilinx 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.

Compiler Options Quick Reference

Table 10-7: **MicroBlaze-Specific Options**

Options	Short Description
`-mcpu=v`*`X.YY.Z`*	This option directs the compiler to generate code suited to MicroBlaze hardware version v.X.YY.Z
`-xl-mode-executable`	Use this option to compile regular embedded applications for MicroBlaze. This is the default mode for compilation.
`-xl-mode-xmdstub`	Use this option to compile applications being debugged in a software-intrusive manner (XMDSTUB) on the board. Use this option only with the XMDStub downloaded on to MicroBlaze.
`-xl-mode-bootstrap`	Use this option to compile applications that are bootloaded using a bootloader. Applications compiled with this switch lack a reset vector.
`-xl-mode-novectors`	Use this option to compile applications that do not need any of the MicroBlaze vectors.
`-mxl-gp-opt`	Use the small data area anchors. Optimize for performance and size.

Table 10-7: **MicroBlaze-Specific Options *(Continued)***

Options	Short Description
`-mxl-soft-mul`	Use the software routine for all multiply operations. Use this option for devices without the hardware multiplier. This is the default option in the mb-gcc compiler.
`-mno-xl-soft-mul`	Use the hardware multiplier. The compiler generates the *mul* instructions.
`-mxl-soft-div`	Use the software routine for all divide operations. This is the default option.
`-mno-xl-soft-div`	Use the hardware divider. The compiler generates *div* instructions.
`-mxl-stack-check`	Generates code for checking stack for overflow at run-time.
`-mxl-barrel-shift`	Use hardware barrel shifter. The compiler generates the hardware barrel shift instructions.
`-mxl-pattern-compare`	Use pattern compare instructions while generating code. Use this option when the MicroBlaze hardware has the pattern compare option enabled.
`-mhard-float`	Generate hardware floating point instructions. Use this option when MicroBlaze has the hardware floating point unit (FPU) enabled.
`-msmall-divides`	Generate code optimized for small divides, when no hardware divider exists. For signed integer divisions where the numerator and denominator are between 0 and 15 inclusive, this switch provides very fast table-lookup-based divisions. This switch has no effect when the hardware divider is enabled. ***Note:*** This switch entails a code size increase of 256 bytes if divisions are present in the code.

-mcpu=v*X.YY.Z*

This option directs the compiler to generate code suited to MicroBlaze hardware version v.X.YY.Z. To get the most optimized and correct code for a given processor, use this switch with the hardware version of the processor.

The `-mcpu` switch behaves differently for different versions, as described below:

- Pr-v3.00.a

  Uses 3-stage processor pipeline mode. Does not inhibit exception causing instructions being moved into delay slots. Does not use pattern comparator instructions by default.

- v3.00.a and v4.00.a

  Uses 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots. Does not use pattern comparator instructions by default.

- v5.00.a and later

  Uses 5-stage processor pipeline model. Does not inhibit exception causing instructions from being moved into delay slots. Uses pattern comparator instructions by default.

-mxl-soft-mul

In some devices, a hardware multiplier is not present. In such cases, you can either build the multiplier in hardware or use the provided software multiplier library routine. The MicroBlaze compiler assumes by default that the target device does not have a hardware multiplier, so every 32-bit multiply operation is replaced by a call to the software emulation routine__**mulsi3**.

-mno-xl-soft-mul

This option permits use of hardware multiply instructions for 32-bit multiplications. The MicroBlaze processor has an option to turn the use of hardware multiplier resources on or off. This option should be used when the hardware multiplier option is enabled on MicroBlaze. Using the hardware multiplier can improve the performance of your application. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Manual* (in the `/doc` directory of you EDK installation) for more details about the usage of the multiplier option in MicroBlaze.

-mxl-soft-div

This option tells the compiler that there is no hardware divide unit on the target MicroBlaze hardware. This option is the default. The compiler replaces all 32-bit divisions with a call to the corresponding software emulation routines (__**divsi3**, __**udivsi3**).

-mno-xl-soft-div

You can instantiate a hardware divide unit in MicroBlaze. When the divide unit is present, this option tells the compiler that hardware divide instructions can be used in the program being compiled. This option can improve the performance of your program if it has a significant amount of division operations. The compiler automatically defines the C pre-processor definition `HAVE_HW_DIV` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Manual* for more details about the usage of the hardware divide option in MicroBlaze.

-mxl-stack-check

With this option, you can check whether 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.

You can override the standard stack overflow handler 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 assumes that no barrel shifter is present, and the compiler uses add and multiply operations to shift the operands. Enabling barrel shifts can speed up your application significantly, especially while using a floating point library. The compiler automatically

defines the C pre-processor definition `HAVE_HW_BSHIFT` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether or not this feature is specified as available. Refer to the *MicroBlaze Processor Reference Manual* for more details about the use of the barrel shifter option in MicroBlaze.

-mxl-gp-opt

If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), then load/store operations to that address require two instructions. MicroBlaze ABI offers two global small data areas that can contain up to 64 K 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, needing only one instruction for a 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 and can be addressed with fewer instructions. The addresses are calculated during the linking stage.

-mxl-pattern-compare

This option activates the use of pattern compare instructions in the compiler. Using pattern compare instructions can speed up boolean operations in your program. Pattern compare operations also permit operating on word-length data as opposed to byte-length data on string manipulation routines such as strcpy, strlen, and strcmp. On a program heavily dependent on string manipulation routines, the speed increase obtained will be significant. The compiler automatically defines the C pre-processor definition `HAVE_HW_PCMP` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Manual* for more details about the use of the pattern compare option in MicroBlaze.

-mhard-float

This option turns on the usage of `fadd`, `frsub`, `fmul`, and `fdiv` instructions in the compiler. It also uses fcmp.*p* instructions, where ***p*** is a predicate condition such as le, ge, lt, gt, eq, ne. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware. The compiler automatically defines the C pre-processor definition `HAVE_HW_FPU` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not. Refer to the *MicroBlaze Processor Reference Manual* for more details about the use of the hardware floating point unit option in MicroBlaze.

-mno-clearbss

This option is useful for compiling programs used in simulation. According to the C language standard, uninitialized global variables are allocated in the `.bss` section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the `.bss` section with zero when the program starts execution. Optimizing compilers also allocates global variables that are assigned zero in C code to the `.bss` section.

In a simulation environment, the above two language features can be unwanted overhead. Some simulators automatically zero the entire memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios. It causes the C startup files to not initialize the `.bss` section with zeroes. It also internally forces the compiler to not allocate zero-initialized global

variables in the `.bss` and instead move them to the `.data` section. This option might improve startup times for your application. Use this option with care and ensure either that you do not use code that relies on global variables being initialized to zero, or that your simulation platform performs the zeroing of memory.

Application Execution Modes

-xl-mode-executable

This is the default mode used for compiling programs with mb-gcc. This option need not be provided on the command line for mb-gcc. This uses the startup file `crt0.o`.

-xl-mode-xmdstub

The Xilinx Microprocessor Debugger (XMD) allows debugging of applications in a software-intrusive manner. This mode is known as XMDSTUB mode. Compile programs being debugged in such a manner with this switch. In such programs, the address locations 0x0 to 0x800 are reserved for use by XMDStub. Using -xl-mode-xmdstub has two effects:

- The start address of your program is set to 0x800. You 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 "Linker Options," page 108. 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 your program execution is complete.

Note: Use -xl-mode-xmdstub for designs when XMDStub is part of the bitstream. Do not use this mode when the system is complied for No Debug or when "Hardware Debugging" is turned ON. For more details on debugging with XMD, refer to Chapter 12, "Xilinx Microprocessor Debugger (XMD)".

-xl-mode-bootstrap

This option is used for applications that are loaded using a bootloader. Typically, the bootloader resides in non-volatile memory mapped to the processor reset vector. If a normal executable is loaded by this bootloader, the application reset vector overwrites the reset vector of the bootloader. In such a scenario, on a processor reset, the bootloader does not execute first (it is typically required to do so) to reload this application and do other initialization as necessary.

To prevent this, you must compile the bootloaded application with this compiler flag. On a processor reset, control then reaches the bootloader instead of the application. Using this switch on an application that is deployed in a scenario different from the one described above will not work. This mode uses `crt2.o` as a startup file.

-xl-mode-novectors

This option is used for applications that do not require any of the MicroBlaze vectors. This is typically used in standalone applications that do not use any of the processor's reset, interrupt, or exception features. Using this switch leads to smaller code size due to the elimination of the instructions for the vectors. This mode uses `crt3.o` as a startup file.

Caution! Do not use more than one mode of execution on the command line. You will receive link errors due to multiple definition of symbols if you do so.

MicroBlaze Assembler

The mb-as assembler for the Xilinx 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 Processor Reference Manual*.

The mb-as assembler requires all MicroBlaze 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 computes it and includes 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. Use the `relax` option of the linker remove any unnecessary `imm` instructions.

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 adds the constant 200,000 to the contents of register r3, and stores the results in register r4:

```
addi r4, r3, 200000
```

The mb-as assembler recognizes that this operand needs an `imm` instruction, and inserts 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. Table 10-8 lists the supported pseudo-opcodes.

Table 10-8: **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 the Xilinx 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 0x28 (0x800 in XMDStub mode). This can be overridden by using the above options. If this is supplied to mb-gcc, the text section of the output code starts from the given *value*.

You do not have to use **-defsym _TEXT_START_ADDR** if you want to use the default start address set by the compiler.

This is a linker option and should be used when you invoke the linker separately. If the linker is being invoked as a part of the mb-gcc flow, you must use the following option

```
-Wl,-defsym -Wl,_TEXT_START_ADDR=value
```

-relax

This is a linker option that removes all unwanted `imm` instructions generated by the assembler. The assembler generates an `imm` instruction for every instruction where the value of the immediate cannot be calculated during the assembler phase. Most of these instructions do not 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 as 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, use this option.

For more details on this option, refer to the GNU manuals online at http://www.gnu.org/manual/manual.html.

Startup Files

The compiler includes pre-compiled startup and end files in the final link command when forming an executable. Startup files set up the language and the platform environment before your application code executes. The following actions are typically performed by startup files:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers. Refer to Table 10-9 for details.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions, such as C++ constructors.
- Initialize the hardware sub-system. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for the main procedure and invoke it.

Similarly, end files are used to include code that must execute after your program ends. The following actions are typically performed by end files:

- Invoke language cleanup functions, such as C++ destructors.
- De-initialize the hardware sub-system. For example, if the program is being profiled, clean up the profiling sub-system.

Table 10-9: **Register initialization in the C-Runtime files**

Register	Value	Description
`r1`	`_stack-16`	The stack pointer register is initialized to point to the bottom of the stack area with an initial negative offset of 16 bytes. The 16 bytes can be used for passing in arguments.
`r2`	`_SDA2_BASE`	_SDA2_BASE_ is the read-only small data anchor address.
`r13`	`_SDA_BASE_`	_SDA_BASE is the read-write small data anchor address.
`Other registers`	`Undefined`	Other registers do not have defined values.

The rest of this section describes the initialization files used for various application modes. This information is for advanced users who want to change or understand the startup code of their application. For MicroBlaze, there are two distinct stages of C run-time initialization. The first stage is primarily responsible for setting up vectors, after which it invokes the second stage initialization. It also provides exit stubs based on the different application modes.

First Stage Initialization Files

crt0.o

This initialization file is used for programs which are to be executed in standalone mode, without the use of any bootloader or debugging stub such as **xmdstub**. This CRT populates the reset, interrupt, exception, and hardware exception vectors and invokes the second stage startup routine **_crtinit**. On returning from **_crtinit**, it ends the program by infinitely looping in the exit label.

crt1.o

This initialization file is used when the application is debugged in a software-intrusive manner. It populates all the vectors *except the breakpoint and reset vectors* and transfers control to the second-stage **_crtinit** startup routine. On returning from _crtinit it returns program control back to the XMDStub, which signals to the debugger that the program has finished.

crt2.o

This initialization file is used when the executable is loaded using a bootloader. It populates all the vectors *except the reset vector* and transfers control to the second-stage **_crtinit** startup routine. On returning from **_crtinit,** it ends the program by infinitely looping at the exit label. Because the reset vector is not populated, on a processor reset, control is transferred to the bootloader, which can reload and restart the program.

crt3.o

This initialization file is employed when the executable does not use any vectors and wishes to reduce code size. It populates only the reset vector and transfers control to the second stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `exit` label. Because the other vectors are not populated, the GNU linking mechanism does not pull in any of the interrupt and exception handling related routines, thus saving code space.

Second Stage Initialization Files

According to the C standard specification, all global and static variables must be initialized to 0. This is a common functionality required by all the CRTs above. Another routine `_crtinit` is invoked. The `_crtinit` routine initializes memory in the `.bss` section of the program. `_crtinit` is also the wrapper that invokes the `main` procedure. Before invoking the main procedure, it may invoke other initialization functions. `_crtinit` is supplied by the following startup files, as described below.

crtinit.o

This is the default second stage C startup file. This startup file performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes "constructor" functions (`__init`).
4. Sets up the arguments for `main` and invokes `main`.
5. Invokes "destructor" functions (`__fini`).
6. Invokes `_program_clean` and returns.

pgcrtinit.o

This second stage startup file is used during profiling. This startup files performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes `_profile_init` to initialize the profiling library.
4. Invokes "constructor" functions (`__init`).
5. Sets up the arguments for `main` and invokes `main`.
6. Invokes "destructor" functions (`__fini`).
7. Invokes `_profile_clean` to cleanup the profiling library.
8. Invokes `_program_clean` and then returns.

sim-crtinit.o

This second-stage startup file is used when the *-mno-clearbss* switch is used in the compiler. This startup file performs the following steps:

1. Invokes `_program_init`.
2. Invokes "constructor" functions (`__init`).
3. Sets up the arguments for `main` and invokes `main`.
4. Invokes "destructor" functions (`__fini`).

5. Invokes `_program_clean` and then returns.

sim-pgcrtinit.o

This second stage startup file is used during profiling in conjunction with the *-mno-clearbss* switch. This startup files performs the following steps in order:

1. Invokes `_program_init`.
2. Invokes `_profile_init` to initialize the profiling library.
3. Invokes "constructor" functions (`__init`).
4. Sets up the arguments for `main` and invokes `main`.
5. Invokes "destructor" functions (`__fini`).
6. Invokes `_profile_clean` to cleanup the profiling library.
7. Invokes `_program_clean` and then returns.

Other files

The compiler also uses certain standard start and end files for C++ language support. These are `crti.o`, `crtbegin.o`, `crtend.o`, and `crtn.o`. These files are standard compiler files that provide the content for the `.init`, `.fini`, `.ctors`, and `.dtors` sections.

Modifying Startup Files

The initialization files are distributed in both pre-compiled and source form with EDK. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the MicroBlaze GNU compiler can be found in the `<XILINX_EDK>/sw/lib/microblaze/src` directory, where <*XILINX_EDK*> is the EDK installation area.

To fulfill a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, you can assemble the files into `.o` files and place them in a common area. 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. To prevent the default startup files from being used, use `-nostartfiles` on final compile line. Note that the miscellaneous compiler standard CRT files, such as `crti.o`, and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You may need to bring them in on your final link command.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you may wish to eliminate all sources of overhead. This section describes how to reduce the overhead of invoking the C++ constructor or destructor code in a C program that does not need them. You maybe able to save approximately 220 bytes of code space by making these modifications.

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, copy over the particular versions of `crtn.s` and *x*`crtinit.s` that suit your application. For example, if your application is being bootstrapped and profiled, copy `crt2.s` and `pg-crtinit.s` from the installation area.

2. Modify **pg-crtinit.s** to remove the following lines:

```
brlid   r15, __init /* Invoke language initialization functions
*/
nop
```

and

```
brlid   r15, __fini /* Invoke language cleanup functions */
nop
```

This avoids referencing the extra code usually pulled in for constructor and destructor handling, reducing code size.

3. Compile these files into `.o` files and place them in a directory of your choice, or include them as a part of your application sources.
4. Add the `-nostartfiles` switch to the compiler. Add the `-B <directory>` switch if you have chosen to assemble the files in a particular folder.
5. Compile your application.

If your application is executing in a different mode, then you must pick the appropriate CRT files based on the description in "Startup Files," page 120.

Compiler Libraries

The mb-gcc compiler requires the GNU C standard library and the GNU math library. Precompiled versions of these libraries are shipped with EDK. The CPU driver for MicroBlaze copies over the correct version, based on the hardware configuration of MicroBlaze, during the execution of Libgen. To manually select the library version that you would like to use, look in the following folder:

`$XILINX_EDK/gnu/microblaze/<platform>/microblaze/lib`

The filenames are encoded based on the compiler flags and configurations used to compile the library. For example, `libc_m_bs.a` is the C library compiled with hardware multiplier and barrel shifter enabled in the compiler.

Of special interest are the math library files (`libm*.a`). Using MicroBlaze with a single-precision hardware Floating-Point Unit (FPU), you might want to use the version of the library that is tailored to best use the hardware unit, instead of the default floating point emulation. The CPU driver by default picks up the double precision (`libm_*_fpd.a`) version of the library. However, you can use the single-precision version of the library because it might be sufficient and more efficient for your application. In this case, copy over the corresponding, `libm_*_fps.a` file into your processor library folder (such as `microblaze_0/lib`) as `libm.a`.

Once you have copied over the library that you want to use, rebuild your application software project. The library that you copied over is used to build the final executable.

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 must be compiled in a different manner than normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers must save the volatile registers that are being used. Interrupt handlers should also store the value of the machine status register (RMSR) when an interrupt occurs.

_interrupt_handler attribute

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: The attribute for the interrupt handler is to be given only in the prototype and not in the definition.

Interrupt handlers might also call other functions, which might use volatile registers. To maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function.

Note: Functions that have calls to other sub-routines are called *non-leaf* functions.

Interrupt handlers are defined in the MicroBlaze Hardware Specification (MHS) and the MicroBlaze Software Specification (MSS) files. These definitions automatically add the attributes to the interrupt handler functions. For more information, refer to Appendix B, "Interrupt Management."

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 attribute saves all the volatiles for non-leaf functions and only the used volatiles in the case of leaf functions.

```
void function_name () __attribute__((save_volatiles));
```

Table 10-10 lists the attributes with their functions.

Table 10-10: **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` returns 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 uses `rtsd` to return to the interrupted function, instead of `rtid`.

PowerPC Compiler Usage and Options

Compiler Options

The PowerPC GNU compiler (*powerpc-eabi-gcc*) is built out of the sources for the PowerPC port as distributed by GNU foundation. The compiler is customized slightly for Xilinx purposes. The features and options that are unique to the version distributed with EDK are described in the following sections. When compiling with the PowerPC compiler, the pre-processor automatically provides the definition *__PPC__*. You can use this definition in any conditional code that you have.

-mfpu={sp_lite, sp_full}

Generate hardware floating point instructions to use with the Xilinx PowerPC APU FPU coprocessor hardware. The instructions and code output follow the floating point specification in the PowerPC Book-E, with some exceptions tailored to the APU FPU hardware. Information about Book-E is available at http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600682CC7. Refer to the FPU hardware documentation for more information on the architecture.

The option given to *-mfpu=* determines which variant of the FPU hardware to target. The variants are as follows:

- `sp_lite`
  Produces code targeted to the Single precision Lite FPU coprocessor. This version supports only single precision hardware floating point and does not use hardware divide and square root instructions.
- `sp_full`
  Produces code targeted to the Single precision Full FPU coprocessor. This version supports only single precision hardware floating point and uses hardware divide and square root instructions.

  Caution! Do not link code compiled with one variant of the `-mfpu` switch with code compiled with other variants (or without the `-mfpu` switch). You must use the switch even when you are only linking object files together. This allows the compiler to use the correct set of libraries and prevent incompatibilities.

Refer to the APU FPU user guide, located at `$XILINX\doc\apu_fpu_v2_01_a.pdf`, for detailed information on how to optimally use the hardware floating point co-processor.

-msoft-float

This option is enabled by default and tells the compiler to use software emulation libraries for floating point code.

-mppcperflib

Use IBM performance libraries for low-level integer and floating emulation, and some simple string routines. These libraries are used in the place of the default emulation routines provided by GCC and simple string routines provided by Newlib. The IBM performance libraries show an average of three times increase in speed on applications that heavily use these routines. The SourceForge project Web page contains more information and detailed documentation. It is available at http://sourceforge.net/projects/ppcperflib.

Caution! You cannot use the performance libraries in conjunction with the `-mfpu` switch.

-mno-clearbss

This option is useful for compiling programs used in simulation. According to the C language standard, uninitialized global variables are allocated in the .bss section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the `.bss` section with zero when the program starts execution. Additionally optimizing compilers will also allocate global variables that are assigned zero in C code to the `.bss` section.

In a simulation environment, the two language features above can be unwanted overhead. Some simulators automatically zero the whole memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios. It causes the C startup files to not initialize the `.bss` section with zeroes. It also internally forces the compiler not to allocate zero-initialized global variables in the `.bss` and instead move them to the `.data` section. This option may improve startup times for your application. Use this option with care. Do not use code that relies on global variables being initialized to zero, or ensure that your simulation platform performs the zeroing of memory.

Linker Options

-defsym _START_ADDR=*value*

By default, the text section of the output code starts with the base address 0xffff0000 because this is the start address listed in the default linker script. This can be overridden by using the above option or providing a linker script that lists the value for the start address.

You are not required to use `-defsym _START_ADDR`, if you want to use the default start address set by the compiler.

This is a linker option. Use this option when you invoke the linker separately. If the linker is being invoked as a part of the powerpc-eabi-gcc flow, use the option `-Wl,-defsym -Wl,_START_ADDR=`*`value`*.

Startup Files

When the compiler forms an executable, it includes pre-compiled startup and end files in the final link command. Startup files set up the language and the platform environment before your application code can execute. The following actions are typically performed by startup files:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers as required.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions such as C++ constructors.
- Initialize the hardware sub-system. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for the main procedure and invoke it.

End files are used to include code that must execute after your program is finished. The following actions are typically performed by end files:

- Invoke language cleanup functions, such as C++ destructors.
- Clean up the hardware sub-system. For example, if the program is being profiled, clean up the profiling sub-system.

Table 10-11: **Register initialization in the C-Runtime files**

Register	Value	Description
r1	**_stack-8**	Stack pointer register initializes the bottom of the allocated stack, offset by 16 bytes. The 16 bytes can be used for passing in arguments.
r2	**_SDA2_BASE**	_SDA2_BASE_ is the read-only small data anchor address.
r13	**_SDA_BASE_**	_SDA_BASE is the read-write small data anchor address.
Other registers	**Undefined**	Other registers do not have defined values.

The rest of this section describes the initialization files. This information is for advanced users who want to change or understand the startup code of their application.

The PowerPC compiler uses four different CRT files – `xil-crt0.o`, `xil-pgcrt0.o`, `xil-sim-crt0.o`, and `xil-sim-pgcrt0.o`. The various CRT files perform the following steps, with exceptions as described.

1. Invoke the function `__cpu_init`. This function is provided by the board support package library and contains processor architecture specific initialization.
2. Clear the BSS memory regions to zero.
3. Set up registers. Refer to Table 10-11 for details.
4. Initialize the timer base register to zero.
5. Optionally, enable the floating point unit bit in the MSR.
6. Invoke `main`.
7. Invoke C++ language destructors (`__fini`).
8. Transfer control to `exit`.

Additionally, main invokes the library routine `__eabi`. This routine is responsible for invoking the C++ language initialization functions (`__init`).

The routine `__eabi` is defined and used differently in other architectures and or distributions of the PowerPC toolchain. Specifically, some distributions use `__eabi` to setup r2 and r13 registers for small-data support. In the EDK distribution, these actions are performed by the main CRT file instead.

xil-crt0.o

This is the default initialization file used for programs that are to be executed in standalone mode, with no other special requirements. This performs all the common actions described above.

xil-pgcrt0.o

This initialization file is used when the application is to be profiled in a software-intrusive manner. In addition to all the common CRT actions described, it also invokes the `_profile_init` routine before invoking `main`. This initializes the software profiling library before your code executes. Similarly, upon exit from `main`, it invokes the `_profile_clean` routine, which cleans up the profiling library.

xil-sim-crt0.o

This initialization file is used when the application is compiled with the *-mno-clearbss* switch. It performs all the common CRT setup actions, except that it does not clear the `.bss` section to zero.

xil-sim-pgcrt0.o

This initialization file is used when the application is compiled with the *-mno-clearbss* switch. It performs all the common CRT setup actions, except that it does not clear the `.bss` section to zero. It also invokes the `_profile_init` routine before invoking `main`. This initializes the software profiling library before your code executes. Similarly, upon exit from `main`, it invokes the `_profile_clean` routine, which cleans up the profiling library.

Other files

The compiler also uses certain standard start and end files for C++ language support. These are `ecrti.o`, `crtbegin.o`, `crtend.o`, and `crtn.o`. These files are standard compiler files that provide the content for the .init, .fini, .ctors and .dtors sections.

The PowerPC default and generated linker scripts also make boot.o a startup file. This file is present in the standalone Board Support Package for PowerPC.

Modifying Startup Files

The initialization files are distributed in both pre-compiled and source form with EDK. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the PowerPC compiler can be found in the `<XILINX_EDK>/sw/lib/ppc405/src` directory, where <XILINX_EDK> is the EDK installation area.

Any time you need a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, they can be assembled into `.o` files and placed in a common area. To refer to the newly created object files instead of the standard files, use the `-B directory-name` command line option while invoking powerpc-eabi-gcc. To prevent the default startup files being used, add `-nostartfiles` on final compile line. Note that the compiler standard CRT files for C++ support, such as `ecrti.o` and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You may need to bring them in on your final link command if your code uses constructors and destructors.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you can eliminate all sources of overhead. This section documents how to remove the overhead of invoking the C++ constructor or destructor code in a C program that does not need them.

You might be able to save approximately 500 bytes of code space by making these modifications.

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, you need to copy over the particular version of `xil-crt.s` that suits your application. For example, if your application is being profiled, copy `xil-pgcrt0.s` from the installation area.
2. Modify the CRT file to remove the following lines:

   ```
   /* Invoke the language cleanup functions */
   bl      __fini
   ```

   and add the following lines at the end of the file:

   ```
   .text
   .align 2
   .global __eabi
   __eabi: blr
   ```

   This avoids referencing the extra code that is usually pulled in for constructor and destructor handling, and reducing code size.
3. Either compile these files into .o files and place them in a directory of your choice, or include them as a part of your application sources.
4. Add the `-nostartfiles` switch to the compiler. Add the `-B <directory>` switch if you have chosen to assemble the files in a particular folder.
5. Compile your application.

Modifying Startup Files for Bootstrapping an Application

If your application is going to be loaded from a bootloader, you might not want to overwrite the bootloader's processor reset vector with that of your application. This re-executes the bootloader on a processor reset instead of your application. To achieve this, your application must not bring in `boot.o` as a startup file. Unlike other compiler startup files, `boot.o` is not explicitly linked in by the compiler. Instead, the default linker scripts and the tools for generating the linker scripts specify `boot.o` as a startup file. You must remove the *`STARTUP`* directive in such linker scripts. You must also modify the *`ENTRY`* directive to be `_start` instead of `_boot`.

Compiler Libraries

The `powerpc-eabi-gcc` compiler requires the GNU C standard library and the GNU math library.

Precompiled versions of these libraries are shipped with EDK. These libraries are located in `$XILINX_EDK/gnu/powerpc-eabi/<platform>/powerpc-eabi/lib`.

Other Notes

C++ Code Size

The GCC toolchain combined with the latest open source C++ standard library (`libstdc++-v3`) might be found to generate large code and data fragments as compared to an equivalent C program. A significant portion of this overhead comes from code and data for exception handling and runtime type information. Some C++ applications do not require these features. To remove the overhead and optimize for size, use the `-fno-exceptions` and/or the `-fno-rtti` switches. This is recommended only for advanced users who know the requirements of their application and understand these language features. Refer to the GCC manual for more specific information on available compiler options and their impact.

C++ programs might have more intensive dynamic memory requirements (stack and heap size) due to more complex language features and library routines. Many of the C++ library routines can request memory to be allocated from the heap. Review your heap and stack size requirements for C++ programs to ensure that they are satisfied.

C++ Standard Library

The C++ standard defines the C++ standard library. A few of these platform features are unavailable on the default Xilinx EDK software platform. For example, file I/O is supported in only a few well-defined STDIN/STDOUT streams. Similarly, locale functions, thread-safety and other such features may not be supported. Refer to http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html for a comprehensive documentation of the GNU C++ standard library.

Position Independent Code (Relocatable Code)

The PowerPC compiler supports the `-fPIC` switch to generate position independent code. It also supports the `-mrelocatable` switches to generate a slightly different form of relocatable code. While both these features are supported in the Xilinx compiler, they are not supported by the rest of the libraries and tools, since Xilinx EDK only provides a standalone platform. No loader or debugger can understand relocatable code and perform the correct relocations at runtime. These independent code features are not supported by the Xilinx libraries, startup files, or other tools.

The MicroBlaze compiler does not provide support for `-fPIC`.

Other Switches and Features

Other switches and features might not be supported by the Xilinx EDK compilers and/or platform, such as `-fprofile-arcs`. Some features may also be experimental in nature (as defined by open source GCC) and may produce incorrect code if used inappropriately. Refer to the GCC manual for more information on specific features.

XILINX®

Chapter 11

GNU Debugger (GDB)

This chapter describes the general usage of the Xilinx® GNU debugger (GDB) for MicroBlaze™ and PowerPC™. This chapter contains the following sections:

- Overview
- MicroBlaze GDB Targets
- PowerPC Targets
- Console Mode
- GDB Command Reference

Overview

GDB is a powerful yet flexible tool that provides a unified interface for debugging and 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 following options are the most common in the GNU debugger:

--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 the Processor target, located in `Simulator/Hardware/Virtual Platform`. This action 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 supported by the MicroBlaze GNU Debugger and XMD tools.

Remote debugging is done through XMD. The XMD server program can be started on a host computer with the Simulator target, Hardware target, or Virtual Platform target transparent to mb-gdb. The Cycle-Accurate Instruction Set Simulator (ISS) and the Hardware interface provide powerful debugging tools for verifying a complete MicroBlaze system. The debugger mb-gdb connects to XMD using the GDB Remote Protocol over TCP/IP socket connection.

Simulator Target

The XMD simulator is a cycle-accurate ISS 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 Chapter 12, "Xilinx Microprocessor Debugger (XMD)."

Virtual Platform Target

Virtual Platform is a Cycle-Accurate MicroBlaze fixed Reference design. It supports LMB and External Memory, UARTlite, and GPIO interface.

Refer to Chapter 5, "Virtual Platform Generator (VPgen)" for more information on generating Virtual Platform. Refer to the "Virtual Platform MicroBlaze Target" section of Chapter 12, "Xilinx Microprocessor Debugger (XMD)" for more information on Virtual Platform debugging.

Compiling for Debugging on MicroBlaze Targets

In order to debug a program, you must 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 the Xilinx 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. The debugger mb-gcc adds appropriate information to the executable file, which helps in debugging the code. The debugger mb-gdb provides debugging at source, assembly, and mixed source and assembly.

Note: While initially verifying the functional correctness of a C program, do not use any mb-gcc optimization option like **-O2** or **-O3** as mb-gcc does aggressive code motion optimizations which might make debugging difficult to follow.

Note: For debugging with XMD in hardware mode using XMDStub, you must specify the mb-gcc option **-xl-mode-xmdstub**. Refer to Chapter 12, "Xilinx Microprocessor Debugger (XMD)" 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 Virtex-4, and *Cycle-Accurate PowerPC Instruction Set Simulator*.

To connect to a PowerPC target:

1. Start XMD and connect to the board using the **connect ppc** command as described in Chapter 12, "Xilinx Microprocessor Debugger (XMD)."
2. Select **Run →Connect to target** from GDB.
3. In the GDB target selection dialog box, specify the following:
   - Target: **Remote/TCP**
   - Hostname: **localhost**
   - Port: **1234**
4. Click **OK**.

The compiler powerpc-eabi-gdb attempts to make a connection to XMD. If successful, a message is printed in the shell window where XMD started.

At this point, the compiler is connected to XMD and controls the debugging. The GUI can be used to debug the program and read and write memory and registers.

Console Mode

To start powerpc-eabi-gdb in the console mode, type the following:

```
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
```

The following text displays:

```
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 console 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, select **Help →Help Topics** in the XPS main dialog box or type **`help`** in the console mode.

To open a console window from the XPS main dialog box, select **View →Console**.

For a comprehensive online documentation on using GDB, refer to the GNU website at http://www.gnu.org.

For information about the mb-gdb Insight GUI, refer to the Red Hat Insight webpage at http://sources.redhat.com/insight.

Table 11-1: **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:*** Do not use the `run` command.
`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*.

Table 11-1 describes the commonly used mb-gdb console commands. The equivalent GUI versions can be identified in the mb-gdb GUI window icons. Some of the commands, such as info target and monitor info, might be available only in the console mode.

Chapter 12

Xilinx Microprocessor Debugger (XMD)

The Xilinx® Microprocessor Debugger (XMD) is a tool that facilitates debugging programs and verifying systems using the PowerPC™ 405GP (Virtex™-II Pro & Virtex™-4) or MicroBlaze™ microprocessors. You can use it to debug programs running on a hardware board, Cycle-Accurate Instruction Set Simulator (ISS), or MicroBlaze Cycle-Accurate Virtual Platform (VP) system.

XMD provides a Tool Command Language (Tcl) 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 a complete system.

XMD supports GNU Debugger (GDB) Remote TCP protocol to control debugging of a target. Some graphical debuggers use this interface for debugging, including PowerPC and MicroBlaze GDB (`powerpc-eabi-gdb` and `mb-gdb`) and the Platform Studio Software Development Kit (SDK), EDK's Eclipse-based Software IDE. In either case, the debugger connects to XMD running on the same computer or on a remote computer on the Network.

XMD reads Xilinx Microprocessor Project (XMP), Microprocessor Hardware Specification (MHS), and (Microprocessor Software Specification) (MSS) system files to better understand the hardware system on which the program is debugged. The information is used to perform memory range tests, determine MicroBlaze to Microprocessor Debug Module (MDM) connectivity for faster download speeds, and perform other system actions.

This chapter contains the following sections.

- XMD Usage
- XMD Command Reference
- Connect Command Options
- XMD Internal Tcl Commands

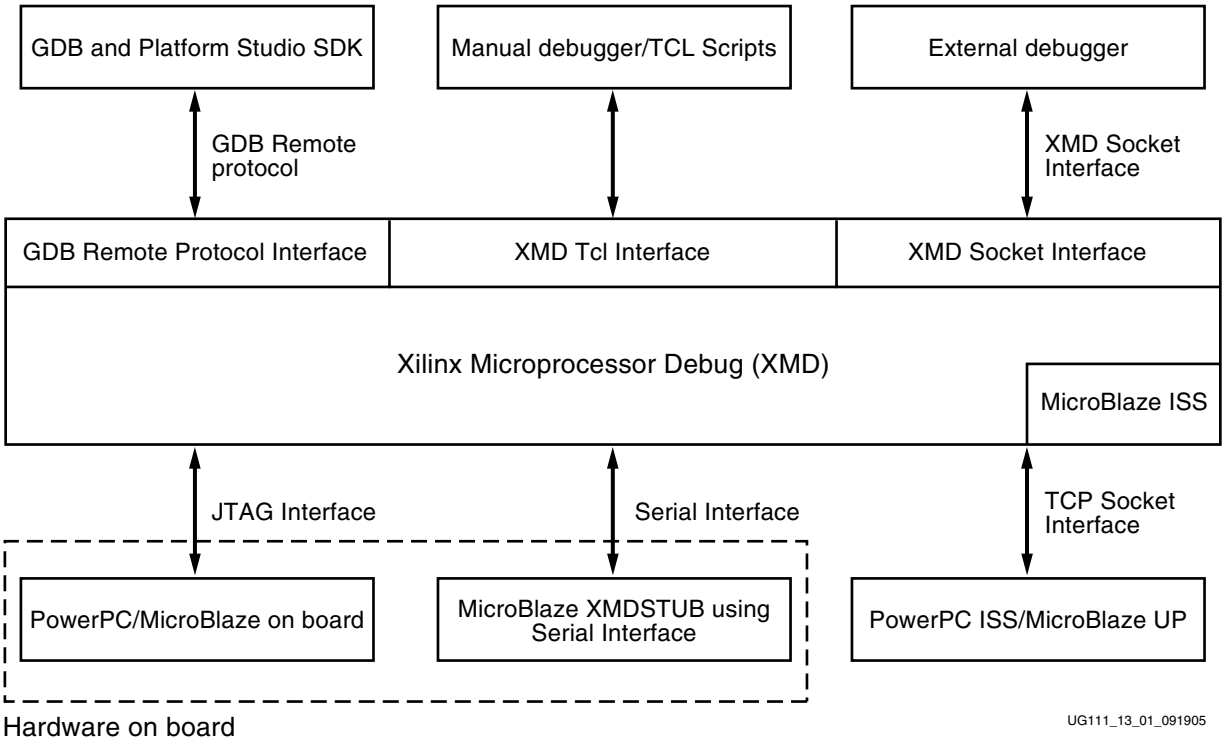

Figure 12-1: **XMD Targets**

XMD Usage

```
xmd [-v] [-h] [-help] [-nx] [-ipcport [portnum]] [-xmp xmpfile]
[-opt optfile] [-tcl {<tcl_file_args}]
```

Table 12-1: **XMD Options**

Option	Command	Description
Help	**-h, -help**	Displays the usage menu and then quits.
Version	**-v**	Displays the version and then quits.
No Initialization file	**-nx**	Does not source `xmd.ini` file on startup.
Port Number	**-ipcport *portnum***	Starts the 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 File	**-xmp *xmpfile***	Specifies the XMP file to load.

Table 12-1: **XMD Options *(Continued)***

Option	Command	Description
Option File	**-opt *connect_option_file***	Specify the option file to use to connect to target. The option file contains the XMD connect command to target.
Tcl File	**-tcl *tclfile [tclarg]***	XMD Tcl script to run. *tclargs* are arguments to the Tcl script. This Tcl file is sourced from XMD. XMD quits after executing the script. No other option can follow **-tcl**.

On startup, XMD does the following:

- If an XMD Tcl script is specified, XMD executes the script and then quits.
- If an XMD Tcl script is *not* specified, XMD starts in *interactive mode*. In this case, XMD does the following:
  - Creates source `${HOME}/.xmdrc` file. You can use this configuration file to form custom Tcl commands using XMD commands.
  - If **-ipcport** option is given, opens XMD Socket server.
  - If **-xmp** option is given, loads system XMP file.
  - If **-opt** option is given, uses Connect option file to connect to processor target.
  - If **-nx** option is not given, source `xmd.ini` file if present in the current directory.
  - Displays the `XMD%` prompt. From the `XMD Tcl` prompt, you can use XMD commands for debugging, as described in the next section, "XMD Command Reference."

XMD Command Reference

XMD User Commands

Table 12-2 displays XMD user commands and options. For a list of special register names for MicroBlaze and PowerPC, refer to "Special Purpose Register Names" on page 144. For `connect` command options, refer to "Connect Command Options" on page 145.

Table 12-2: **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**	Loads XMP, MHS, and MSS system files. XMD reads MHS and MSS system files for the following reasons: • To infer the connectivity of Fast Synchronous Link (FSL) Bus between opb_mdm (Microprocessor Debug Module) module and MicroBlaze. This connectivity is used to download the program and data at a very fast rate. For additional information, search on "fast download" in the *Platform Studio Online Help*. • To infer Instruction and Data memory address maps of the processor. This information is used to verify the program and data downloaded to processor memory.
connect *Target Connect Type Options*	**connect mb mdm** **connect ppc hw**	Connects to *Target*. Valid target types: `mb`, `ppc`, and `mdm`. For additional information, refer to "Connect Command Options" on page 145.
vpconnect *mb*	**vpconnect mb**	Connects to MicroBlaze virtual platform.
vpio	**vpio**	Starts the I/O interfaces (UART and gpio) for the virtual platform system.
targets [*target id*]	**targets** **targets 0**	Lists information about all current targets or changes the current target.
disconnect <*target id*>	**disconnect 0**	Disconnects from the current processor target, closes the corresponding GDB server, and reverts to the previous processor target, if any.

Table 12-2: **XMD User Commands *(Continued)***

command [options]	Example Usage	Description
debugconfig [-step_mode <step type>] [-memory_datawidth_matching [disable | enable]] [-vpoptions <virtual platform options>] [-reset_on_run <Options>]	**debugconfig -step_mode enable_interrupt** **debugconfig -memory_datawidth_matching enable**	Configures the debug session for the target. For additional information, refer to "Configure Debug Session" on page 171.
dow *[-data] <filename> [addr]*	**dow executable.elf** **dow executable.elf 0x400** **dow -data system.dat 0x400**	Downloads the given Executable Linked Format (ELF) or data file (with the `-data` option) onto the memory of the current target. • If no address is provided along with the 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 Position Independent Code (PIC code) and downloaded at the specified address. In addition, Register R20 is set to the start address according to the PIC code semantics.
run	**run**	Runs program from `Program Start Address`.
con *[address]*	**con** **con 0x400**	Continues from current PC or `address`.
stp *[number of instructions]*	**stp** **stp 10**	Steps one or `number` of instructions.
cstp *[number of cycles]*	**cstp** **cstp 10**	Steps one or `number` of cycles. ***Note:*** This is supported only on ISS/VP targets.
rst [-processor]	**rst**	Resets the system. If the `-processor` option is specified, the current processor target is reset.
stop	**stop**	Stops target.
rrd *[reg num]*	**rrd** **rrd r1 (or) rrd R1** **rrd 1**	Reads all registers or reads `reg num` register.
srrd *[reg name]*	**srrd** **srrd pc**	Reads special purpose registers or reads `reg name` register.
rwr *[reg num | reg name] word*	**rwr pc 0x400**	Registers Write.

Table 12-2: **XMD User Commands *(Continued)***

command [options]	Example Usage	Description
mrd [<address> [<num> [w | h | b]]] [<variable>]	**mrd 0x400** **mrd 0x400 10** **mrd 0x400 10 h**	Reads num memory locations starting at address. Defaults to a word (**w**) read. If variable name is specified, reads memory corresponding to global variable in the previously downloaded ELF file.
mrd_var *<variable> [Elf filename]*	**mrd global_var1 executable.elf**	Reads 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**	Writes to num memory locations starting at address. Defaults to a word (**w**) write.
bps *<address|function>* [sw | hw]	**bps 0x400** **bps main hw**	Sets a software or hardware breakpoint at address or start of function. The last downloaded ELF file is used for function lookup. Defaults to software breakpoint.
watch [r | w] *<address> [data]*	**watch r 0x400 0x1234** **watch r 0x40X 0x12X4** **watch r 0b01000000XXXX 0b00010010XXXX0100** **watch r 0x40X**	Sets a 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, it matches any value. ***Note:*** For PowerPC, only absolute values are supported.
bpr *<address|function|bp id|all>*	**bpr 0x400** **bpr main** **bpr all**	Removes Breakpoint/Watchpoint.
bpl	**bpl**	Lists Breakpoints/Watchpoints.

Table 12-2: **XMD User Commands *(Continued)***

command [options]	Example Usage	Description
tracestart *[pc trace filename] [-function_name <func trace filename>]*	**tracestart pctrace.txt** **tracestart pctrace.txt -function_name fntrace.txt** **tracestart**	Starts collecting instruction and function trace information to `filename`. • Trace collection can be stopped and started any time the program runs. • `filename` should be specified only on first tracestart. • `pc trace filename` defaults to `isstrace.out`. • `func trace filename` defaults to `fntrace.out`. ***Note:*** This is supported only on ISS/VP targets.
tracestop *[done]*	**tracestop** **tracestop done**	Stops collecting trace information. Option `done` signifies the end of tracing. ***Note:*** This is supported only on ISS/VP targets.
stats *[filename]*	**stats trace.txt** **stats**	Displays execution statistics for the ISS and VP target. The `filename` is the trace output from trace collection.
profile *[-o GMON output file]*	**profile -o gproff.out**	Writes Profile output file, which can be interpreted by mb-gprof or powerpc-eabi-gprof to generate profiling information. Specify the profile configuration sampling frequency in Hz, Histogram Bin size, and Memory address for collecting profile data. For details about Profiling using XPS, search on "Profiling" in the *Platform Studio Online Help*.
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 [-jtag_uart_server [port no]]	**terminal** **terminal -jtag_uart_server 4321**	JTAG-based hyperterminal to communicate with `opb_mdm` UART interface. The UART interface should be enabled in the opb_mdm. If the `-jtag_uart_server` option is specified, a TCP server is opened at *port_no*. Use any hyperterminal utility to communicate with `opb_mdm` UART interface over TCP sockets. Default value for `port_no` is `4321`.

Table 12-2: **XMD User Commands *(Continued)***

command [options]	Example Usage	Description
read_uart [start | stop] *[TCL Channel ID]*	**read_uart start** **read_uart stop** **read_uart start $channel_id**	Reads from opb_mdm UART interface. O/P can be redirected to file by specifying a Tcl channel ID.
verbose *[level]*	**verbose**	Toggles verbose mode ON/OFF. In verbose mode, XMD prints debug information.
help *[options]*	**help** **help init** **help connect** **help connect mb**	Lists all commands.

Special Purpose Register Names

MicroBlaze Special Purpose Register Names

The following special register names are valid for MicroBlaze processors:

pc	msr	ear	esr
fsr	btr	pvr0	pvr1
pvr2	pvr3	pvr4	pvr5
pvr6	pvr7	pvr8	pvr9
pvr10	pvr11		

For additional information, descriptions, and usage of MicroBlaze special register names, refer to the "Special Purpose Registers" section of the "MicroBlaze Architecture" chapter in the *MicroBlaze Processor Reference Manual.*

Note: When MicroBlaze is debugged in XMDStub mode, only PC and MSR registers are accessible.

PowerPC Special Purpose Register Names

The following special register names are valid for PowerPC processors:

ccr0	f2	f18	iac4	
cr	f3	f19	iccr	sprg5
ctr	f4	f20	icdbdr	sprg6
dac1	f5	f21	lr	sprg7
dac2	f6	f22	msr	srr0
dbcr0	f7	f23	pc	srr1

dbcr1	f8	f24	pid	srr2
dbsr	f9	f25	pit	srr3
dccr	f10	f26	pvr	su0r
dcwr	f11	f27	sgr	tbl
dear	f12	f28	sler	tbu
dvc1	f13	f29	sprg0	tcr
dvc2	f14	f30	sprg1	tsr
esr	f15	iac1	sprg2	usprg0
evpr	f16	iac2	sprg3	xer
f0	f17	iac3	sprg4	zpr
f1				

For additional information about PowerPC special register names, refer to the "Register Set Summary" section of the *PowerPC 405 Processor Block Reference Manual* (included in the `/doc` directory of your EDK installation).

Connect Command Options

XMD can debug programs on different targets (Processor, Virtual Platform, or Peripheral). To communicate with a target, XMD connects to the target. A unique target ID is assigned to each target after connection. When connecting to a Processor or Virtual Platform target, `gdbserver` starts, enabling communication with GDB or Platform Studio SDK.

Usage

`connect [mb | ppc | mdm] <Connection_Type> [Options]`

Table 12-3: **Connect Command Options**

Option	Description
`ppc`	Connects to PowerPC Processor
`mb`	Connects to MicroBlaze Processor
`mdm`	Connects 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 Virtex-4 devices contain one or two PowerPC405 processor cores. XMD can connect to these PowerPC targets over a JTAG connection on the board. XMD also communicates over a TCP socket interface to an 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.

Note: XMD does not support Virtual Addressing. Debugging is only supported for Programs running in Real Mode.

PowerPC Hardware Connection

When connecting to a PowerPC hardware target, XMD automatically detects the JTAG chain, and the PowerPC processors, and connects to the first processor. You 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

The following options allow you to specify the Xilinx JTAG cable used to connect to target.

Table 12-4: **JTAG Cable Options**

Option	Description
type *cable_type*	Valid cable types are: • xilinx_parallel3 • xilinx_parallel4 • xilinx_platformusb • 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,usb2**
fname *filename*	The filename for creating the SVF file
frequency *<cable speed in Hz>*	Specify the Cable Clock Speed. Valid Cables speeds are: • For Parallel 4: 5000000 (default), 2500000, 200000 • For Platform USB: 24000000, 12000000, 6000000 (default), 3000000, 1500000, 750000

JTAG Chain Options

The following options allow you to specify device information of Non-Xilinx devices in the JTAG chain. Refer to "Example Showing Special JTAG Chain Setup for Non-Xilinx Devices" on page 152.

Table 12-5: **JTAG Chain Options**

Option	Description
devicenr *<device position>*	The position of the device in the JTAG chain. The device position number starts from '1'.
partname *devicename*	The name of the device.
irlength *<length of the JTAG Instruction Register>*	The length of the IR register of the device. This information can be found in the device BSDL file.
idcode *<device idcode>*	JTAG ID code of the device.

PowerPC (PPC405) Options

The following options allow you to specify the FPGA device to debug and the processor number in the device. You can also map special PowerPC features such as ISOCM, Caches, TLB, and DCR registers to unused memory addresses, and then access them from the debugger 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.

Table 12-6: **PowerPC Options**

Option	Description
devicenr *<PowerPC device position>*	The position in the JTAG chain of the Virtex-II Pro or Virtex-4 device containing the PowerPC.
cpunr *<CPU Number>*	The PowerPC Processor number to be debugged in a Virtex-II Pro or Virtex-4 containing multiple PowerPC processors. The Processor number starts from '1'.
romemstartadr *<ROM start address>*	The start address of Read-Only Memory. This can be used to specify flash memory range. XMD sets hardware breakpoints instead of software breakpoints.
rommemsize *<ROM Size>*	The size of Read-Only Memory (ROM).
isocmstartadr *<ISOCM start address>*	The start address for the ISOCM.
isocmsize *<ISOCM size>*	The size of the ISBRAM memory connected to the ISOCM interface.
isocmdcrstartadr *<ISOCM DCR address>*	The DCR address corresponding to the ISOCM interface specified using the C_ISOCM_DCR_BASEADDR parameter on PowerPC.
icachestartadr *<I-Cache start address>*	The start address for reading or writing the instruction cache contents.

Table 12-6: **PowerPC Options *(Continued)***

Option	Description
dcachestartadr *<D-Cache start address>*	The start address for reading or writing the data cache contents.
itagstartadr *<I-Cache start address>*	The start address for reading or writing the instruction cache tags.
dtagstartadr *<D-Cache start address>*	The start address for reading or writing the data cache tags.
tlbstartadr *<TLB start address>*	The start address for reading and writing the Translation Look-aside Buffer.
dcrstartadr *<DCR start address>*	The start address for reading and writing the Device Control Registers. Using this option, the entire DCR address space (210 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.

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 12-2. Refer to the *PowerPC 405 Processor Block Reference Manual* for more information.

Note: The `jtagppc_cntlr` core in EDK helps in setting up this connection.

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 IO pins, then XMD can directly connect to the PowerPC for debugging. Refer to the *PowerPC 405 Processor Block Reference Manual* for more information about this debug setup.

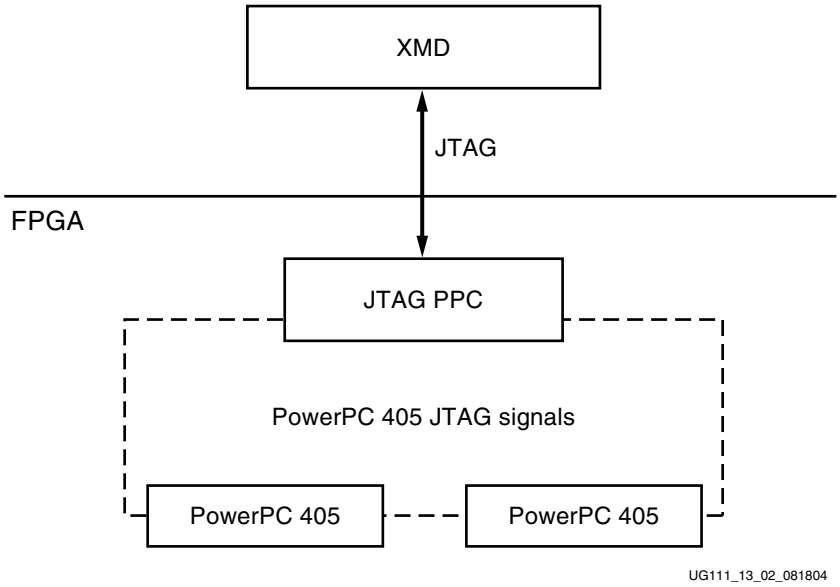

Figure 12-2: **PowerPC Target**

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, powerpc-eabi-gdb is connected to XMD using the GDB remote target. Refer to Chapter 11, "GNU Debugger (GDB)," 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%
```

Example with a Program Running in ISOCM Memory and Accessing DCR Registers

This example demonstrates a simple debug session with a program running on ISOCM memory of the PowerPC target. The ISOCM address parameters can be specified during the **connect** command. If the XMP file is loaded, XMD infers the ISOCM address parameters of the system from the MHS file.

Note: In a Virtex-4 device, ISOCM memory is readable. This enables better debugging of a program running from ISOCM memory. In a Virtex-II Pro device, ISOCM memory is not readable.

```
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%
```

Example Showing Special JTAG Chain Setup for Non-Xilinx Devices

This example demonstrates the use of the **-configdevice** option to specify the JTAG chain on the board, if XMD is unable to automatically detect the JTAG chain. Automatic detection 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 such as `partname` and `idcode` are optional.

The following describes the options used 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 second device is specified. Partname is inferred from the ID code since XMD knows about the XC2VP4 device.
- The **debugdevice** option explicitly specifies to XMD that the FPGA device of interest is the second device in the JTAG chain. In the Virtex-II Pro, it is also explicitly specified that the connection is for the first PowerPC processor, if there is more than one.

```
XMD% connect ppc hw -cable type xilinx_parallel port LPT1 -configdevice
devicenr 1 partname PROM_XC18V04 irlength 8 idcode 0x05026093 -
configdevice devicenr 2 partname XC2VP4 irlength 10 idcode 0x0123e093 -
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`. To add support for a device in XMD, do the following:

1. Edit the `devicetable.lst` file. Add the Device ID Code, Instruction Register Length, and Name information.
2. If XMD is open, close it and restart. XMD automatically discovers the Device in the JTAG chain.

PowerPC Simulator Target

XMD can connect to one or more PowerPC 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. You can also use **`connect ppc sim`** to connect to a 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 a default configuration. The ISS executable file is located in the `${XILINX_EDK}/third_party/bin/<platform>/` directory. The configuration file used is `${XILINX_EDK}/third_party/data/iss405.icf`. You can run ISS with different configuration options and XMD can connect to the ISS target. Refer to the *IBM Instruction Set Simulator User Guide* for more details. The following are the default configurations for ISS.

- Two local memory banks, as displayed in Table 12-7
- Connect to Debugger (XMD)
- Debugger Port at 6470..6490
- Data Cache size of 8 K
- Instruction Cache size of 16 K
- Non-Deterministic Multiply cycles
- Processor Clock Period and Timer Clock Period of 5 ns (200 MHz)

Table 12-7: **Local Memory Banks**

Name	Start Address	Length	Speed
Mem0	0x0	0x80000	0
Mem1	0xfff80000	0x80000	0

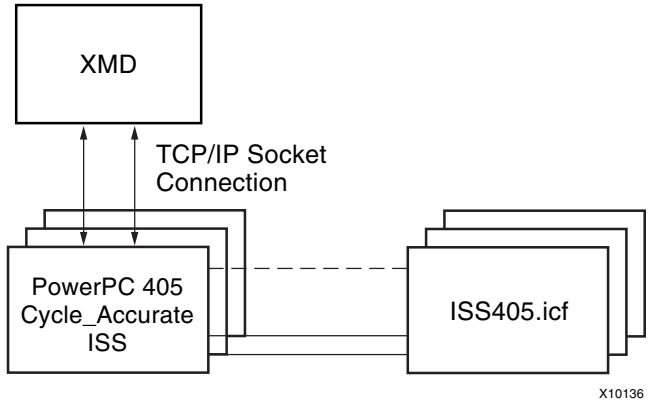

Figure 12-3: **PowerPC ISS Target**

Usage

`connect ppc sim [-icf <Configuration File>] [-ipcport IP:port]`

Table 12-8: **PowerPC ISS Options**

Option	Description
`-icf`	Uses the given ISS Configuration file instead of the default configuration file. You can customize the PowerPC ISS features such as cache size, memory address map, and memory latency.
`-ipcport`	Specifies the IP address and debug port of PowerPC ISS that you started. XMD does not spawn a ISS, but connects to the ISS that you started.

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
```

Advanced PowerPC Debugging Tips

Support for Running Programs from ISOCM and ICACHE

There are restrictions on debugging programs from PowerPC ISOCM memory and instruction caches (ICACHEs). One such restriction is that you cannot use software breakpoints. In such cases, XMD can automatically set hardware breakpoints if the address ranges for the ISOCM or ICACHEs are provided as options to the `connect` command in XMD. In this case of ICACHE, this is only necessary if you try to run programs completely from the ICACHE by locking its contents in ICACHE.

For more information, refer to the Xilinx Platform Studio online help. Accessing DCR Registers, TLB, ISOCM, Instruction, and Data Caches

The previously mentioned special features of the PowerPC can be accessed from XMD by specifying the appropriate options to the `connect` command in the XMD console.

Debugging Setup for Third-Party Debug Tools

In order to use third-party debug tools such as Wind River SingleStep and Green Hills Multi, Xilinx recommends that you bring the JTAG signals of the PowerPC out of the FPGA as User IO to appropriate debug connectors on the hardware board. Apart from the JTAG signals, TCK, TMS, TDI, and TDO, you must also bring the DBGC405DEBUGHALT and C405JTGTDOEN signals out of the FPGA as User IO. In the case of multiple PowerPCs, Xilinx recommends that you chain the PowerPC JTAG signals inside the FPGA. For more information about connecting the PowerPC JTAG port to FPGA User IO, refer to the PPC405 JTAG port section of the *PowerPC 405 Processor Block Reference Manual.*

Note: You must NOT use the JTAGPPC module while bringing the PowerPC JTAG signals out as User IO.

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 a ROM monitor such as XMDStub through JTAG or Serial interface. You 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`** 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.

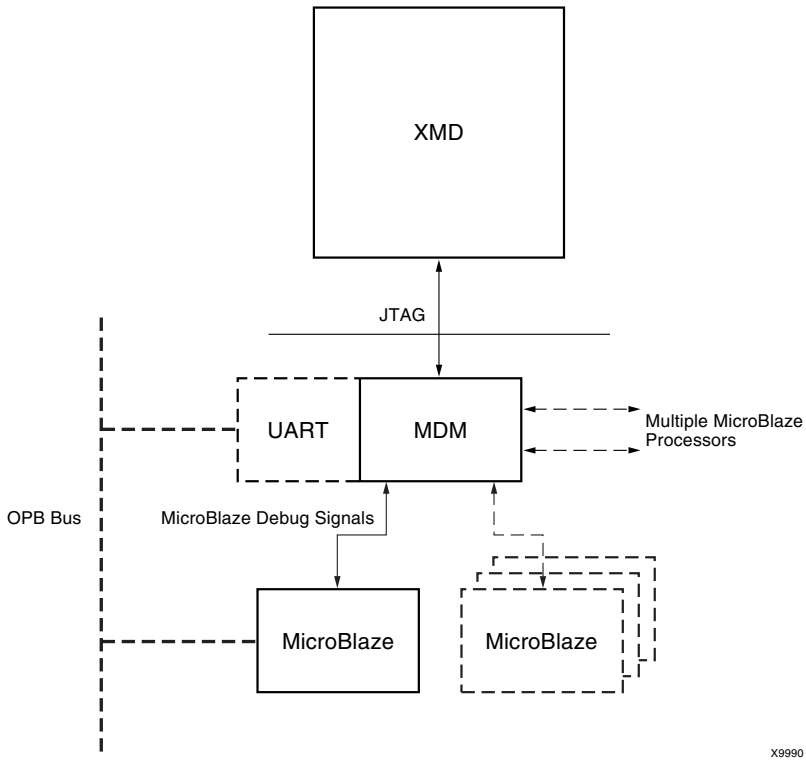

Figure 12-4: **MicroBlaze MDM Target**

When no option is specified to the `connect mb mdm`, XMD automatically detects the JTAG cable and chain and the FPGA device containing the MicroBlaze-MDM system. If XMD is unable to automatically detect the JTAG chain or the FPGA device, you 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

For JTAG cable and chain option descriptions, refer to Table 12-4, JTAG Cable Options on page 146, and Table 12-5, JTAG Chain Options on page 147, respectively.

MicroBlaze Options

MicroBlaze options are described in Table 12-9.

Table 12-9: **MicroBlaze Options**

Option	Description
devicenr *<MicroBlaze device position>*	The position in the JTAG chain of the FPGA device containing the MicroBlaze processor.
cpunr *<CPU Number>*	The specific MicroBlaze processor number to be debugged in an FPGA containing multiple MicroBlaze processors connected to opb_mdm. The processor number starts from 1.
romemstartadr *<ROM start address>*	The start address of Read-Only Memory. Use this to specify flash memory range. XMD sets hardware breakpoints instead of software breakpoints.
romemsize *<ROM Size>*	The size of Read-Only Memory.

MicroBlaze FSL Options

Table Table 12-10 describes MicroBlaze FSL options.

Table 12-10: **MicroBlaze FSL Options**

Option	Description
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 and hardware debug control functions like stopping and stepping, MicroBlaze's hardware debug port must be connected to the MDM, 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 = 4.00.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, you must set the C_USE_UART parameter while instantiating the opb_mdm in a system. To print program STDOUT onto the XMD console, set C_UART_WIDTH to **8**.

3. In order to perform fast download on a MicroBlaze target, connect the opb_mdm Master FSL Bus Interface to the MicroBlaze Slave FSL Bus Interface. The following MHS snippet demonstrates the necessary debug port connection 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
```

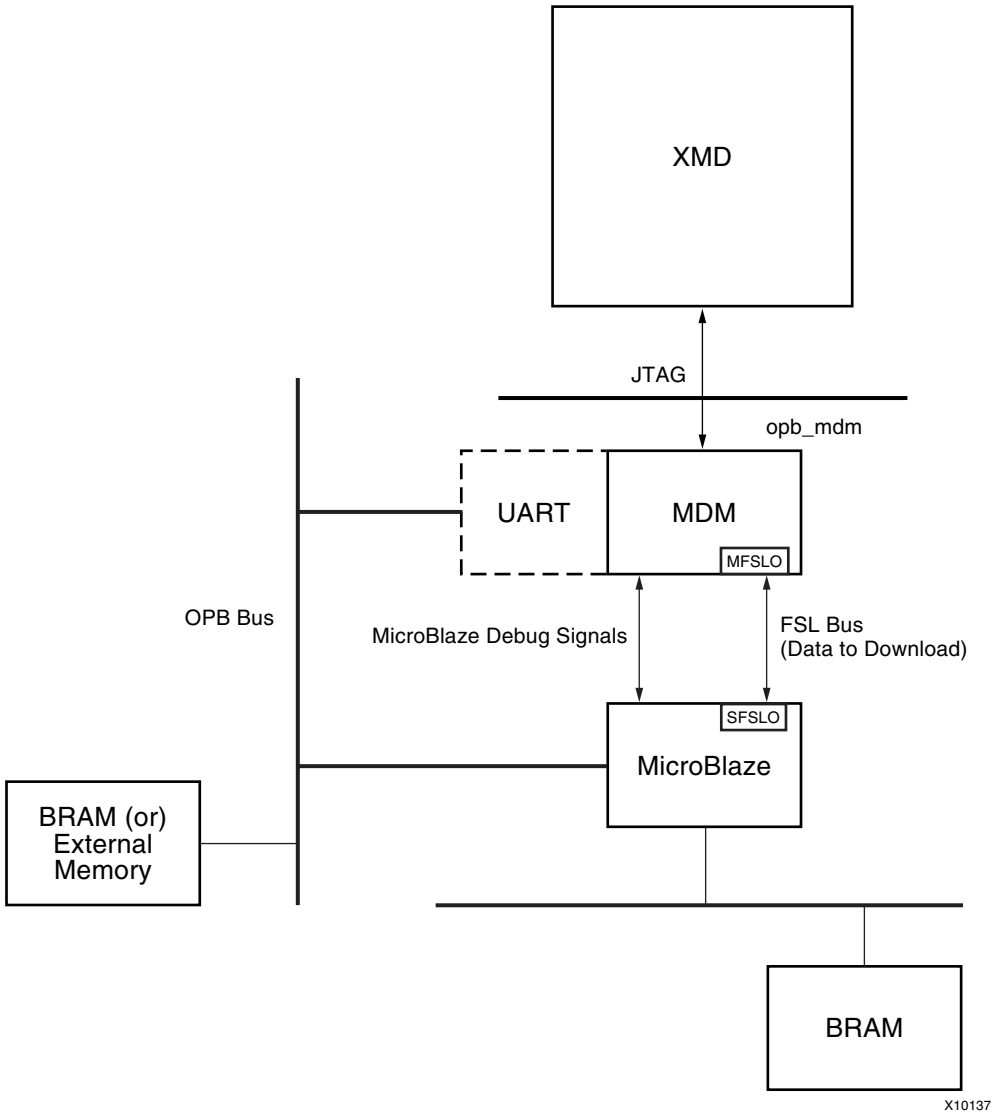

Figure 12-5: **MicroBlaze-MDM Connection for Fast Download**

When the MHS file is loaded, XMD automatically infers this connectivity. When the size of the program or data is greater than 256 bytes, fast download is automatically used. For more information about fast download on MicroBlaze, search on "fast download" in the *Platform Studio Online Help*.

Note: Unlike the MicroBlaze stub target, programs should be compiled in executable mode and NOT in XMDStub mode while using the MDM target. Consequently, you do *not* need to 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, mb-gdb connects to XMD using the GDB remote target. Refer to Chapter 11, "GNU Debugger (GDB)," 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.............................4.00.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
XMD% con
Processor started. Type "stop" to stop processor
RUNNING> rrd pc
pc : 0x000000f4 <--- With the MDM, the current PC of MicroBlaze can be
                     read while the program is running
RUNNING> rrd pc
pc : 0x00000110 <--- Note: the PC is constantly changing, as the
                      program is running
RUNNING> stop
Processor stopped at PC: 0x0000010C
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

This example demonstrates a debug session with two MicroBlaze Processors connected through MDM. In this example, the opb_mdm has the UART interface enabled. This interface can be as STDIN/OUT for the processor. This example demonstrates a method to communicate to the MDM UART interface.

```
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............................4.00.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 Microblaze 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............................4.00.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 Microblaze 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, a program running on the board polls and waits 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 stops as soon as there is load from address 0xC000. In the MicroBlaze configuration below, there are four 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.............................4.00.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 Microblaze 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 0xC000
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 the **-configdevice** option to specify the JTAG chain on the board, in case XMD is unable to automatically detect 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 ID code are optional.

The following describes the options used in the example below.

- The 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 ID code of the second device is specified.
- The debugdevice option explicitly specifies to XMD that the FPGA device of interest is the second 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.............................4.00.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 Microblaze 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) and start a GDB server for the target. XMD connects to XMDStub through a JTAG or Serial interface. The default option connects using a 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

For JTAG cable and chain option descriptions, refer to Table 12-4, JTAG Cable Options on page 146 and Table 12-5, JTAG Chain Options on page 147, respectively.

MicroBlaze Options

Table 12-11: **MicroBlaze Options**

Option	Description
devicenr *<MicroBlaze device position>*	The position in the JTAG chain of the FPGA device containing MicroBlaze.

MicroBlaze Stub-Serial Target Options

Usage

connect mb stub -comm serial *<Serial Communication options>*

Serial Communication Options

The following options can be used to specify the MicroBlaze stub-serial target.

Table 12-12: **MicroBlaze Stub-Serial Target Options**

Option	Description
-port *<serial port>*	Specifies 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>*	Specifies the serial port baud rate in bits per second (bps). The default value is *19200* bps.
-timeout *<timeout in secs>*	Timeout period while waiting for a reply from XMDStub for XMD commands.

Note: User Program outputs: if the program has any I/O functions such as `print() or putnum()` that write output onto the UART or JTAG UART, it is printed on the console/terminal where XMD was started. Refer to Chapter 4, "Library Generator (Libgen)," for more information about libraries and I/O functions.

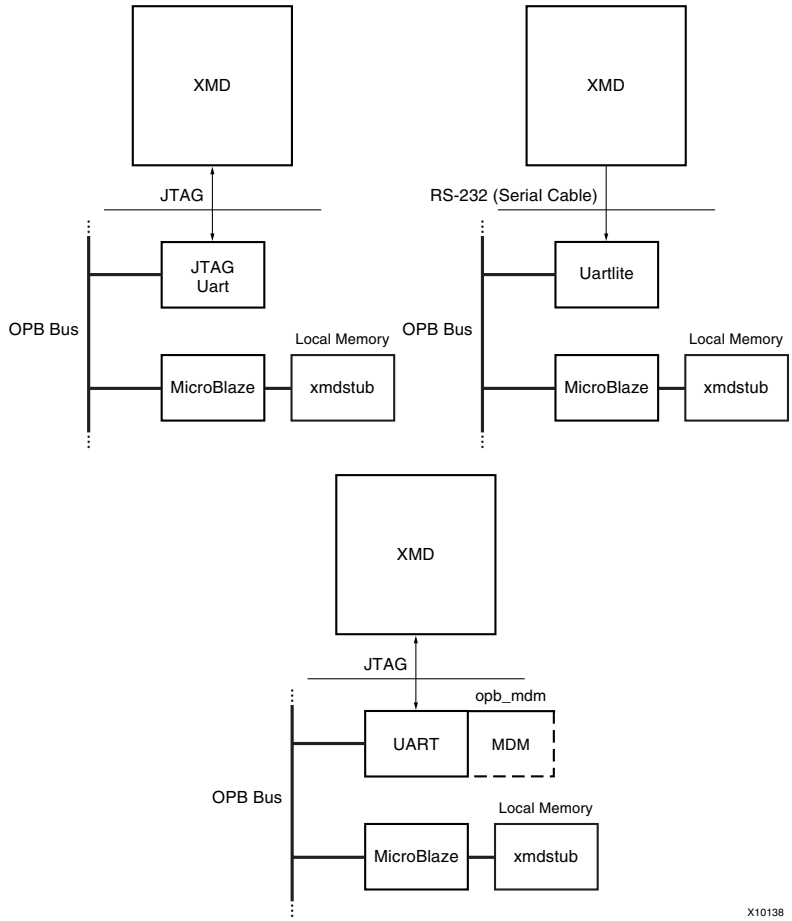

Figure 12-6: **MicroBlaze Stub Target with JTAG UART and UARTlite**

Stub Target Requirements

To debug programs on the hardware board using XMD, the following requirements must be met.

- XMD uses a JTAG or serial connection to communicate with XMDStub on the board. Therefore, an opb_mdm or a UART designated as `XMDSTUB_PERIPHERAL` in the MSS file is necessary on the target MicroBlaze system.

  Platform Generator can create a system that includes a opb_mdm or a UART, if specified in its MHS file. The JTAG cables supported with the XMDStub mode are Xilinx Parallel Cable and Platform USB Cable.

- XMDStub on the board uses the opb_mdm or UART to communicate with the host computer. Therefore, it must be configured to use the opb_mdm or UART in the MicroBlaze system.

  The Library Generator (Libgen) can configure the XMDStub to use the `XMDSTUB_PERIPHERAL` in the system. Libgen generates an XMDStub configured for the `XMDSTUB_PERIPHERAL` and puts it in `code/xmdstub.elf` as specified by the `XMDStub` attribute in the MSS file. For more information, refer to Chapter 4, "Library Generator (Libgen)."

- The 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 DEFAULT_INIT.

- For any program that must be downloaded on the board for debugging, the program start address must be higher than 0x800 and the program must 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 such as **`-O2`** or **`-O3`**, as mb-gcc performs aggressive code motion optimizations which might make debugging difficult to follow.

MicroBlaze Simulator Target

You can use mb-gdb and XMD to debug programs on the cycle-accurate simulator built in to XMD.

Usage

```
connect mb sim [-memsize <size>]
```

MicroBlaze Simulator Option

Table 12-13: **MicroBlaze Simulator Option**

Option	Description
memsize *<size>*	The width of the memory address bus allocated in the simulator. Programs can access the memory range from 0 to $(2^{size})-1$. The default memory size is 64 kB.

Simulator Target Requirements

To debug programs on the Cycle-Accurate Instruction Set Simulator using XMD, you must compile programs for debugging and link them 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. The explicit option is **`-xl-mode-executable`**.

The program memory size must not exceed 64 kB and must begin at address 0. The program must be stored in the first 64 kB of memory.

Note: Currently, XMD with a simulator target does not support the simulation of OPB peripherals.

MDM Peripheral Target

You can connect to the `opb_mdm` peripheral and use the UART interface for debugging and collecting information from the system.

Usage

```
connect mdm <-uart>
```

MDM Target Requirements

To use the UART functionality in the MDM target, you must set the `C_USE_UART` parameter while instantiating the `opb_mdm` in a system. To print program STDOUT onto the XMD console, set `C_UART_WIDTH` to **`8`**.

UART input can also be provided from the host to the program running on MicroBlaze using the **`xuart w <byte>`** command. You can use the **`terminal`** command to open a hyperteminal-like interface to read and write from the UART interface. The **`read_uart`** command provides interface to write to STDIO or to file.

Virtual Platform MicroBlaze Target

You can connect to the MicroBlaze VP target for debugging. VP is a cycle-accurate model of a MicroBlaze system used for accurately debugging, profiling, and tracing programs. XMD opens VP if the executable is present in the `<system>/virtualplatform` directory and communicates over TCP socket interface.

Refer to Chapter 5, "Virtual Platform Generator (VPgen)" for more information on generating Virtual Platform. For more information, search on "debugging" and "profiling" in the *Platform Studio Online Help*.

Usage

```
vpconnect mb
```

Configure Debug Session

Configure the debug session for a target using the **debugconfig** command. You can configure the behavior of instruction stepping and memory access method of the debugger.

Usage

```
debugconfig [-step_mode [disable_interrupt | enable_interrupt]] [-memory_datawidth_matching [disable | enable]] [-vpoptions <virtual platform options>] [-reset_on_run <system | processor> <enable | disable>]
```

Table 12-14: **Debug Config Options**

Option	Description
No Option	Lists the current debug configuration for the current session.
-step_mode	Configures how XMD handles Instruction Stepping. There are two modes: • *disable_interrupt* – This is the default mode. The interrupts are disabled during Step. • *enable_interrupt* – The interrupts are enabled during Step. If an interrupt occurs during Step, the interrupt is handled by the registered interrupt handler of the program.
-memory_datawidth_matching	Configures how XMD handles Memory Read/Write. By default, the data width matching is enabled. All data width (byte, half and word) accesses are handled using the appropriate data width access method. This method is especially useful for memory controllers and flash memory, where the Data Width access should be strictly followed. When data width matching is disabled, XMD uses the best possible method, such as word access.

Table 12-14: **Debug Config Options *(Continued)***

Option	Description
-vpoptions [bus_cycle_accuracy <enable | disable>]	Configures the virtual platform. The virtual platform option `bus_cycle_accuracy` is enabled by default. For faster program execution disable `bus_cycle_accuracy`. For more information refer to Chapter 5, "Virtual Platform Generator (VPgen)."
-reset_on_run <system | processor> <enable | disable>	Configures how XMD handles Reset on program execution. A reset brings the system to a known consistent state for program execution. This ensures correct program execution without any side effects from previous program run. By default, XMD performs *System* reset on run (on program download or program run). • To enable different reset types, specify: **debugconfig -reset_on_run processor enable** **debugconfig -reset_on_run system enable** • To enable different reset, specify: **debugconfig -reset_on_run disable** **Note:** When debugging a multiple-processor design, configure XMD to perform *Processor* reset.

Configuring Instruction Step

XMD supports two Instruction Step modes. You can use the **debugconfig** command to select between the modes. The two modes are:

- Instruction step with Interrupts disabled

  This is the default mode. In this mode the interrupts are disabled.

- Instruction step with Interrupts enabled

  In this mode the interrupts are enabled during step operation. XMD sets a hardware breakpoint at the next instruction and executes the processor. If an interrupt occurs, it is handled by the registered interrupt handler. The program stops at the next instruction.

Note: The instruction memory of the program should be connected to the processor d-side interface.

```
.XMD% debugconfig
Debug Configuration for Target 0
---------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Disabled

XMD% debugconfig -step_mode enable_interrupt
XMD% debugconfig
Debug Configuration for Target 0
---------------------------------
Step Mode..................... Interrupt Enabled
Memory Data Width Matching... Disabled
```

Configuring Memory Access

XMD supports handling different memory data width accesses. The supported data widths are Word (32 bits), Half Word (16 bits), and Byte (8 bits). By default, XMD uses appropriate data width accesses when performing memory read/write operations. You can use the **debugconfig** command to configure XMD to match the data width of memory operation. This is usually necessary for accessing Flash devices of different data widths.

```
XMD% debugconfig
Debug Configuration for Target 0
---------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Enabled

XMD% debugconfig -memory_datawidth_matching disable
XMD% debugconfig
Debug Configuration for Target 0
---------------------------------
Step Mode..................... Interrupt Disabled
Memory Data Width Matching... Disabled
```

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 you can list them from XMD by typing **x?**. It is recommended to use the Tcl wrappers for these internal commands as described in Figure 12-1 on page 138. The Tcl wrappers print the output of most of these commands and provide more options. While the Tcl wrappers are backwards-compatible, these **x<*name*>** commands might be deprecated in a future EDK release.

Program Initialization Options

Table 12-15: **Program Initialization Option**

Option	Description
xload_sysfile [xmp | mhs | mss] *<XMP|MHS|MSS filename>*	Loads the XMP, MHS, or MSS file.
xrut *[Session ID]*	Authenticates the XMD session when communicating over XMD sockets interface. The session ID is first assigned and subsequent call returns the session ID.
xconnect *<target> <connect type> [options]*	Connects to Processor or Peripheral target. Valid Target types are mb, ppc, and mdm. Refer to "Connect Command Options" on page 145 for more information on options.
xvpconnect *mb*	Connects to the MicroBlaze VP Target.
xdisconnect *<target id>*	Disconnects from the target.

Table 12-15: **Program Initialization Option *(Continued)***

Option	Description
xtargets [*<target id>*]	Prints the target ID and target type of all current targets or a specific target.
xdebugconfig *<target id>* [-step_mode <Step Type>] [-memory_datawidth_matching [disable | enable]] [-vpoptions <virtual platform options>] [-reset_on_run <system | processor> <enable | disable>]	Configures the Debug session for the target. For additional information, refer to "Configure Debug Session" on page 171.

Register/Memory Options

Table 12-16: **Register/Memory Options**

Option	Description
xrmem <target id> [<address> [num of bytes|half|word] [b | h | w]] [-var <Global Variable Name>]	Reads *num* of memory locations from memory address *addr*. Defaults to byte (**b**) read. A list of Data values is returned. The Data type depends on the Data-Width of Memory access.
xwmem *<target id> addr [num]* [b | h | w] *value*	Writes *num* data *value* from the specified memory *addr*. Defaults to byte (**b**) write.
xrreg *<target id>* [*reg*]	Reads all registers or only register number *reg*.
xwreg *<target id> reg value*	Writes a 32-bit *value* into register number *reg*.
xdownload *<target id>* [-data] *filename* [*addr*]	Downloads the given ELF or data file, using the **-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. Otherwise, it is treated as Position Independent Code (PIC code) and downloaded at the specified address and Register R20 is set to the start address according to the PIC code semantics. NO Bounds checking is done by XMD, except preventing writes into the XMDStub area (address 0x0 to 0x800).
xdisassemble *inst*	Disassembles and displays one 32-bit instruction.

Program Control Options

Table 12-17: **Program Control Options**

Option	Description
xcontinue ***<target id>*** **[*addr*] *[-quit]***	Continues execution from the current PC or from the optional address argument.
xstop ***<target id>***	Stops the Program execution.
xcycle_step ***<target id> [cycles]***	Cycle steps through one clock cycle of PowerPC ISS. If *cycles* is specified, then step *cycles* number of clock cycles.[a]
xstep ***<target id>***	Single steps one MicroBlaze instruction. If the PC is at an IMM instruction, the next instruction also runs. 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*]**	Resets target. Optionally, provide target-specific reset types such as the signals mentioned in Table 12-18 on page 175.
xbreakpoint ***<target id> [addr | function name]*** **[sw | hw]**	Sets 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*]**	Sets read/write watchpoints at a given *address* and check for *value*. If *value* is not specified, watchpoints match any value. The address and value can be specified in hex or binary format.
xremove ***<target id> [addr | function name | bp id |*** **all]**	Removes one or more breakpoints/watchpoints.
xlist ***<target id>***	Lists all of the breakpoint addresses.

a. This command is only for Simulator targets.

Table 12-18: **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 address 0x18.
Non-maskable Break (0x10)	Similar to the Break signal, but works even while the BIP flag is already set. Refer the *MicroBlaze Processor Reference Manual* for more information about the BIP flag.

Table 12-18: **XMD MicroBlaze Hardware Target Signals *(Continued)***

Signal Name (Value)	Description
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 Options

Table 12-19: **Program Trace/Profile Options**

Option	Description
`xstats <target id> [options]`	Displays the simulation statistics for the current session. Use the **`reset`** option to reset the simulation statistics.[a]
`xtracestart <target id>`	Starts collecting trace information.
`xtracestop <target id>`	Stops collecting trace information.[a]
`xprofile <target id> [-o <GMON Output File>]` **`xprofile <target id> -config [sampling_freq_hw <value>] [binsize <value>] [profile_mem <start addr>]`**	Generates profile output that can be read by mb-gprof or `powerpc-eabi-gprof`. Specify the profile configuration sampling frequency in Hz, Histogram Bin size, and Memory address for collecting profile data.

a. This command is only for ISS/Virtual Platform targets.

Miscellaneous Commands

Table 12-20: **Miscellaneous Commands**

Command	Description
`xuart [r | w | s] [<data>]`	Performs one of three UART operations on the MDM UART if it is enabled. This command is valid only for the MDM target. **`xuart <r>`** reads byte from the MDM UART. **`xuart <w> <data>`** writes byte onto the MDM UART. **`xuart <s>`** reads the status of MDM UART.
`xforce_use_fsl_dow <target id>`	Forces XMD to use FSL-based fast download. Use this command when the cable type is `xilinx_svffile` and when reading from a target is not possible, especially when the System ACE™ file is generated.

Table 12-20: **Miscellaneous Commands *(Continued)***

Command	Description
xverbose	Toggles verbose mode on and off. Dumps debugging information from XMD.
xhelp	Lists the XMD commands.

Chapter 13

System ACE File Generator (GenACE)

This chapter describes the steps to generate Xilinx® System ACE™ configuration files from an FPGA bitstream and Executable Linked Format (ELF)/data files. The ACE file generated can be used to configure the FPGA, initialize BRAM, initialize external memory with valid program or data, and bootup the processor in a production system. EDK provides a Tool Command Language (Tcl) script, `genace.tcl`, which uses Xilinx Microprocessor Debug (XMD) commands to generate ACE files. ACE files can be generated for PowerPC™ and MicroBlaze™ with Microprocessor Debug Module (MDM) systems.

This chapter contains the following sections:

- Assumptions
- Tool Requirements
- GenACE Features
- GenACE Model
- The Genace.tcl Script
- Generating ACE Files
- Related Information

Assumptions

This chapter assumes that you:

- Are familiar with debugging programs using XMD and with using XMD commands.
- Are familiar with general hardware and software system models in EDK.
- Have a basic understanding of Tcl scripts.

Tool Requirements

Generating an ACE file requires the following tools:

- `genace.tcl`
- `xmd`
- `iMPACT` (from ISE)

GenACE Features

GenACE has the following features:

- Supports PowerPC and MicroBlaze with MDM targets
- Generates ACE files from hardware (Bitstream) and software (ELF/data) files.
- Initializes external memories on PowerPC and MicroBlaze systems.
- Supports Multi-Processor systems.
- Supports Single/Multiple FPGA device systems.

GenACE Model

The System ACE files generated support the System ACE CF family of configuration solutions. System ACE CF configures devices using Boundary-Scan (JTAG) instructions and a Boundary-Scan Chain. System ACE CF is a two-chip solution that requires the System ACE CF controller, and either a CompactFlash card or one-inch Microdrive disk drive technology as the storage medium. The System ACE file is generated from a Serial Vector Format (SVF) file. An SVF file is a text file containing both programming instructions and configuration data to perform JTAG operations.

XMD and iMPACT generate SVF files for software and hardware system files respectively. The set of JTAG instructions and data used to communicate with the JTAG chain on board is an SVF file. It includes the instructions and data to perform operations such as configuring FPGA using iMPACT, connecting to the processor target, downloading the program, and running the program from XMD are captured in an SVF file format. The SVF file is then converted to an ACE file and written to the storage medium. These operations are performed by the System ACE controller to achieve the determined operation.

The following is the sequence of operations using iMPACT and XMD for a simple hardware and software configuration that gets translated into an ACE file.

1. Download the bitstream using iMPACT. The bitstream, `download.bit`, contains system configuration and bootloop code.
2. Bring the device out of reset, causing the Done pin to go high. This starts the Processor system.
3. Connect to the Processor using XMD.
4. Download multiple data files to BRAM or External memory.
5. Download multiple executable files to BRAM or External memory. The PC points to the start location of the last downloaded ELF file.
6. Continue execution from the PC instruction address.

The flow for generating System ACE files is bit →svf, elf →svf, binary data →svf and svf →ace file. The `genace.tcl` script allows the following operations to perform.

The Genace.tcl Script

Syntax

```
xmd -tcl genace.tcl [-opt <genace_options_file>]|[-jprog]
  [-target <target_type>] [-hw <bitstream_file>] [-elf <Elf_Files>]
  [-data <Data_files>] [-board <board_type>] -ace <ACE_file>
```

Table 13-1: **genace.tcl Script Command Options**

Options	Default	Description
`-opt <genace_options_file>`	none	GenACE options are read from the options file.
`-jprog`	false	Clear the existing FPGA configuration. This option should not be specified if performing runtime configuration.

`-target <target_type>`	ppc_hw	Target to use in the system for downloading ELF/Data file. Target types are:
		`ppc_hw`
		To connect to a ppc405 system
		`mdm`
		To connect to a MicroBlaze system. This assumes the presence of `opb_mdm` in the system.

`-hw <bitstream_file>`	none	The bitstream file for the system. If an SVF file is specified, it is used.
`-elf <list_of_Elf_Files>`	none	List of ELF files to download. If an SVF file is specified, it is used.
`-data <data_file> <load_address>`	none	List of Data/Binary file and its load address. The load address can be in decimal or hex format (0x prefix needed). If an SVF file is specified, it is used.
`-board <board_type>`	none	This identifies the JTAG chain on the board (Devices, IR length, Debug device, and so on). The options are given with respect to the System ACE controller. The script contains the options for some pre-defined boards. Board type options are: `user` The user specifies the `-configdevice` and `-debugdevice` option in the Options file. Refer to the `genace.opt` file for details. For Supported board type refer to the section "Supported Target Boards," below.
`-ace <ACE_file>`	none	The output ACE file. The file prefix should not match any of input files (bitstream, elf, data files) prefix.

The options can be specified in an options file and passed to the GenACE script. The options syntax is described in Table 13-2.

Table 13-2: **Genace File Options**

Options	Default	Description
`# <Some Text>`	none	The line starting with # is treated as a comment.
`-jprog`	false	Clear the existing FPGA configuration. This option should not be specified if performing runtime configuration.
`-ace <ACE_file>`	none	The output ACE file. The file prefix should not match any input file (bitstream, elf, data files) prefix.
`-hw <bitstream_file>`	none	The bitstream file for the system. If an SVF file is specified, it is used.
`-board <board_type>`	none	This identifies the JTAG chain on the board (Devices, IR length, Debug device, and so on). The options are given with respect to the System ACE controller. The script contains the options for some pre-defined boards. Board type options are: `user` The user specifies the `-configdevice` and `-debugdevice` option in the Options file. Refer to the `genace.opt` file for details. For Supported board type refer , "Supported Target Boards" below.
`-configdevice (only for -user board type)`	none	Configuration parameters for the device on the JTAG chain: • `devicenr`: Device position on the JTAG chain • `idcode`: ID code • `irlength`: Instruction Register (IR) length • `partname`: Name of the device The device position is relative to the System ACE device and these JTAG devices should be specified in the order in which they are connected in the JTAG chain on the board.
`-debugdevice`	none	The device containing PowerPC/MicroBlaze to debug or configure in the JTAG chain. Specify the device position on the chain, `devicenr`, number of processors, `cpunr` and processor options (such as OCM, Cache addresses). For MicroBlaze system, the script assumes the MicroBlaze v4.00.a processor version. To specify other MicroBlaze versions use "**cpu_version**" option. -debugdevice cpu_version **microblaze_v3** -debugdevice cpu_version **microblaze_v4**

`-target <target_type>`	ppc_hw	Target to use in the system for downloading ELF/Data file. Target types are:
		`ppc_hw`
		To connect to a ppc405 system
		`mdm`
		To connect to a MicroBlaze system. This assumes the presence of `opb_mdm` in the system.

Table 13-2: **Genace File Options *(Continued)***

Options	Default	Description
`-elf <list_of_Elf_Files>`	none	List of ELF files to download. If an SVF file is specified, it is used.
`-data <data_file> <load_address>`	none	List of Data/Binary file and its load address. The load address can be in decimal or hex format (0x prefix needed). If an SVF file is specified, it is used.
`-start_address <processor run address>`	If ELF files specified, the Start Address of the last ELF file. Else none.	Specify the address at which to start processor execution. This is useful when a data file is being loaded and processor should execute from load address.

Usage

```
xmd -tcl genace.tcl -jprog -target mdm -hw implementation/download.bit
-elf executable1.elf executable2.svf -data image.bin 0xfe000000 -board
ml401 -ace system.ace
```

Equivalent `genace.opt` file:

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board ml401
-target mdm
-elf executable1.elf executable2.svf
-data image.bin 0xfe000000
```

Supported Target Boards

The Tcl script supports the following three boards.

- Memec 2VP4/7 FG456: Board type - *memec*. This board has the following devices in the JTAG chain: XC18V04 →XC18V04 →XC2VP4/7
- ML300: Board type is *ml300*. This board has the following device in the JTAG chain: XC2VP7.
- ML310: Board type is *ml310*. This board has the following device in the JTAG chain: XC2VP30.
- MicroBlaze Demo Board: Board type is *mbdemo*. This board has the following device in the JTAG chain: XC2V1000.
- ML401: Board type is *ml401*. This board has the following devices in the JTAG chain: XCF32P →XC4VLX25 →XC95144XL.
- ML402: Board type is *ml402*. This board has the following devices in the JTAG chain: XCF32P →XC4VSX35 →XC95144XL.
- ML403: Board type is *ml403*. This board has the following devices in the JTAG chain: XCF32P →XC4VFX12 →XC95144XL.
- ML410: Board type is *ml410*. This board has the following device in the JTAG chain: XC4FX60

- ML411: : Board type is *ml411*. This board has the following device in the JTAG chain: XC4FX100

For a custom board, use the `-configdevice` option to specify the JTAG chain and use an OPT file.

Generating ACE Files

Single FPGA Device

System ACE files can be generated for the following scenarios. An example OPT file is given for each. Use the OPT file as "**xmd -tcl genace.tcl -opt genace.opt**".

Hardware and Software Configuration

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board ml401
-target mdm
-elf executable1.elf executable2.elf
```

Hardware and Software Partial Reconfiguration

```
-hw implementation/download.bit
-ace system.ace
-board ml401
-target mdm
-elf executable1.elf executable2.elf
```

Hardware Only Configuration

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board ml401
```

Hardware Only Partial Reconfiguration

```
-hw implementation/download.bit
-ace system.ace
-board ml401
```

Software Only Configuration

```
-jprog
-ace system.ace
-board ml401
-target mdm
-elf executable1.elf executable2.elf
```

Software Only Configuration on MicroBlaze; Downloading Using Fast Download

For a MicroBlaze target, software files can be downloaded with greater speeds using Fast Download methodology. This requires the presence of an FSL link between MicroBlaze and MDM, which makes downloading faster and creates a small ACE file.

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board ml401
-debugdevice devicenr 2 cpunr 1 -pfsl port 0 type s
-target mdm
-elf executable1.elf executable2.elf
```

ACE Generation for a Single Processor in Multi-Processor System:

Many of the Virtex™-II Pro and Virtex-4 devices contain two PowerPC processors or the System might contain multiple MicroBlaze processors. To generate an ACE file for a single processor use **-debugdevice** option. Use `cpunr` to specify the Processor instance.

In the example we assume a configuration with two PowerPC processors and ACE file is generated for processor number 2. The options file for this configuration is:

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board user
-configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
-debugdevice devicenr 1 cpunr 2 <= Note: The cpunr is 2
-target ppc_hw
-elf executable1.elf executable2.elf
```

Multi-Processor System Configuration:

We assume a configuration with two PowerPC processors and a MicroBlaze processor, each loaded with a single ELF file. The configuration of the board is specified in the options file.

```
-jprog
-hw implementation/download.bit
-ace system.ace
-board user
-configdevice devicenr 1 idcode 0x1266093 irlength 14 partname XC2VP20
# Options for PowerPC Processor 1 - Target Type, ELF files & Data files
-debugdevice devicenr 1 cpunr 1
-target ppc_hw
-elf executable1.elf
# Options for PowerPC Processor 2 - Target Type, ELF files & Data files
-debugdevice devicenr 1 cpunr 2
-target ppc_hw
-elf executable2.elf
# Options for MicroBlaze Processor - Target Type, ELF files & Data files
-debugdevice devicenr 1 cpunr 1
-target mdm
-elf executable3.elf
```

Note: When multi-processors are specified in an OPT file, processor-specific options such as target type, ELF/data files should follow `-debugdevice` option for that processor. The `cpunr` of the processor is inferred from `-debugdevice` option.

Multiple FPGA Devices

We assume a configuration with two FPGA devices, each with a single processor and a single ELF file. The configuration of the board is specified in the options file.

This configuration requires multiple steps to generate the ACE file.

1. Generate an SVF file for the first FPGA device.

   The options file is given below:

   ```
   -jprog
   -target ppc_hw
   -hw implementation/download.bit
   -elf executable1.elf
   -ace fpga1.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -configdevice devicenr 2 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 1 cpunr 1
   ```

   This generates the file `fpga1.svf`.

2. Generate an SVF file for the second FPGA device.

   The options file is given below:

   ```
   -jprog
   -target ppc_hw
   -hw implementation/download.bit
   -elf executable2.elf
   -ace fpga2.ace
   -board user
   -configdevice devicenr 1 idcode 0x123e093 irlength 10 partname XC2VP4
   -configdevice devicenr 2 idcode 0x123e093 irlength 10 partname XC2VP4
   -debugdevice devicenr 2 cpunr 1 <= Note: The change in Devicenr
   ```

   This generates the file `fpga2.svf`.

3. Concatenate the files in the following order: `fpga1.svf` and `fpga2.svf` to `final_system.svf`.
4. Generate the ACE file by calling `impact -batch svf2ace.scr`. The following SCR file should be used:

   ```
   svf2ace -wtck -d -i final_system.svf -o final_system.ace
   quit
   ```

Related Information

Adding a New Device to the JTAG Chain

An XMD supported device list is located at `$XILINX_EDK/data/xmd/deviceid.lst` or `$XILINX_EDK/data/xmd/devicetable.lst`. If XMD detects a device as "UNKNOWN" in the JTAG chain, you must specify the IR length of the device manually. This can be done in the following ways:

- Edit the `devicetable.list`.

Add a new entry for the device to the `devicetable.lst`. XMD then supports the device. You must add Device ID Code, IR Length, and Name in the list file. For more details, refer the `devicetable.lst` file.

- Edit the `deviceid.list`.

  Add a new entry for the device to the `deviceid.lst`. XMD then supports the device. You must add entries to all three sections in the list file. For more details, refer to the `deviceid.lst` file.

- Provide a GenACE OPT file with options.

  You can specify the `-configdevice` and `-debugdevice` options in the options file.

CF Device Format

To have the System ACE controller read the CF device, do the following:

1. Format the CF device as FAT 16.
2. Create a `Xilinx.sys` file in the root directory. This file contains the directory structure to use by the ACE controller. Copy the generated ACE file to the appropriate directory. For more information refer to the "iMPACT" section of the ISE documentation.

Chapter 14

EDK Shell

This chapter introduces the EDK Cygwin-based shell. It contains the following sections:

- Summary
- EDK Shell

Summary

The Xilinx® Embedded Development Kit (EDK) includes a few GNU-based tools such as the compiler, the debugger, and the `make` utility. These require a Unix® emulation shell; the Red Hat Cygwin™ shell and utilities are provided as part of the EDK installation.

The EDK-Installed Cygwin Environment

Xilinx EDK installs a Cygwin environment under `%XILINX_EDK%\cygwin`.

Requirements for Using an Existing Cygwin Environment

You can also use your existing Cygwin environment. Pre-existing Cygwin environments must conform to the following requirements:

- The Cygwin revision level must be 1.3.17 (December 2002) or later.
- The TC-shell utility (tcsh) must be available.

If your pre-existing Cygwin environment meets these requirements, it is used. If your existing Cygwin environment does not conform to these standards, you must use the EDK Shell. Error messages and warnings may be displayed based on state of the existing Cygwin installation.

EDK Shell

The EDK Shell is a Linux environment emulation mechanism based on Cygwin. It is used to run EDK tools and other bin utilities with a Linux look and feel on the Windows platform. To invoke the shell from the Windows Start menu, select **Start →Programs → Xilinx Platform Studio 8.2i →Accessories →Launch EDK Shell**. This launches the `xbash` utility, which is located at `%XILINX_EDK%\bin\nt\xbash.exe`.

The `xbash` utility requires that the `%XILINX%` and `%XILINX_EDK%` environment variables are already set.

Using xbash

To find information about xbash, use the **xbash -help** command.

Usage: **xbash [-c *<COMMAND>*] [-override] [-undo]**

-c *<COMMAND>*	Run *COMMAND* on the Xilinx EDK Cygwin Shell.
-override	Override local Cygwin installation, and use EDK's Cygwin.
-undo	Undo the effect of the **-override** option.
-help	Print this help menu.

When using an existing Cygwin installation on the machine, the specifications in the Cygwin Requirements section need to be met. If not, you will be prompted to upgrade to a newer version of Cygwin, or to install the required tools. In the event that a Cygwin version upgrade is necessary, you may choose to use the EDK Cygwin by using the **-override** and **-undo** options.

The -override and -undo Options

If an existing installation of Cygwin cannot be used, the Cygwin provided with EDK is available. To access it, use **xbash -override**.

Note: Use this option with caution because it changes the existing Cygwin setup on your machine. Using this option upgrades only essential Cygwin tools and DLLs on your machine; it does not upgrade all the tools.

If you prefer to revert this change and restore the original state of the Cygwin setup, use **xbash -undo**.

Cygwin Commands

You can run the following commands using EDK-installed Cygwin:

Table 14-1: **Cygwin Commands**

basename	cygwish80	fmt	mkfifo	rm	touch
bash	data	fold	mkinit	rmdir	tr
bigram	date	frcode	mknod	sdiff	true
cat	dd	gethost	mknodes	sed	tsort
chgrp	df	ginstall	mksignames	seq	tty
chmod	diff	grep	mksyntax	sh	umount
chown	diff3	gunzip	more	shred	uname
chroot	dir	gzip	mount	sleep	unexpand
cksum	dircolors	head	mv	sort	uniq
clear	dirname	hostname	nice	split	users
cmp	du	id	nl	stty	vdir

Table 14-1: **Cygwin Commands *(Continued)***

code	echo	join	od	sum	vi
comm	egrep	less	paste	sync	vim
cp	env	ln	pathchk	tac	wc
csplit	expand	locate	pinky	tail	which
cut	expr	logname	pr	tar	who
cvs	factor	ls	printenv	tcsh	whoami
cygitclsh30	false	make	printf	tee	xargs
cygitkwish30	fgrep	md5sum	ptx	test	yes
cygtclsh80	find	mkdir	pwd	tix4180	

XILINX®

Chapter 15

Command Line (no window) Mode

To invoke the XPS command line or "no window" mode, type the command **xps -nw** at the prompt in the EDK shell (the EDK Cygwin shell for a Windows Platform/Unix shell, with appropriate environment variables set up for Unix-based platforms). From the command line, you can generate the Microprocessor Software Specification (MSS) file and MAKE files and run the complete project flow in batch mode. You can also create an XMP project file or load a Xilinx® Microprocessor Project (XMP) file that was created by the XPS GUI.

When invoking the batch mode for XPS, you can specify a Tcl script along with the **-scr** option. XPS sources the Tcl script and then provides a command prompt. You can also provide an existing XMP file as input to XPS. XPS loads the project before presenting the command prompt.

XPS batch provides the ability to query the EDK design database. New Tcl commands were added for this purpose.

This chapter includes the following sections:

- Creating a New Empty Project
- Creating a New Project With an Existing MHS
- Opening an Existing Project
- Reading an MSS File
- Saving Files and Your Project
- Setting Project Options
- Executing Flow Commands
- Reloading an MHS File
- Adding a Software Application
- Deleting a Software Application
- Adding a Program File to a Software Application
- Deleting a Program File from a Software Application
- Setting Options on a Software Application
- Settings on Special Software Applications
- Restrictions

Creating a New Empty Project

To create a new project with no components, use the command:

```
xload new <basename>.xmp
```

XPS creates a project with an empty Microprocessor Hardware Specification (MHS) file and also creates the corresponding MSS file. All of the files have same base name as the XMP file. If XPS finds an existing project in the directory with same base name, then the XMP file is overwritten. However, if an 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 an Existing MHS

To create a new project, use the command:

```
xload mhs <basename>.mhs
```

XPS reads in the MHS file and creates the new project. The project name is the same as the MHS base name. All of the files generated have the same name as MHS. After reading in the MHS file, XPS also assigns various default drivers to each of the peripheral instances, if a driver is known and available to XPS.

Opening an Existing Project

If you already have an XMP project file, you can load that file using the command:

```
xload xmp <basename>.xmp
```

XPS reads in the XMP file. XPS takes the name of the MSS file from the XMP file, if specified. Otherwise, it assumes that these files are based on the XMP file name. If the XMP file does not refer to an MSS file, but the file exists in the project directory, XPS reads that MSS file. If the file does not exist, then XPS creates a new MSS file.

Reading an MSS File

To read an MSS file, use the command:

```
xload mss <filename>
```

If you do not specify <*filename*>, it is assumed to be the MSS file associated with this project. Loading an MSS file overrides 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 is overridden.

Saving Files and Your Project

To save MSS, XMP, and MAKE files for your project, use the command:

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

Command **save proj** saves all of the files.

Setting Project Options

You can set various project options and other fields in XPS using the **xset** command. You 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 15-1.

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

Table 15-1: **Options for xset and xget Commands**

Option Name	Description
arch	Set the target device architecture.
dev	Set the target part name.
package	Set the package of the target device.
speedgrade	Set the speedgrade of the target device.
searchpath [*dirs*]	Set the Search Path as a semicolon-separated list of directories.
hier [top|sub]	Set the design hierarchy.
topinst [*instname*]	Set the name by which the processor design is instantiated (if submodule).
hdl [vhdl|verilog]	Set the HDL language to be used.
sim_model [structural|behavioral |timing]	Set the current simulation mode.
simulator [mti|ncsim|none]	Set the simulator for which you want simulation scripts generated.
sim_x_lib **sim_edk_lib**	Set the simulation library paths. For details, refer to Chapter 3, "Simulation Model Generator (Simgen)."
pnproj [isefile]	Set the Project Navigator Project file to which designs will be exported.
usercmd1	Set the user command 1.
usercmd2	Set the user command 2.
pn_import_bit_file	Set the bit file to be imported from Project Navigator.
pn_import_bmm_file	Set the Block Memory Map (BMM) file to be imported from Project Navigator.
user_make_file	Specify a path to your make file. This file should not be same as the MAKE file generated by XPS.
ucf_file	Specify a path to the User Constraints File (UCF) to be used for implementation tools.

Executing Flow Commands

You can run various flow tools using the **run** command with appropriate options. XPS creates a MAKE file for the project and runs that MAKE file with the appropriate target. XPS generates the MAKE file every time the **run** command is executed. Valid options for the **run** command are shown in Table 15-2.

```
run option
```

Table 15-2: **Options for Command Run**

Option Name	Description
netlist	Generate the netlist.
bits	Run Xilinx Implementation tools flow and generate the bitstream.
libs	Generate the software libraries.
bsp	Generate the VxWorks Board Support Package (BSP) for the given PowerPC405 system.
program	Compile your program into Executable Linked Format (ELF) files.
init_bram	Update the bitstream with BRAM initialization information.
ace	Generate the SystemACE file after the BIT file is updated with BRAM info.
simmodel	Generate the simulation models without running the simulator.
sim	Generate the simulation models and run the simulator.
vp	Generate the virtual platform.
download	Download the bitstream onto the FPGA.
exporttopn	Export the processor design to Project Navigator.
importfrompn	Import BIT and BMM files from Project Navigator.
netlistclean	Delete the NGC/EDN netlist.
bitsclean	Delete the BIT, NCD, and BMM files in the implementation directory.
hwclean	Delete the implementation directory.
libsclean	Delete the software libraries.
programclean	Delete the ELF files.
swclean	Calls `libsclean` and `programclean`.
simclean	Delete the `simulation` directory.
vpclean	Delete the `virtualplatform` directory.
clean	Delete the all tool-generated files and directories.

Table 15-2: **Options for Command Run *(Continued)***

resync	Update any MHS file changes into the memory.
assign_default_drivers	Assign default drivers to all peripherals in the MHS file and save to MSS file.

Reloading an MHS File

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

```
run resync
```

This causes XPS to re-read MHS, MSS, and XMP files again.

Adding a Software Application

You can add new software application projects in an XPS batch using the **xadd_swapp** command. When adding a new software application, you must specify a name for that application and a processor instance on which that application runs. By default, XPS assumes that the ELF file related to a new software application is 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 existing software application can be deleted from project in the XPS batch using the **xdel_swapp** command. You must specify the name of the software application that you want to delete.

```
xdel_swapp <swapp_name>
```

Adding a Program File to a Software Application

You 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 software application to which the file must be added and the location of the program file must be specified. XPS automatically adds it as a source or header based on the extension of the file.

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

Deleting a Program File from a Software Application

You 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 software application and the program file location needs to be specified.

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

Setting Options on a Software Application

You can set various software application options and other fields in XPS using the **xset_swapp_prop_value** command. You can also display the current value of those fields using the **xget_swapp_prop_value** command. The **xget_swapp_prop_value** command also returns the result as a Tcl string result. The various options taken by the two commands are shown in Table 15-3.

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

Table 15-3: **Options for xset_swapp_prop_value and xget_swapp_prop_value Commands**

Option Name	Description
sources	Displays a list of sources. For adding sources, use the **xadd_swapp_progfile** command.
headers	Displays a list of headers. For adding header files, use the **xadd_swapp_progfile** command.
executable	The path to the executable (ELF) file.
procinst	The processor instance associated with this software application.
compileroptlevel	Specify the 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	The debug symbol setting. Value can be from none to two corresponding none, **-g**, and **-gstabs** options.
searchlibs	The library search path option (**-L**).
searchincl	The include search path option (**-I**).
lflags	The libraries to link (**-l**).
progstart	The program start address.
stacksize	The stack size.
heapsize	The heap size.
linkerscript	The linker script (**-Wl**, **-T -Wl**,***<linker_script_file>***)
progccflags	All other compiler options which cannot be set using the above options.
init_bram	Specify whether the ELF file should be used for BRAM initialization.
mode	Specify whether the ELF should be compiled in XMDStub mode (MicroBlaze™ only) or executable mode.

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 an XMDStub application provided by XPS. The only setting available on these special software applications is to "**Mark for BRAM Initialization**." You can use the **xset_swapp_prop_value**. 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 recognizes mymblaze_bootloop and mymblaze_xmdstub as software applications. You can set the **init_bram** option on this application.

```
XPS% xset_swapp_prop_value mymblaze_bootloop init_bram true
XPS% xset_swapp_prop_value mymblaze_xmdstub init_bram false
```

This assumes that there is no software application by the same name. If there is an application with same name, you will not be able to change the settings using the XPS Tcl interface. Therefore, 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.

Restrictions

MSS Changes

XPS-batch supports limited MSS editing. If you want to make any changes in the MSS file, you must hand edit the file, make the changes, and then run the **xload mss** command to load the changes into XPS. You do not have to close the project. You can save the MSS file, edit it, and then re-load it into the project with the **xload mss** command.

XMP Changes

Xilinx recommends that you *do not* edit the XMP file manually. XPS-batch supports changing of project options through commands. It also supports adding source and header files to a processor and setting any compiler options. Any other changes must be done from XPS.

Appendix A

GNU Utilities

This appendix describes the GNU utilities available for use with EDK. It contains the following sections:

- General Purpose Utility for MicroBlaze and PowerPC
- Utilities Specific to MicroBlaze and PowerPC
- Other Programs and Files

General Purpose Utility for MicroBlaze and PowerPC

cpp

Pre-processor for C and C++ utilities. The preprocessor is automatically invoked by GCC (GNU Compiler Collection) and implements directives such as file-include and define.

gcov

This is a program used in conjunction with GCC to profile and analyze test coverage of programs. It can also be used with the `gprof` profiling program.

Utilities Specific to MicroBlaze and PowerPC

Utilities specific to MicroBlaze™ have the prefix "mb," as shown below. The PowerPC™ versions of the programs are prefixed with "powerpc-eabi."

mb-addr2line

This program uses debugging information in the executable to translate a program address into a corresponding line number and file name.

mb-ar

This program creates, modifies, and extracts files from archives. An archive is a file that contains one or more other files, typically object files for libraries.

mb-as

This is the assembler program.

mb-c++

This is the same cross compiler as mb-gcc, invoked with the programming language set to C++. This is the same as mb-g++.

mb-c++filt

This program performs name demangling for C++ and Java function names in assembly listings.

mb-g++

This is the same cross compiler as mb-gcc, invoked with the programming language set to C++. This is the same as mb-c++.

mb-gasp

This is the macro preprocessor for the assembler program.

mb-gcc

This is the cross compiler for C and C++ programs. It automatically identifies the programming language used based on the file extension.

mb-gdb

This is the debugger for programs.

mb-gprof

This is a profiling program that allows you to analyze how much time is spent in each part of your program. It is useful for optimizing run time.

mb-ld

This is the linker program. It combines library and object files, performing any relocation necessary, and generates an executable file.

mb-nm

This program lists the symbols in an object file.

mb-objcopy

This program translates the contents of an object file from one format to another.

mb-objdump

This program displays information about an object file. This is very useful in debugging programs, and is typically used to verify that the correct utilities and data are in the correct memory location.

mb-ranlib

This program creates an index for an archive file, and adds this index to the archive file itself. This allows the linker to speed up the process of linking to the library represented by the archive.

mb-readelf

This program displays information about an Executable Linked Format (ELF) file.

mb-size

This program lists the size of each section in the object file. This is useful to determine the static memory requirements for utilities and data.

mb-strings

This is a useful program for determining the contents of binary files. It lists the strings of printable characters in an object file.

mb-strip

This program removes all symbols from object files. It can be used to reduce the size of the file, and to prevent others from viewing the symbolic information in the file.

Other Programs and Files

The following Tcl and Tk shells are invoked by various front-end programs:

- cygitclsh30
- cygitkwish30
- cygtclsh80
- cygwish80
- tix4180

XILINX®

Appendix B

Interrupt Management

This appendix describes interrupt handling and the role of Libgen in MicroBlaze™ and PowerPC™. The following sections are included:

- Overview of Interrupt Management in EDK
- Libgen Customization
- Example Systems for MicroBlaze
- Example Systems for PowerPC

Note: The Board Support Package (BSP) handles some interrupt management functions. For information on these, refer to the "Standalone Board Support Package" document in the OS and Libraries document collection (included in the `/doc` directory of your EDK installation).

Overview of Interrupt Management in EDK

Steps Involved in Interrupt Management

Interrupt management requires you to:

- Write interrupt handler routines or interrupt service routines (ISRs) for peripherals.
- Register the ISRs in the interrupt vector table.
- Enable the interrupts in the processor and interrupt controller.

Furthermore, you must set up the Microprocessor Hardware Specification (MHS) and Microprocessor Software Specification (MSS) files appropriately.

For more information and examples, refer toXilinx Application Note number 778.

Interrupt Handling in MicroBlaze and PowerPC

Interrupt Ports

MicroBlaze has one interrupt port; the PowerPC has critical and a non-critical interrupt ports.

Enabling Interrupts

- For Microblaze: use the function **`microblaze_enable_interrupts`** to enable interrupts.
- For PowerPC: use the function **`XExc_mEnableExceptions`** to enable interrupts.

For Additional Information

Refer to the "Standalone Board Support Package" document (linked to the OS and Libraries document collection page in the `\doc` directory of your EDK installation) for more information about these functions.

If using an embedded OS, refer to its specific document for functions related to enabling interrupts.

Connecting Interrupts

An interrupt peripheral is any component that sends a signal (the interrupt) to an interrupt port on the processor and which causes the processor to pause its program to service the interrupt.

An interrupt controller is *not* required if there is a single interrupt peripheral or if there is an external interrupt pin.

Note: If the single peripheral can generate multiple interrupts, an interrupt controller is required.

To connect more than one interrupt to the processor's interrupt port, you must use an interrupt controller. Xilinx provides two interrupt controllers: (1) the DCR Interrupt Controller and (2) the OPB Interrupt Controller. Both allow up to 32 interrupts. The controllers manage multiple interrupts through a simple prioritization scheme, as shown in Figure B-1.

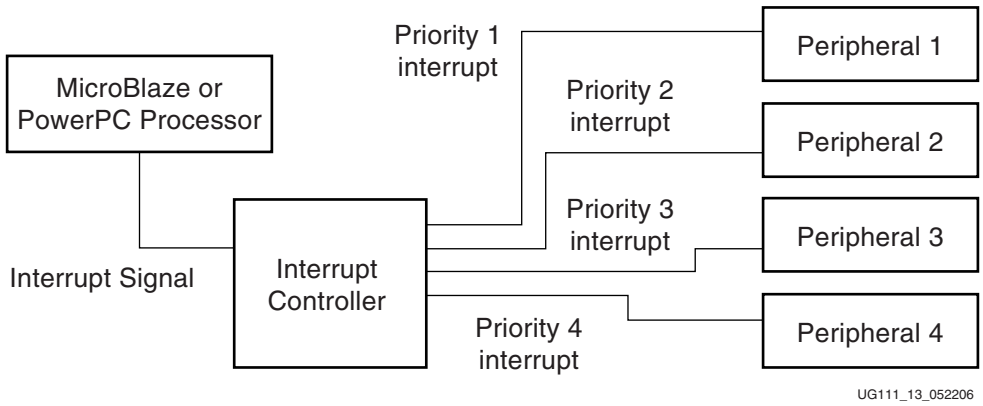

Figure B-1: **Interrupt Controller and Peripherals (Example)**

The section of MHS code corresponding to the figure above would be as follows:

```
BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.c
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10FF
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = Priority4_interrupt & Priority3_interrupt &
Priority2_interrupt & Priority1_interrupt
END
## MicroBlaze example
BEGIN microblaze
:
:
port INTERRUPT = interrupt
:
END
## PPC example
BEGIN ppc405
:
:
port EICC405EXTINPUTIRQ = interrupt
:
END
```

You must connect the interrupt signal from the interrupt peripheral to the interrupt input of the processor. In the example above, the interrupt peripheral is the OPB Interrupt Controller, and the interrupt signal is coming from port IRQ. The order of priority for each interrupt signal managed by the OPB Interrupt Controller is defined by the lowest priority signal first. See, for example, the interrupt controller Intr port entry in the MHS file code, above.

If you are connecting an external signal to the processor interrupt port, the external port in the MHS will be similar to the following:

```
PORT Interrupt_In = interrupt, DIR = IN, LEVEL = LOW, SIGIS = INTERRUPT
```

For specific information about MHS syntax, refer to the Microprocessor Hardware Specification chapter in the *Platform Specification Format Reference Manual*, available in the `\doc` directory of your EDK installation.

When interrupt management is accomplished using an interrupt controller peripheral, the following restrictions apply:

- The priorities associated with the interrupt sources connected to the interrupt controller peripheral are fixed when you define them in the MHS file. They cannot be changed in your code.
- There cannot be any gaps in the range of interrupt priority sources defined in the MHS file. For example, in the MHS file snippet on page 207, a definition such as the following would not be acceptable:

  ```
  port Intr = Priority4_interrupt & 0x0 & Priority2_interrupt ...
  ```

Interrupt Service Routines (ISRs)

On encountering an interrupt, the processor must call an Interrupt Service Routine (ISR), also known as an "interrupt handler" or "exception handler," to manage the interrupt. You create ISRs as a C function in the form of **`void func (void *)`** for any custom peripheral that generates an interrupt. The ISR for the MicroBlaze and PowerPC are registered with **`microblaze_register_handler`** and **`XExc_RegisterHandler`**, respectively. When using the DCR Interrupt Controller or OPB Interrupt Controller, the ISR for the interrupt peripheral is registered with the **`XIntc_RegisterHandler`** function.

For Additional Information

Refer to the "Standalone Board Support Package" document (linked to the OS and Libraries document collection page (in the `\doc` directory of your EDK installation) for more information about register handler functions.

If using an embedded OS, please refer to its specific document for functions related to enabling interrupts.

For more information on the functions associated with the interrupt controller, refer to the API documentation found the EDK install directory `C:/EDK/doc/xilinx_drivers_api_toc.htm`.

Interrupt Vector Tables

MicroBlaze

On interrupts, the processor jumps to a predefined address location that contains the main ISR. MicroBlaze jumps to address 0x10. The OPB and DCR Interrupt Controllers: the controller's ISR manages ISRs for each of its 32 interrupts.

Table B-1: **MicroBlaze Interrupt Handling Mechanism**

0x10	Main ISR Address
*	Individual ISR 1
*	Individual ISR 2
	. . . . .
*	Individual ISR 32

*After going to 0x10 to find the main ISR, the processor jumps to the main ISR address, which contains the routine that determines which individual ISR should be executed. From there, the processor jumps to the address of the individual ISR and executes the routine programmed at that location.

PowerPC

The PowerPC processor initiates interrupt handling differently from MicroBlaze, but the process, once started, is very similar.

For a detailed description of interrupt handling for PowerPC processors, see the "Exceptions and Interrupts" chapter of the *PowerPC Processor Reference Guide* (in the `\doc` directory of your EDK installation).

Libgen Customization

Purpose of the Libgen Tool

The Libgen tool generates the hardware system address map, which defines the base and high addresses for each peripheral connected to the processor. It also generates interrupt priorities for each peripheral connected to an interrupt controller peripheral. The information is generated in the header file `xparameters.h`. Based on the MSS file, Libgen performs the following interrupt management tasks:

- Registers an ISR with the interrupt vector table for MicroBlaze.
- If an interrupt controller peripheral is used, generates the vector table for the interrupt controller peripheral.
- Registers ISRs for each peripheral interrupt signal connected to the interrupt controller peripheral in the vector table, if defined in the MSS file.

Introducing xparameters.h

The `xparameters.h` file defines the hardware system used by the software. The file includes an address map of the hardware system, which includes the base and high addresses for each peripheral connected to a processor. The tool uses the following naming conventions for generating base and high addresses:

```
XPAR_<PERIPHERAL_INSTANCE_NAME>_BASE_ADDR

XPAR_<PERIPHERAL_INSTANCE_NAME>_HIGH_ADDR
```

The interrupt controller driver uses the definitions in `xparameters.h` to establish the priorities and the maximum number of interrupt sources in a hardware system. Libgen generates priorities for each interrupt signal as #defines in `xparameters.h,` using the following naming conventions:

```
XPAR_<INTC_INSTANCE_NAME>_<PERIHPERAL_INSTANCE_NAME>_
  <PERIPHERAL_INTERRUPT_SIGNAL_NAME>_INTR

XPAR_<PERIHPERAL_INSTANCE_NAME>_<PERIPHERAL_INTERRUPT_SIGNAL_NAME>
  _MASK
```

For example, the `priority 1` interrupt is defined as:

```
XPAR_OPB_INTC_0_PERIPHERAL_1_PRIORITY_1_INTERRUPT_INTR

XPAR_PERIPHERAL_1_PRIORITY_1_INTERRUPT_MASK
```

in `xparameters.h`, where `opb_intc_0` is the instance name of the interrupt controller peripheral.

Libgen also generates XPAR_<INTC_INSTANCE_NAME>_MAX_NUM_INTR_INPUTS to define the total number of interrupting sources connected to the interrupt controller peripheral, as shown in Figure B-1 on page 206. The INTR definitions define the identification of the interrupting sources and should be in the range of:

```
XPAR_<INTC_INSTANCE_NAME>_MAX_NUM_INTR_INPUTS - 1
```

with 0 being the highest priority interrupt.

Example Systems for MicroBlaze

MicroBlaze System *Without* an Interrupt Controller (Single Interrupt Signal)

An interrupt controller is not required if:

- There is a single interrupting peripheral.
- There is an external interrupting pin.

Note: If the single peripheral can generate multiple interrupts, an interrupt controller is required.

Procedure

To set up a system without an interrupt controller, that is, a system that handles only one level-sensitive interrupt signal, you must:

1. In XPS, with the ports filter selected in the System Assembly View, connect the interrupt signal for the peripheral, or the external interrupt signal to the interrupt input of the MicroBlaze processor.
2. Write the interrupt handler routine for the signal. The base address of the peripheral instance is accessed as XPAR_<INSTANCE_NAME>_BASEADDR.
3. In your software application, the ISR is registered to the processor through the generic interrupt controller driver called intc. There are both low level and high level drivers. Please refer to the interrupt controller's API documentation in your EDK installation at: ../EDK/sw/XilinxProcessorIPLib/drivers/intc_v1_00_c/doc/html/api/index.html.

   Examples on using some of the interrupt controller's functions can be found in Xilinx Application Note 778 at:
   http://direct.xilinx.com/bvdocs/appnotes/xapp778.pdf
4. Libgen and mb-gcc are executed. For details on this process, see "Libgen Customization" on page 209.

Example MHS File Snippet (for an Internal Interrupt Signal)

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = interrupt
port CaptureTrig0 = net_gnd
END
```

```
begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 4.00.a
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt
end
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END
```

Example C Program

```
#include <xtmrctr_l.h>
#include <xgpio_l.h>
#include <xparameters.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/* timer interrupt service routine */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  unsigned int gpio_data;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(baseaddr_p, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
    /* Increment the count */

    if ((count <<= 1) > 8) {
      count = 1;
    }

    /* Write value to gpio. 0 means light up, hence count is negated */
    gpio_data = ~count;

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(baseaddr_p, 0, csr);

  }
}

void
```

```
main() {

  unsigned int gpio_data;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

/* Set the gpio as output on high 3 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 1000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

  /* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);

  /* Wait for interrupts to occur */
  while (1)
      ;

}
```

Example MHS File Snippet for an External Interrupt Signal

```
PORT interrupt_in1 = interrupt_in1, DIR = IN, LEVEL = LOW, SIGIS =
INTERRUPT

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 4.00.a
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt_in1
end
```

Example MSS File Snippet

```
PARAMETER int_handler = global_int_handler, int_port = interrupt_in1
```

Example C Program

```
#include <xparameters.h>

/* global interrupt service routine */
void global_int_handler(void * arg) {
/* Handle the global interrupts here */

}

void
```

```
main() {

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();
/* Wait for interrupts to occur */
  while (1)
      ;

}
```

MicroBlaze System *With* an Interrupt Controller (One or More Interrupt Signals)

An Interrupt Controller peripheral (INTC) must be present if two or more interrupts might be generated at the same time. When an interrupt is generated, the interrupt handler for the INTC, XIntc_DeviceInterruptHandler, is called. This function accesses the interrupt controller to find the highest priority device that generated an interrupt. The priority level is determined via the vector table that Libgen creates automatically. On return from the peripheral interrupt handler, the INTC acknowledges the interrupt and handles any remaining interrupts in order of priority.

Procedure

To set up a system with one or more interrupting devices and an interrupt controller, you must complete the following steps:

1. In XPS, with the ports filter selected in the System Assembly View, assign the interrupt signals for all the peripherals to the interrupt port (Intr in most cases) of the interrupt controller.

   The interrupt signal output of INTC is then connected to the interrupt input of the MicroBlaze processor.

   Libgen creates an interrupt mask and interrupt ID for each interrupt signal (see "Libgen Customization" on page 209).

2. Write the interrupt handler functions for each interruptible peripheral.
3. In your software application, the UART ISR is registered to the processor through the generic interrupt controller driver called intc. There are both low level and high level drivers. Please refer to the interrupt controller's API documentation in your EDK installation at:
   ../EDK/sw/XilinxProcessorIPLib/drivers/intc_v1_00_c/doc/html/api/index.html.

   Examples on using some of the interrupt controller's functions can be found in Xilinx Application Note 778 at:
   http://direct.xilinx.com/bvdocs/appnotes/xapp778.pdf

   Note: Do not give the INTC interrupt signal an INT_HANDLER keyword. If the INT_HANDLER keyword is not present for a particular peripheral, a default dummy interrupt handler is used.
4. Run Libgen and mb-gcc. For details on this process, see "Libgen Customization" on page 209.

MHS File Snippet Showing an INTC for a Timer and UART

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = timer1
port CaptureTrig0 = net_gnd
END

BEGIN opb_uartlite
parameter INSTANCE = myuart
parameter HW_VER = 1.00.b
parameter C_BASEADDR  = 0xFFFF8000
parameter C_HIGHADDR = 0xFFFF80FF
parameter C_DATA_BITS  = 8
parameter C_CLK_FREQ   = 30000000
parameter C_BAUDRATE   = 19200
parameter C_USE_PARITY = 0
bus_interface SOPB = opb_bus
port RX = rx
port TX = tx
port Interrupt = uart1
END

BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.c
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10ff
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = timer1 & uart1
END

begin microblaze
parameter INSTANCE = mblaze
parameter HW_VER = 4.00.a
bus_interface DOPB = opb_bus
bus_interface DLMB = d_lmb
bus_interface ILMB = i_lmb
port INTERRUPT = interrupt
end
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END

BEGIN DRIVER
parameter HW_INSTANCE = myuart
parameter DRIVER_NAME = uartlite
```

```
parameter DRIVER_VER = 1.00.b
END
```

Example C Program

```
#include <xtmrctr_l.h>
#include <xuartlite_l.h>
#include <xintc_l.h>
#include <xgpio_l.h>
#include <xparameters.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/* uartlite interrupt service routine */
void uart_int_handler(void *baseaddr_p) {
  char c;
  /* till uart FIFOs are empty */
  while (!XUartLite_mIsReceiveEmpty(XPAR_MYUART_BASEADDR)) {
    /* read a character */
    c = XUartLite_RecvByte(XPAR_MYUART_BASEADDR);
    /* if the character is between "0" and "9" */
    if ((c>47) && (c<58)) {
      timer_count = c-48;
      /* print character on hyperterminal (STDOUT) */
      putnum(timer_count);
      /* Set timer with new value of timer_count */
      XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0, (timer_count*tim
er_count+1) * 1000000);
    }
  }
}


/* timer interrupt service routine */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  unsigned int gpio_data;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
    /* Increment the count */

    if ((count <<= 1) > 8) {
      count = 1;
    }

    /* Write value to gpio. 0 means light up, hence count is negated */
    gpio_data = ~count;

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);
```

```
    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0, csr);

  }
}

void
main() {

  unsigned int gpio_data;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

/* Connect uart interrupt handler that will be called when an interrupt
* for the uart occurs
*/
  XIntc_RegisterHandler(XPAR_MYINTC_BASEADDR,
XPAR_MYINTC_MYUART_INTERRUPT_INTR,
(XInterruptHandler)uart_int_handler,
(void *)XPAR_MYUART_BASEADDR);


  /* Start the interrupt controller */
  XIntc_mMasterEnable(XPAR_MYINTC_BASEADDR);

  /* Set the gpio as output on high 3 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 1000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

  /* Enable timer and uart interrupts in the interrupt controller */
  XIntc_mEnableIntr(XPAR_MYINTC_BASEADDR, XPAR_MYTIMER_INTERRUPT_MASK
| XPAR_MYUART_INTERRUPT_MASK);

/* Enable Uartlite interrupt */
XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);

  /* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);


  /* Wait for interrupts to occur */
  while (1)
      ;
}
```

Example Systems for PowerPC

PowerPC System *Without* Interrupt Controller (Single Interrupt Signal)

An interrupt controller is not required if:

- There is a single interrupting peripheral or an external interrupting pin.
- Its interrupt signal is level-sensitive.

Note: If a single peripheral can generate multiple interrupts, an interrupt controller is required.

Procedure

To set up a system without an interrupt controller, that is, a system that handles only one level-sensitive interrupt signal:

1. In XPS, with the ports filter selected in the System Assembly View, connect the interrupt signal for the peripheral, or the external interrupt signal, to one of the interrupt inputs of the PowerPC. The interrupt inputs can be critical or non-critical.

   Libgen creates a definition in `xparameters.h`. See "Libgen Customization" on page 209 for more information.

2. Write the interrupt handler routine for the signal. The base address of the peripheral instance is accessed as `XPAR_<PERIPHERAL_INSTANCE_NAME_BASEADDR`.

3. In your software application, the ISR is registered to the processor through the generic interrupt controller driver called `intc`. There are both low level and high level drivers. Please refer to the interrupt controller's API documentation in your EDK installation at: `../EDK/sw/XilinxProcessorIPLib/drivers/intc_v1_00_c/doc/html/api/index.html`.

   Examples on using some of the interrupt controller's functions can be found in Xilinx Application Note 778 at:
   `http://direct.xilinx.com/bvdocs/appnotes/xapp778.pdf`.

4. Run Libgen and `powerpc-eabi-gcc`. For details on this process, see "Libgen Customization" on page 209.

Example MHS File Snippet (for an Internal Interrupt Signal)

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = interrupt
port CaptureTrig0 = net_gnd
END

BEGIN ppc405
 PARAMETER INSTANCE = ppc405_0
 PARAMETER HW_VER = 2.00.c
 BUS_INTERFACE JTAGPPC = jtagppc_0_0
 BUS_INTERFACE IPLB = myplb
 BUS_INTERFACE DPLB = myplb
 PORT PLBCLK = sys_clk_s
```

```
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT CPMC405CLOCK = sys_clk_s
PORT EICC405EXTINPUTIRQ = interrupt
END
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END
```

Example C Program

```
/*****************************************************************
*   Copyright (c) 2001 Xilinx, Inc.  All rights reserved.
*   Xilinx, Inc.
*
*
* This program uses the timer and gpio to demonstrate interrupt
handling.
* The timer is set to interrupt regularly. The frequency is set in the
code.
* Every time there is an interrupt from the timer,
* a rotating display of LEDs on the board is updated.
*
* The LEDs and switches are in these bit positions:
* LSB 0: gpio_io<3>
* LSB 1: gpio_io<2>
* LSB 2: gpio_io<1>
* LSB 3: gpio_io<0>
*****************************************************************
/

/* This is the list of files that must be included to access the
peripherals:
 * xtmrctr.h - to access the timer
 * xgpio_l.h  - to access the general purpose I/O
* xparameters.h  - General purpose definitions.  Must always be included
 *          when any drivers/print routines are accessed.  This defines
 *         addresses of all peripherals, declares the interrupt service
 *           routines, etc.
 */
#include <xtmrctr_l.h>
#include <xgpio_l.h>
#include <xparameters.h>
#include <xexception_l.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */
```

```
unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/*
 * Interrupt service routine for the timer.  It has been declared as an
ISR in
 * the mss file using the attribute INT_HANDLER. The ISR can be written
as a normal C routine.
 * The peripheral can be accessed using XPAR_<peripheral name in the mhs
file>_BASEADDR
 * as the base address.
 */
void timer_int_handler(void * baseaddr_p) {
   int baseaddr = (int)baseaddr_p;
   unsigned int csr;
   unsigned int gpio_data;
  int baseaddr = (int) baseaddr_p;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(baseaddr, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
    /* Shift the count */

    if ((count <<= 1) > 16) {
      count = 1;
    }

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, ~count);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0, csr);

  }
}

void
main() {

  int i, j;

  /* Initialize exception handling */
  XExc_Init();

  /* Register external interrupt handler */
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
(XExceptionHandler)timer_int_handler, (void *)XPAR_MYTIMER_BASEADDR);

/* Set the gpio as output on high 4 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 8000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );
```

```
/* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);

  /* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);

  /* Wait for interrupts to occur */
  while (1)
      ;
}
```

Example MHS File Snippet (For External Interrupt Signal)

```
PORT interrupt_in1 = interrupt_in1, DIR = IN, LEVEL = LOW, SIGIS =
INTERRUPT

BEGIN ppc405
 PARAMETER INSTANCE = ppc405_0
 PARAMETER HW_VER = 2.00.c
 BUS_INTERFACE JTAGPPC = jtagppc_0_0
 BUS_INTERFACE IPLB = myplb
 BUS_INTERFACE DPLB = myplb
 PORT PLBCLK = sys_clk_s
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT CPMC405CLOCK = sys_clk_s
PORT EICC405EXTINPUTIRQ = interrupt_in1
END
```

Example MSS File Snippet

```
PARAMETER int_handler = global_int_handler, int_port = interrupt_in1
```

Example C Program

```
#include <xparameters.h>

/* global interrupt service routine */
void global_int_handler(void * arg) {
/* Handle the global interrupts here */

}

void
main() {

/* Initialize exception handling */
  XExc_Init();

  /* Register external interrupt handler */
```

```
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
(XExceptionHandler)global_int_handler, (void *)0);

/* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);

/* Wait for interrupts to occur */
  while (1)
      ;

}
```

PowerPC System *With* an Interrupt Controller (One or More Interrupt Signals)

An Interrupt Controller peripheral (INTC) should be present if more than one interrupt can be generated. When an interrupt is generated, the interrupt handler for the Interrupt Controller, XIntc_DeviceInterruptHandler, is called. This function accesses the interrupt controller to find the highest priority device that generated an interrupt. The priority level is determined via the exception table that Libgen creates automatically. On return from the peripheral interrupt handler, the intc interrupt handler acknowledges the interrupt and handles any remaining interrupts in order of priority.

Procedure

To set up a system with one or more interrupting devices and an interrupt controller, you must:

1. In XPS, with the ports filter selected in the System Assembly View, assign the interrupt signals for all peripherals to the Intr port of the interrupt controller. The interrupt signal output of INTC is then connected to one of the interrupt inputs of the PowerPC processor. The interrupt inputs can be either critical or non-critical.

   Libgen creates a definition in xparameters.h for XPAR_<*INTC_INSTANCE_NAME*>_BASEADDR, which is mapped to the base address of each peripheral specified in your program. Libgen also creates an interrupt mask and interrupt ID for each interrupt signal. This can be used to enable or disable interrupts. For more information, see "Libgen Customization" on page 209.

2. Write the interrupt handler functions for each interruptible peripheral.
3. Using your software application, the UART ISR is registered to the processor through the generic interrupt controller driver called intc. There are both low level and high level drivers. Please refer to the interrupt controller's API documentation in your EDK installation at:
   ../EDK/sw/XilinxProcessorIPLib/drivers/intc_v1_00_c/doc/html/api/index.html.

   Examples on using some of the interrupt controller's functions can be found in Xilinx Application Note 778 at:
   http://direct.xilinx.com/bvdocs/appnotes/xapp778.pdf.

   Note: Do not give the INTC interrupt signal an INT_HANDLER keyword. If the INT_HANDLER keyword is not present for a particular peripheral, a default dummy interrupt handler is used.
4. Run Libgen and mb-gcc. For details on this process, see "Libgen Customization" on page 209.

Example MHS File Snippet

```
BEGIN opb_timer
parameter INSTANCE = mytimer
parameter HW_VER = 1.00.b
parameter C_BASEADDR = 0xFFFF0000
parameter C_HIGHADDR = 0xFFFF00ff
bus_interface SOPB = opb_bus
port Interrupt = timer1
port CaptureTrig0 = net_gnd
END

BEGIN opb_uartlite
parameter INSTANCE = myuart
parameter HW_VER = 1.00.b
parameter C_BASEADDR  = 0xFFFF8000
parameter C_HIGHADDR = 0xFFFF80FF
parameter C_DATA_BITS  = 8
parameter C_CLK_FREQ   = 30000000
parameter C_BAUDRATE   = 19200
parameter C_USE_PARITY = 0
bus_interface SOPB = opb_bus
port RX = rx
port TX = tx
port Interrupt = uart1
END

BEGIN opb_intc
parameter INSTANCE = myintc
parameter HW_VER = 1.00.c
parameter C_BASEADDR = 0xFFFF1000
parameter C_HIGHADDR = 0xFFFF10ff
bus_interface SOPB = opb_bus
port Irq = interrupt
port Intr = timer1 & uart1
END

BEGIN ppc405
 PARAMETER INSTANCE = ppc405_0
 PARAMETER HW_VER = 2.00.c
 BUS_INTERFACE JTAGPPC = jtagppc_0_0
 BUS_INTERFACE IPLB = myplb
 BUS_INTERFACE DPLB = myplb
 PORT PLBCLK = sys_clk_s
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT CPMC405CLOCK = sys_clk_s
PORT EICC405EXTINPUTIRQ = interrupt
END
```

Example MSS File Snippet

```
BEGIN DRIVER
parameter HW_INSTANCE = mytimer
parameter DRIVER_NAME = tmrctr
parameter DRIVER_VER = 1.00.b
parameter INT_HANDLER = timer_int_handler, INT_PORT = Interrupt
END

BEGIN DRIVER
parameter HW_INSTANCE = myuart
parameter DRIVER_NAME = uartlite
parameter DRIVER_VER = 1.00.b
END
```

Example C Program

```
#include <xtmrctr_l.h>
#include <xuartlite_l.h>
#include <xintc_l.h>
#include <xgpio_l.h>
#include <xparameters.h>

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.*/

unsigned int count = 1;  /* default count */
unsigned int timer_count = 1; /* default timer_count */

/* uartlite interrupt service routine */
void uart_int_handler(void *baseaddr_p) {
  char c;
  /* till uart FIFOs are empty */
  while (!XUartLite_mIsReceiveEmpty(XPAR_MYUART_BASEADDR)) {
    /* read a character */
    c = XUartLite_RecvByte(XPAR_MYUART_BASEADDR);
    /* if the character is between "0" and "9" */
    if ((c>47) && (c<58)) {
      timer_count = c-48;
      /* print character on hyperterminal (STDOUT) */
      putnum(timer_count);
      /* Set timer with new value of timer_count */
      XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0, (timer_count*tim
er_count+1) * 1000000);
    }
  }
}


/* timer interrupt service routine */
void timer_int_handler(void * baseaddr_p) {
  unsigned int csr;
  unsigned int gpio_data;
int baseaddr = (int) baseaddr_p;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = XTmrCtr_mGetControlStatusReg(baseaddr, 0);

  if (csr & XTC_CSR_INT_OCCURED_MASK) {
```

```
    /* Increment the count */

    if ((count <<= 1) > 8) {
      count = 1;
    }

    /* Write value to gpio. 0 means light up, hence count is negated */
    gpio_data = ~count;

    XGpio_mSetDataReg(XPAR_MYGPIO_BASEADDR, gpio_data);


    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0, csr);

  }
}

void
main() {

  unsigned int gpio_data;

/* Initialize exception handling */
  XExc_Init();

  /* Register external interrupt handler */
  XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
(XExceptionHandler)XIntc_DeviceInterruptHandler, (void
*)XPAR_MYINTC_DEVICE_ID);

/* Connect uart interrupt handler that will be called when an interrupt
* for the uart occurs
*/
  XIntc_RegisterHandler(XPAR_MYINTC_BASEADDR,
XPAR_MYINTC_MYUART_INTERRUPT_INTR,
(XInterruptHandler)uart_int_handler,
(void *)XPAR_MYUART_BASEADDR);

  /* Start the interrupt controller */
  XIntc_mMasterEnable(XPAR_MYINTC_BASEADDR);

  /* Set the gpio as output on high 3 bits (LEDs)*/
  XGpio_mSetDataDirection(XPAR_MYGPIO_BASEADDR, 0x00);

  /* set the number of cycles the timer counts before interrupting */
  XTmrCtr_mSetLoadReg(XPAR_MYTIMER_BASEADDR, 0,
(timer_count*timer_count+1) * 1000000);

  /* reset the timers, and clear interrupts */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );

/* Enable timer and uart interrupts in the interrupt controller */
  XIntc_mEnableIntr(XPAR_MYINTC_BASEADDR, XPAR_MYTIMER_INTERRUPT_MASK
| XPAR_MYUART_INTERRUPT_MASK);

/* Enable Uartlite interrupt */
XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);
```

```
/* start the timers */
  XTmrCtr_mSetControlStatusReg(XPAR_MYTIMER_BASEADDR, 0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK);

/* Enable PPC non-critical interrupts */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);

  /* Wait for interrupts to occur */
  while (1)
      ;

}
```

XILINX®

Appendix C

Glossary

B

BBD file

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

BFL

Bus Functional Language.

BFM

Bus Functional Model.

BIT File

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

BitInit

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

block RAM

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

BMM file

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

BSB

Base System Builder. A wizard for creating a complete EDK design. BSB is also the file type used in the Base System Builder.

BSP

See Standalone BSP.

D

DCM

Digital Clock Manager

DCR

Device Control Register.

DLMB

Data-side Local Memory Bus. See also: LMB

DMA

Direct Memory Access.

DOPB

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

DRC

Design Rule Check.

E

EDIF file

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

EDK

Embedded Development Kit.

ELF file

Executable Linked Format file.

EMC

Enclosure Management Controller.

EST

Embedded System Tools.

F

FATfs (XilFATfs)

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

FPGA

Field Programmable Gate Array.

FSL

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

G

GDB

GNU Debugger.

GPIO

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

H

Hardware Platform

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

HDL

Hardware Description Language.

I

IBA

Integrated Bus Analyzer.

IDE

Integrated Design Environment.

ILA

Integrated Logic Analyzer.

ILMB

Instruction-side Local Memory Bus. See also: LMB

IOPB

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

IPIC

Intellectual Property Interconnect.

IPIF

Intellectual Property Interface.

ISA

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

ISC

Interrupt Source Controller.

ISS

Instruction Set Simulator.

J

JTAG

Joint Test Action Group.

L

Libgen

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

LibXil Standard C Libraries

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

LibXil File

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

LibXil Net

The network library for embedded processors.

LibXil Profile

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

LMB

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

M

MDD file

Microprocessor Driver Description file.

MDM

Microprocessor Debug Module.

MFS

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

MHS file

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

MLD file

Microprocessor Library Definition file.

MPD file

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

MSS file

Microprocessor Software Specification file.

MVS file

Microprocessor Verification Specification file.

N

NCF file

Netlist Constraints file.

NGC file

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

NGD file

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

NGO File

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

NPL File

Xilinx® Integrated Software Environment (ISE™) Project Navigator project file.

O

OCM

On Chip Memory.

OPB

On-chip Peripheral Bus.

P

PACE

Pinout and Area Constraints Editor.

PAO file

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

PBD file

Processor Block Diagram file.

Platgen

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

PLB

Processor Local Bus.

PROM

Programmable ROM.

PSF

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

S

SDF file

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

SDK

Software Development Kit.

Simgen

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

Software Platform

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

SPI

Serial Peripheral Interface.

Standalone BSP

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

SVF File

Serial Vector Format file.

U

UART

Universal Asynchronous Receiver-Transmitter.

UCF

User Constraints File.

V

VHDL

VHSIC Hardware Description Language.

VP

Virtual Platform.

VPgen

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

X

XBD File

Xilinx® Board Definition file.

XCL

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

Xilkernel

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

XMD

Xilinx® Microprocessor Debugger.

XMK

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

XMP File

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

XPS

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

XST

Xilinx® Synthesis Technology.

Z

ZBT

Zero Bus Turnaround™.