Embedded System Tools Reference Manual

UG1043 (v2019.1) May 22, 2019

Revision History

05/22/2019: Released with Vivado® Design Suite 2019.1 without changes from 2018.3.

Section	Revision
12/05/2018 Version 2018.3	
General updates	Minor editorial changes.
06/14/2016 Version 2016.2	
General updates	Added information about the supported processors and compilers. Added references to Zynq® UltraScale+™ MPSoC related documentation.

XILINX

Table of Contents

Revision History 2

Chapter 1: Embedded System and Tools Architecture Overview

Design Process Overview 6
Vivado Design Suite Overview 8
Software Development Kit 9

Chapter 2: GNU Compiler Tools

Overview 12
Compiler Framework 12
Common Compiler Usage and Options 14
MicroBlaze Compiler Usage and Options 29
Arm Cortex-A9 Compiler Usage and Options 46
Other Notes 48

Chapter 3: Xilinx System Debugger

SDK System Debugger 50
Xilinx System Debugger Command-Line Interface (XSDB) 51

Chapter 4: Flash Memory Programming

Overview 52
Program Flash Utility 53
Other Notes 55

Appendix A: GNU Utilities

General Purpose Utility for MicroBlaze Processors 60
Utilities Specific to MicroBlaze Processors 60
Other Programs and Files 63

Appendix B: Additional Resources and Legal Notices

Xilinx Resources 64
Solution Centers 64
Documentation Navigator and Design Hubs 64

References 65
Training Resources 65
Please Read: Important Legal Notices 66

Chapter 1

Embedded System and Tools Architecture Overview

This guide describes the architecture of the embedded system tools and flows provided in the Xilinx® Vivado® Design Suite for developing systems based on the MicroBlaze™ embedded processor and the Cortex A9, A53 and R5 Arm processors.

The Vivado Design Suite system tools enable you to design a complete embedded processor system for implementation in a Xilinx FPGA device.

The Vivado Design Suite is a Xilinx development system product that is required to implement designs into Xilinx programmable logic devices. Vivado includes:

- The Vivado IP integrator tool, with which you can develop your embedded processor hardware.
- The Software Development Kit (SDK), based on the Eclipse open-source framework, which you can use to develop your embedded software application. SDK is also available as a standalone program.
- Embedded processing Intellectual Property (IP) cores including processors and peripherals.

For links to Vivado documentation and other useful information, see Appendix B, Additional Resources and Legal Notices.

Design Process Overview

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

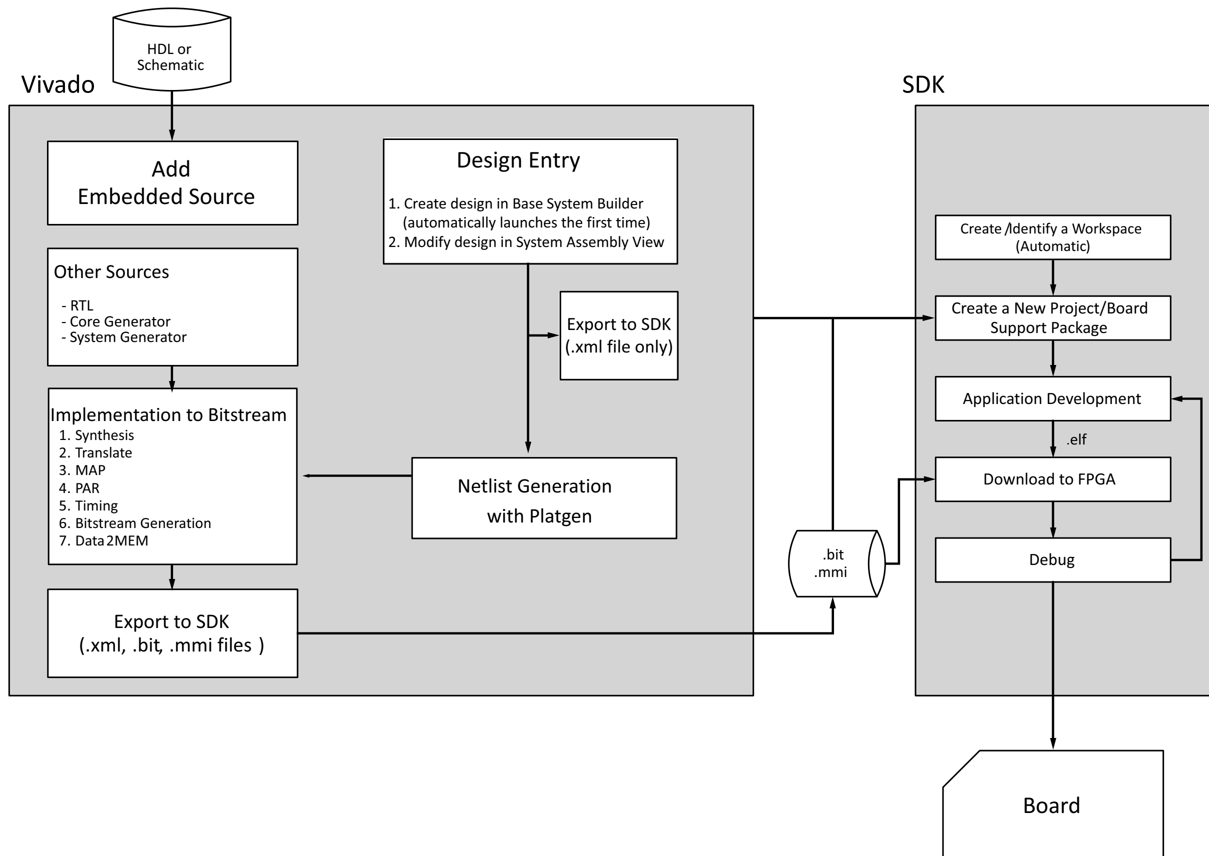

Figure 1-1: **Embedded Design Process Flow**

Hardware Development

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.

The term "Hardware platform" 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.

When the hardware platform description is complete, the hardware platform can be exported for use by SDK.

Software Development

A board support package (BSP) is a collection of software drivers and, optionally, the operating system on which to build your application. The created software image contains only the portions of the Xilinx library you use in your embedded design. You can create multiple applications to run on the BSP.

The hardware platform must be imported into SDK prior to creation of software applications and BSP.

Verification

Vivado provides both hardware and software verification tools. The following subsections describe the verification tools available for hardware and software.

Hardware Verification Using Simulation

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

Software Verification Using Debugging

The following options are available for software verification:

- You can load your design on a supported development board and use a debugging tool to control the target processor.
- You can gauge the performance of your system by profiling the execution of your code.

Device Configuration

When your hardware and software platforms are complete, you then create a configuration bitstream for the target FPGA device.

- For prototyping, download the bitstream along with any software you require to run on your embedded platform while connected to your host computer.
- For production, store your configuration bitstream and software in a non-volatile memory connected to the FPGA.

Vivado Design Suite Overview

An embedded hardware platform typically consists of one or more processors, peripherals and memory blocks, interconnected via processor buses. It also has port connections to the outside world. Each of the processor cores (also referred to as *pcores* or *processor IPs)* 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. IP integrator lets you select from various optional features; consequently, the FPGA needs only implement the subset of functionality required by your application.

Figure 1-2 provides an overview of the Vivado architecture structure of how the tools operate together to create an embedded system.

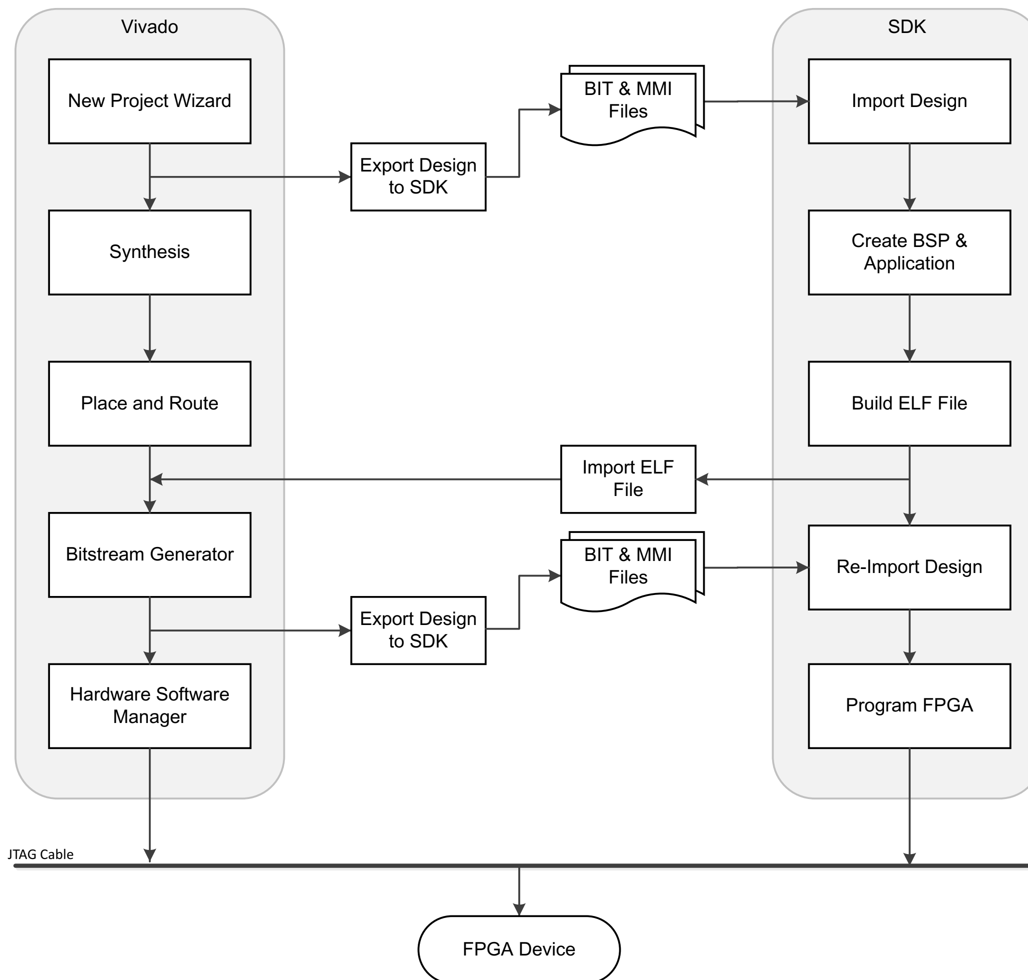

Figure 1-2: **Vivado Design Suite Tools Architecture**

Software Development Kit

The Software Development Kit (SDK) provides a development environment for software application projects. SDK is based on the Eclipse open-source standard. SDK has the following features:

- Can be installed independent of Vivado with a small disk footprint.
- Supports development of software applications on single- or multi-processor systems.
- Imports the Vivado-generated hardware platform definition.
- Supports development of software applications in a team environment.
- Ability to create and configure board support packages (BSPs) for third-party OS.
- Provides off-the-shelf sample software projects to test the hardware and software functionality.
- Has an easy GUI interface to generate linker scripts for software applications, program FPGA devices, and program parallel flash memory.
- Has feature-rich C/C++ code editor and compilation environment.
- Provides project management.
- Configures application builds and automates the make file generation.
- Supplies error navigation.
- Provides a well-integrated environment for seamless debugging and profiling of embedded targets.

For more information about SDK, see the *Software Development Kit (SDK) Help* [Ref 1].

Table 1-1: **Software Development and Verification Tools**

GNU Compiler Tools	Builds a software application based on the platforms created.
Xilinx System Debugger (XSDB)	A command-line interface for hw_server and other TCF servers.
SDK System Debugger	GUI for debugging software on either a simulation model or target device.
Program Flash Utility	Allows you to erase and program on-board serial & parallel flash devices with software and data.

GNU Compiler Tools

GNU compiler tools (GCC) are called for compiling and linking application executables for each processor in the system. Processor-specific compilers are:

- The `mb-gcc` compiler for the MicroBlaze processor.
- The `arm-none-eabi-gcc`, `arm-linux-gnu-eabihf-gcc`, `aarch64-linux-gnu-gcc`, `aarch64-none-eabi-gcc`, `armr5-none-eabi-gcc` compilers for the Arm processor.

As shown in the embedded tools architectural overview (Figure 1-2, page 8):

- The compiler reads a set of C-code 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 that you have provided.

Refer to Chapter 2, "GNU Compiler Tools,", and Appendix A, GNU Utilities for more information about GNU compiler tools and utilities.

Xilinx System Debugger (XSDB)

Xilinx System Debugger (XSDB) is a command-line interface for hw_server and other TCF servers. XSDB interacts with the TCF servers, thereby providing a full advantage of the features supported by the TCF servers.

XSDB supports programming FPGAs, downloading and running programs on targets and other advanced features. Refer to Chapter 3, Xilinx System Debugger for more information.

SDK System Debugger

The Xilinx-customized System Debugger is derived from open-source tools and is integrated with Xilinx SDK. The SDK debugger enables you to see what is happening to a program while it executes. You can set breakpoints or watchpoints to stop the processor, step through program execution, view the program variables and stack, and view the contents of the memory in the system.

The SDK debugger supports debugging through Xilinx System Debugger (XSDB). Refer to Chapter 3, Xilinx System Debugger for more information.

Note: The GDB flow is deprecated and will not be available for future devices. The System Debugger is intended for use only with the Arm over Digilent cable.

Program Flash Utility

The Program Flash utility is designed to be generic and targets a wide variety of flash hardware and layouts. See Chapter 4, "Flash Memory Programming."

Chapter 2

GNU Compiler Tools

Overview

The Vivado® Design Suite includes the GNU compiler collection (GCC) for the MicroBlaze™ processor and the Cortex A9 processor.

- The Vivado 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 Cortex A9 Arm processor tools include `arm-xilinx-eabi-gcc` and `arm-xilinx-eabi-g++` compilers, `arm-xilinx-eabi-as` assembler, and `arm-xilinx-eabi-ld` linker.
- The toolchains also include the C, Math, GCC, and C++ standard libraries.

The compiler also uses the common binary utilities (referred to as binutils), such as an assembler, a linker, and object dump. The MicroBlaze and Arm compiler tools use the GNU binutils based on GNU version 2.16 of the sources. The concepts, options, usage, and exceptions to language and library support are described Appendix A, "GNU Utilities."

Compiler Framework

This section discusses the common features of the MicroBlaze andCortex A9Arm processor compilers. Figure 2-1 displays the GNU tool flow.

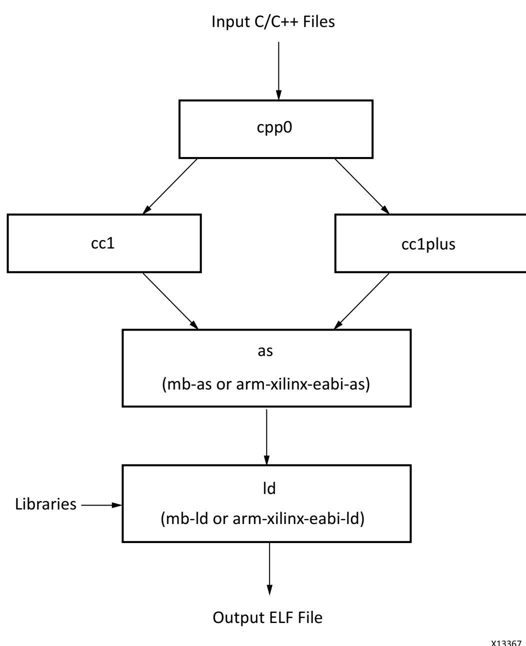

Figure 2-1: **GNU Tool Flow**

The GNU compiler is named `mb-gcc` for MicroBlaze and `arm-xilinx-eabi-gcc` for Arm Cores. 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, `arm-xilinx-eabi-as` for Arm.
  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, `arm-xilinx-eabi-ld for Arm`.
  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.

Executable options are described in:

- Commonly Used Compiler Options: Quick Reference, page 18
- Linker Options, page 23
- MicroBlaze Compiler Options: Quick Reference, page 30
- MicroBlaze Linker Options, page 37
- Arm Cortex-A9 Compiler Usage and Options, page 46

Note: From this point forward the references to GCC in this chapter refer tothe MicroBlaze compiler, `mb-gcc`,and references to G++ refer tothe MicroBlaze C++ compiler, `mb-g++`.

Common Compiler Usage and Options

Usage

To use the GNU compiler, type:

```
<Compiler_Name> options files...
```

where *<Compiler_Name>* is `mb-gcc` or `arm-xilinx-eabi-gcc`. To compile C++ programs, you can use the `mb-g++` or `arm-xilinx-eabi-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 **`arm-xilinx-eabi-ld)`** is used.

The default extensions for each of these types are listed in Table 2-1. 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 Vivado installation directory. When using the G++ compiler, the `libsupc++.a` and `libstdc++.a` files 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.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 2-1 lists the valid extensions and the corresponding file types. The GCC wrapper calls the appropriate lower level tools by recognizing these file types.

Table 2-1: **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 2-2 lists the libraries necessary for the mb_gcc and arm-xilinx-eabi-gcc compilers.

Table 2-2: **Libraries Used by the Compilers**

Library	Particular
libxil.a	Contain drivers, software services (such as XilMFS) and initialization files developed for the Vivado 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.

Libraries are linked in automatically 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 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 2-1, page 15. 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 Vivado drivers and libraries follow these conventions 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. A link to the document is provided in the Appendix B, "Additional Resources and Legal Notices."

- 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 the GNU online documentation.

Commonly Used Compiler Options: Quick Reference

The summary below lists compiler options that are common to the compilers for MicroBlaze and Arm processors.

Note: The compiler options are case sensitive.

To jump to a detailed description for a given option, click its name in the table below.

General Options		Library Search Options
`-E`	`-Wp,option`	`-l libraryname`
`-S`	`-Wa,option`	`-L Lib Directory`
`-c`	`-Wl,option`	
`-g`	`-help`	**Header File Search Option**
`-gstabs`	`-B directory`	`-I Directory Name`
`-On`	`-L directory`	
`-v`	`-I directory`	**Linker Options**
`-save-temps`	`-l library`	`-defsym _STACK_SIZE=value`
`-o filename`		`-defsym _HEAP_SIZE=value`

General Options

-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

This option adds DWARF2-based debugging information to the output file. The debugging information is required by the GNU debugger, `mb-gdb or arm-xilinx-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 STABS-based debugging information on assembly (`.s`) files and assembly file symbols at the source level. This is an assembler option that is provided directly to the GNU assembler, `mb-as` or `arm-xilinx-eabi-as`. If an assembly file is compiled using the compiler `mb-gcc` or `arm-xilinx-eabi-gcc`, prefix the option with **`-Wa`**.

-On

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

Table 2-3: **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 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 `arm-xilinx-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 the following table.

Table 2-4: **Tool-Specific Options Passed to the Top-Level GCC Compiler**

Option	Tool	Example
-Wp,*option*	Preprocessor	**mb-gcc** -Wp,-D -Wp, MYDEFINE ... Signal the pre-processor to define the symbol MYDEFINE with the **-D MYDEFINE** option.
-Wa,*option*	Assembler	mb-as -Wa, ... Signal the assembler to target the MicroBlaze processor.
-Wl,*option*	Linker	**mb-gcc** -Wl,-M ... Signal 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.

-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* for undefined symbols.

> ***Note:*** The compiler prefixes "lib" to the library name indicated in this command line switch.

Library Search Options

-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} -l project
```

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 available libraries.*

-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 */<dir_name>* directory before searching the header files in the standard path.

Default Search Paths

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

Library Search Procedures

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. `${XILINX_}/gnu/processor/platform/processor-lib/lib`
   b. `${XILINX_}/lib/processor`

   Note: `Processor` indicates `microblaze` for MicroBlaze, or `arm-xilinx-eabi` for Arm.

Header File Search Procedures

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. `${XILINX_}/gnu/processor/platform/lib/gcc/processor/{gcc version}/include`
   b. `${XILINX_}/gnu/processor/platform/processor-lib/include`

Initialization File Search Procedures

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 `${XILINX_}/gnu/processor/platform/processor-lib/lib`.
3. The compilers search the following libraries:
   a. `$XILINX_/gnu/<processor>/platform/<processor-lib>/lib`
   b. `$XILINX_/lib/processor`

   Where:

   - <*processor*> is `microblaze` for MicroBlaze processors, and `arm-xilinx-eabi` for Arm processors
   - <*processor-lib*> is `microblaze-xilinx-elf` for MicroBlaze processors, and `arm-xilinx-eabi` for Arm processors.

   Note: *`platform`* indicates `lin` for Linux, `lin64` for Linux 64-bit and `nt` for Windows Cygwin.

Linker Options

-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 `_STACK_SIZE` variable 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 zero.

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`.

For advanced users: you can generate linker scripts directly from IP integrator.

Memory Layout

The MicroBlaze and Arm 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 reserved memory and I/O 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 2-5 lists the reserved memory locations for MicroBlaze and Arm processors as defined by the processor hardware. For more information on these memory locations, refer to the corresponding processor reference manuals.

For information about the Arm memory map, refer to the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 2].

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 2-5: **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`
Cortex A9 Arm			

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 2-5 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 Arm.

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, page 28 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 frequently 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 can be done at the output section level, or even at the individual function and data level. The resulting ELF can be non-contiguous, that is, there can be "holes" in the memory map. Ensure that you do not use documented reserved locations.

Alternatively, if you are an advanced user and want to modify the default binary data provided by the tools for the reserved memory locations, you can do so. In this case, 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 named based on its associated meaning and purpose. The various standard sections of the object file are displayed in the following figure.

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

Sectional Layout of an object or an Executable File

Section	Description
.text	Text Section
.rodata	Read-Only Data Section
.sdata2	Small Read-Only Data Section
.sbss2	Small Read-Only Uninitialized Data Section
.data	Read-Write Data Section
.sdata	Small Read-Write Data Section
.sbss	Small Uninitialized Data Section
.bss	Uninitialized Data Section
.heap	Program Heap Memory Section
.stack	Program Stack Memory Section

X11005

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

The reserved sections that you would not typically modify include:`.init`, `.fini`, `.ctors`, `.dtors`, `.got`, `.got2`, and `.eh_frame`.

.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 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.

`.sbss2`

This section contains small, read-only un-initialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the r (read) flag and can be mapped to ROM.

`.sbss`

This section contains small un-initialized data of a size less than 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. 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.

.tbss

This section holds uninitialized thread-local data that contribute to the program memory image. This section has the same flags as `.bss`, and it must be mapped to RAM.

`.tdata`

This section holds initialized thread-local data that contribute to the program memory image. This section must be mapped to initialized RAM.

`.gcc_except_table`

This section holds language specific data. This section must be mapped to initialized RAM.

`.jcr`

This section contains information necessary for registering compiled Java classes. The contents are compiler-specific and used by compiler initialization functions. This section must be mapped to initialized RAM.

`.fixup`

This section contains information necessary for doing fixup, such as the fixup page table, and the fixup record table. This section must be mapped to initialized RAM.

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 places section contents contiguously.

You can selectively modify only the starting address of your program by defining the linker symbol `_TEXT_START_ADDR` on MicroBlaze processors, or `START_ADDR` on Arm processors, as displayed in this example:

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100
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_/gnu/<procname>/<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>* =microblaze, *<processor_name>* = microblaze, and *<platform>* = lin or nt.

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

```
compiler -T <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 can be generated for your program from within IP integrator and SDK.

In IP integrator or SDK, 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, as well as the memory mapping for Stack and Heap. When the linker script is generated, it is given as input to GCC automatically when the corresponding application is compiled within IP integrator or SDK.

Linker scripts can be used to assign specific variables or functions to specific memories. This is done through "section attributes" in the C code. Linker scripts can also be used to assign specific object files to sections in memory. These and other features of GNU linker scripts are explained in the GNU linker documentation, which is a part of the online binutils manual. A link to the GNU manuals is supplied in the Appendix B, "Additional Resources and Legal Notices." For a specific list of input sections that are assigned by MicroBlaze processor linker scripts, see "MicroBlaze Linker Script Sections" on page 38.

MicroBlaze Compiler Usage and Options

The MicroBlaze GNU compiler is derived from the standard GNU sources as the Xilinx 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 provides the definition __MICROBLAZE__ automatically. You can use this definition in any conditional code.

MicroBlaze Compiler

The mb-gcc compiler for the Xilinx™ MicroBlaze soft processor introduces 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.

MicroBlaze Compiler Options: Quick Reference

Click an option name below to view its description.

Processor Feature Selection Options	General Program Options
-mcpu=vX.YY.Z	-msmall-divides
-mlittle-endian / -mbig-endian	-mxl-gp-opt
-mno-xl-soft-mul	-mno-clearbss
-mxl-multiply-high	-mxl-stack-check
-mno-xl-multiply-high	**Application Execution Modes**
-mxl-soft-mul	-xl-mode-executable
-mxl-barrel-shift	-xl-mode-bootstrap
-mno-xl-barrel-shift	-xl-mode-novectors
-mxl-pattern-compare	**MicroBlaze Linker Options**
-mno-xl-pattern-compare	-defsym _TEXT_START_ADDR=value
-mhard-float	-relax
-msoft-float	-N
-mxl-float-convert	
-mxl-float-sqrt	

Processor Feature Selection Options

`-mcpu=vX.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.
- `v3.00.a and v4.00.a:` Uses 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots.
- `v5.00.a` and later: Uses 5-stage processor pipeline model. Does not inhibit exception causing instructions from being moved into delay slots.

-mlittle-endian / -mbig-endian

Use these options to select the endianness of the target machine for which code is being compiled. The endianness of the binary object file produced is also set appropriately based on this switch. The GCC driver passes switches to the sub tools (`as`, `cc1`, `cc1plus`, `ld`) to set the corresponding endianness in the sub tool.

The default is `-mbig-endian`.

Note: You cannot link together object files of mixed endianness.

-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 the MicroBlaze processor. 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. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the usage of the multiplier option in MicroBlaze.

-mxl-multiply-high

The MicroBlaze processor has an option to enable instructions that can compute the higher 32-bits of a 32x32-bit multiplication. This option tells the compiler to use these multiply high instructions. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL_HIGH` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is available or not. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3],for more details about the usage of the multiply high instructions in MicroBlaze.

-mno-xl-multiply-high

Do not use multiply high instructions. This option is the default.

-mxl-soft-mul

This option tells the compiler that there is no hardware multiplier unit on MicroBlaze, so every 32-bit multiply operation is replaced by a call to the software emulation routine`__mulsi3`. This option is the default.

-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. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the usage of the hardware divide 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`).

-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. See the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the barrel shifter option in MicroBlaze.

-mno-xl-barrel-shift

This option tells the compiler not to use hardware barrel shift instructions. This option is the default.

-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 Guide,* (UG081) [Ref 3], for more details about the use of the pattern compare option in MicroBlaze.

`-mno-xl-pattern-compare`

This option tells the compiler not to use pattern compare instructions. This is the default.

`-mhard-float`

This option turns on the usage of single precision floating point instructions (`fadd`, `frsub`, `fmul`, and `fdiv`) 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 Guide,* (UG081) [Ref 3], for more details about the use of the hardware floating point unit option in MicroBlaze.

`-msoft-float`

This option tells the compiler to use software emulation for floating point arithmetic. This option is the default.

`-mxl-float-convert`

This option turns on the usage of single precision floating point conversion instructions (`fint` and `flt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

Refer to the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the hardware floating point unit option in MicroBlaze.

`-mxl-float-sqrt`

This option turns on the usage of single precision floating point square root instructions (`fsqrt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

Refer to the *MicroBlaze Processor Reference Guide,* (UG081) [Ref 3], for more details about the use of the hardware floating point unit option in the MicroBlaze processor.

General Program Options

-msmall-divides

This option generates 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.

-mxl-gp-opt

If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), then load or store operations to that address require two instructions.

The MicroBlaze processor ABI offers two global small data areas that can each contain up to 64 Kbytes of data. 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 or store to the small data area. This optimization can be turned on with the `-mxl-gp-opt` command line parameter. Variables of size less 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.

CAUTION! *If this option is being used, it must be provided to both the compile and the link commands of the build process for your program. Using the switch inconsistently can lead to compile, link, or run-time errors.*

-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.

`-mxl-stack-check`

With this option, you can check whether the stack overflows when the program runs.

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.

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-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.*

Position Independent Code

The GNU compiler for MicroBlaze supports the `-fPIC` and `-fpic` switches. These switches enable Position Independent Code (PIC) generation in the compiler. This feature is used by the Linux operating system only for MicroBlaze to implement shared libraries and relocatable executables. The scheme uses a Global Offset Table (GOT) to relocate all data accesses in the generated code and a Procedure Linkage Table (PLT) for making function calls into shared libraries. This is the standard convention in GNU-based platforms for generating relocatable code and for dynamically linking against shared libraries.

MicroBlaze Application Binary Interface

The GNU compiler for MicroBlaze uses the Application Binary Interface (ABI) defined in the *MicroBlaze Processor Reference Guide* (UG081) [Ref 3]. Refer to the ABI documentation for register and stack usage conventions as well as a description of the standard memory model used by the compiler.

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 Guide* (UG081) [Ref 3].

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-as` 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-op codes to ease the task of assembly programming. Table 2-6 lists the supported pseudo-opcodes.

Table 2-6: **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 Options

The mb-ld linker for the MicroBlaze soft processor provides additional options to those supported by the GNU compiler tools. The 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. This can be overridden by using the -defsym _TEXT_START_ADDR option. If this is supplied to mb-gcc compiler, 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.

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.

The MicroBlaze linker uses linker scripts to assign sections to memory. These are listed in the following section.

MicroBlaze Linker Script Sections

Table 2-7 lists the input sections that are assigned by MicroBlaze linker scripts.

Table 2-7: **Section Names and Descriptions**

Section	Description
`.vectors.reset`	Reset vector code.
`.vectors.sw_exception`	Software exception vector code.
`.vectors.interrupt`	Hardware Interrupt vector code.
`.vectors.hw_exception`	Hardware exception vector code.
`.text`	Program instructions from code in functions and global assembly statements.
`.rodata`	Read-only variables.
`.sdata2`	Small read-only static and global variables with initial values.
`.data`	Static and global variables with initial values. Initialized to zero by the boot code.
`.sdata`	Small static and global variables with initial values.
.sbss2	Small read-only static and global variables without initial values. Initialized to zero by boot code.
`.sbss`	Small static and global variable without initial values. Initialized to zero by the boot code.
`.bss`	Static and global variables without initial values. Initialized to zero by the boot code.
`.heap`	Section of memory defined for the heap.
`.stack`	Section of memory defined for the stack.

Tips for Writing or Customizing Linker Scripts

Keep the following points in mind when writing or customizing your own linker script:

- Ensure that the different vector sections are assigned to the appropriate memories as defined by the MicroBlaze hardware.
- Allocate space in the .bss section for stack and heap. Set the _stack variable to the location after _STACK_SIZE locations of this area, and the _heap_start variable to the next location after the _STACK_SIZE location. Because the stack and heap need not be initialized for hardware as well as simulation, define the _bss_end variable after the .bss and COMMON definitions.

  Note: The .bss section boundary does not include either stack or heap.
- Ensure that the variables _SDATA_START__ , _SDATA_END__, SDATA2_START, _SDATA2_END__, _SBSS2_START__ , _SBSS2_END__, _bss_start, _bss_end, _sbss_start, and _sbss_end are defined to the beginning and end of the sections sdata, sdata2, sbss2, bss, and sbss respectively.
- ANSI C requires that all uninitialized memory be initialized to startup (not required for stack and heap). The standard CRT that is provided assumes a single .bss section that is initialized to zero. If there are multiple .bss sections, this CRT will not work. You should write your own CRT that initializes all the .bss sections.

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. Start up files typically do the following:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers. Refer to Table 2-8, page 40 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 2-8 lists the register names, values, and descriptions in the C-Runtime files.

Table 2-8: **Register Initialization in 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 following subsections describe 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 runtime 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. 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.

`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. The `_crtinit` routine is also the wrapper that invokes the `main` procedure. Before invoking the main procedure, it may invoke other initialization functions. The `_crtinit` routine is supplied by the startup files described below.

`crtinit.o`

This default, second stage, C 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, and 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, and 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, and 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 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 Vivado. 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_>/SDK/<version>/data/embeddedsw/lib/microblaze/src/ directory, where <XILINX_> is the Vivado installation path and <version> is the release version of the SDK.

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 the `-nostartfiles` on the final compile line.

Note: 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 might 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 might want 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 require that code. You might be able to save approximately 220 bytes of code space by making the following 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 `xcrtinit.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 39.

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 Vivado. The CPU driver for MicroBlaze copies over the correct version, based on the hardware configuration of MicroBlaze. To manually select the library version that you would like to use, look in the following folder:

$XILINX_/gnu/microblaze/<*platform*>/microblaze-xilinx-elf/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.

Table 2-9 shows the current encodings used and the configuration of the library specified by the encodings.

Table 2-9: **Encoded Library Filenames on Compiler Flags**

Encoding	Description
_bs	Configured for barrel shifter.
_m	Configured for hardware multiplier.
_p	Configured for pattern comparator.

Of special interest are the math library files (libm*.a). The C standard requires the common math library functions (sin() and cos(), for example) to use double-precision floating point arithmetic. However, double-precision floating point arithmetic may not be able to make full use of the optional, single-precision floating point capabilities in available for MicroBlaze.

The Newlib math libraries have alternate versions that implement these math functions using single-precision arithmetic. These single-precision libraries might be able to make direct use of the MicroBlaze processor hardware Floating Point Unit (FPU) and could therefore perform better.

If you are sure that your application does not require standard precision, and you want to implement enhanced performance, you can manually change the version of the linked-in library.

By default, the CPU driver copies the double-precision version (libm_*_fpd.a) of the library into your IP integrator project.

To get the single precision version, you can create a custom CPU driver that copies the corresponding libm_*_fps.a library instead. Copy the corresponding libm_*_fps.a file into your processor library folder (such as microblaze_0/lib) as libm.a.

When you have copied the library that you want to use, rebuild your application software project.

Thread Safety

The MicroBlaze processor C and math libraries distributed with Vivado are not built to be used in a multi-threaded environment. Common C library functions such as `printf()`, `scanf()`, `malloc()`, and `free()` are *not* thread-safe and will cause unrecoverable errors in the system at run-time. Use appropriate mutual exclusion mechanisms when using the Vivado libraries in a multi-threaded environment.

Command Line Arguments

The MicroBlaze processor programs cannot take 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 Microprocessor Software Specification (MSS) files. These definitions automatically add the attributes to the interrupt handler functions.

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));
```

fast_interrupt

The MicroBlaze compiler provides the attribute `fast_interrupt`, which is similar to the interrupt_handler attribute. On fast interrupt, MicroBlaze jumps to the interrupt routine address instead jumping to the fixed address 0x10.

Unlike a normal interrupt, when the attribute `fast_interrupt` is used on a C function, MicroBlaze saves only minimal registers.

```
void function_name () __attribute__ ((fast_interrupt));
```

Table 2-10: **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`.
fast_interrupt	This attribute is similar to `interrupt_handler`, but it jumps directly to the interrupt routine address instead of jumping to the fixed address 0x10.

Arm Cortex-A9 Compiler Usage and Options

Arm targets can be complied using Sourcery CodeBench Lite for Xilinx EABI.

Sourcery CodeBench contains the complete GNU Toolchain including all of the following components:

- CodeSourcery Common Startup Code Sequence
- CodeSourcery Debug Sprite for Arm
- GNU Binary Utilities (Binutils)
- GNU C Compiler (GCC)
- GNU C++ Compiler (G++)
- GNU C++ Runtime Library (Libstdc++)
- GNU Debugger (GDB)
- Newlib C Library

Usage

Compiling

```
arm-xilinx-eabi-gcc -c file1.c -I<include_path> -o file1.o
arm-xilinx-eabi-gcc -c file2.c -I<include_path> -o file2.o
```

Linking

```
arm-xilinx-eabi-gcc -Wl,-T -Wl,lscript.ld -L<libxil.a path> -o "App.elf"file1.o
file2.o -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
```

For descriptions of flags used in the commands above, refer to the compiler help, using any of the following commands:

- arm-xilinx-eabi-gcc --help
- arm-xilinx-eabi-gcc -v --help
- arm-xilinx-eabi-gcc --target-help

Compiler Options

Other GNU compiler options that can be applied using Arm-related flags can be found on GNU Website: http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html. These flags can be used in the steps above, as required.

All the Arm GCC compiler options are listed at the link above. However, actual support depends on the target in use (Arm Cortex A9 in this case) and on the compiler toolchain.

For example:

The Sourcery CodeBench Lite for Xilinx EABI does not support `-mhard-float (-mfloat-abi=hard)`. Only soft and `softfp` floating point options are supported.

For more information on the toolchain, refer to the documentation available in the SDK installation path:

```
<Xilinx_Vivado_Installation_Path>\SDK\<2016.1>\gnu\arm\nt\share\doc
```

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 Vivado 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.

Note: The C++ standard library is not built for a multi-threaded environment. Common C++ features such as `new` and `delete` are not thread-safe. Please use caution when using the C++ standard library in an operating system environment.

For more information on the GNU C++ standard library, refer to the documentation available on the GNU website.

Position Independent Code (Relocatable Code)

The MicroBlaze processor compilers support the `-fPIC` switch to generate position independent code.

While both these features are supported in the Xilinx compiler, they are not supported by the rest of the libraries and tools, because Vivado only provides a standalone platform. No loader or debugger can interpret 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. Third-party OS vendors could use these features as a standard in their distribution and tools.

Other Switches and Features

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

Xilinx System Debugger

Xilinx® System Debugger enables you to see what is happening to a program while it executes. You can set breakpoints or watchpoints to stop the processor, step through program execution, view the program variables and stack, and view the contents of the memory in the system.

Xilinx System Debugger supports debugging through SDK and Command-line interface (CLI).

SDK System Debugger

SDK System Debugger, uses the Xilinx hw_server as the underlying debug engine. SDK translates each user interface action into a sequence of TCF commands. It then processes the output from system Debugger to display the current state of the program being debugged. It communicates to the processor on the hardware using Xilinx hw_server. The debug workflow is described in the following diagram:

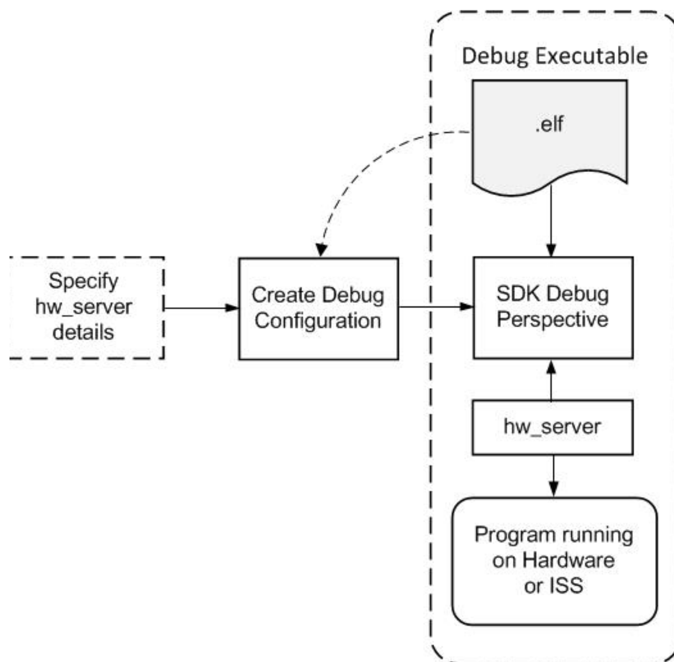

Figure 3-1: **Debug Workflow**

The workflow is made up of the following components:

- **Executable ELF File:** To debug your application, you must use an Executable and Linkable Format (ELF) file compiled for debugging. The debug ELF file contains additional debug information for the debugger to make direct associations between the source code and the binaries generated from that original source.
- **Debug Configuration:** In order to launch the debug session, you must create a debug configuration in SDK. This configuration captures options required to start a debug session, including the executable name, processor target to debug, and other information.
- **SDK Debug Perspective:** Using the Debug perspective, you can manage the debugging or running of a program in the Workbench. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

You can repeat the cycle of modifying the code, building the executable, and debugging the program in SDK.

Note: If you edit the source after compiling, the line numbering will be out of step because the debug information is tied directly to the source. Similarly, debugging optimized binaries can also cause unexpected jumps in the execution trace.

For more details on SDK System Debugger, see the *Software Development Kit (SDK) Help* [Ref 1].

Xilinx System Debugger Command-Line Interface (XSDB)

Xilinx System Debugger Command-line Interface (XSDB) provides a user-friendly, interactive, and scriptable command line interface to Xilinx hw_server and other incarnations of TCF servers used by Xilinx. You can take full advantage of the features supported by the TCF servers as XSDB interacts with the TCF servers.

XSDB is designed to do the following:

- Allow interaction with the whole system
- Support a software engineer's view of both hardened and programmable logic
- Provide performance measurement
- Integrate with hw_server and other incarnations of TCF servers.

For a detailed list of the XSDB commands and their explanation, see the *Software Development Kit (SDK) Help* [Ref 1].

Flash Memory Programming

Overview

Program Flash Utility is used to erase and program flash memories on the board. Some other options include blank check and verify, which are useful to verify the erase and program features. Blank Check, if enabled, reads the content from the flash and checks whether the flash part is blank or not. Similarly, the verify feature, if enabled, reads back and compares the data read with the data programmed, to check if the data was written properly.

Zynq Devices

Program Flash utility supports programming of QSPI, NAND & NOR types of flashes. QSPI can used in different configurations such as QSPI Single, QSPI Dual Parallel and QSPI Dual Stacked. FSBL file has to be provided in case of NAND & NOR types.

You can program boot images created from Bootgen. Bootgen stitches the components like First Stage Boot Loader (FSBL), bitstream (to configure the PL part of Zynq®), the applications, RTOS and other data files.

When the processor comes out of reset in case of Zynq, the control is with BootROM, which copies the FSBL from the flash to the on chip memory and hands over the control to it. The FSBL starts executing, which then copies the bitstream from flash and configures the PL. Once the PL is configured, the FSBL copies the next partition, say, an application from the flash to DDR, and hands over the control to the application. The application starts executing. In order to load the Linux, U-boot can be used as one more partition

Other Devices

The flashes are broadly categorized into Parallel Flash (BPI) and Serial Flash (SPI). Both the SPI and BPI flashes are available from various makes such as Micron, Spansion etc. 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 executable or bootable images of applications is the most common use case. When the processor in your design comes out of reset, it starts executing code stored in block RAM at the processor reset location. Typically, block RAM size is only a few kilobytes or so and is too small to accommodate your entire software application image. You can store your software application image (typically, a few megabytes-worth of data) in flash memory. A small bootloader is then designed to fit in block RAM. The processor executes the bootloader on reset, which then copies the software application image from flash into external memory. The bootloader then transfers control to the software application to continue execution.

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 Motorola S-record (SREC). This keeps the bootloader smaller and more simple.

Program Flash Utility

Program Flash is a command-line utility which allows you to erase and program on-board serial & parallel flash devices with software and data.

Usage

```
program_flash <flash options> <cable device options>
```

Flash Options

Option	Description
`-f <image file>`	Image to be written onto the flash memory (bin/mcs only)
`-offset <address>`	Offset within the flash memory at which the image should be written.
`-no_erase`	Do not erase the flash memory before programming
`-erase_only`	Erases the flash as per the size of the image file
`-blank_check`	Check if the flash memory is erased
`-verify`	Check if the flash memory is programmed correctly
`-fsbl <fsbl file>`	For NAND & NOR flash types only (Zynq only)
`-erase_sector <size>`	For flashes whose erase sector is other than 64KB (size in bytes)

Option	Description

`-flash_type <type>`	Supported flash memory types:
	• For Zynq devices
	  ◦ qspi_single
	  ◦ qspi_dual_parallel
	  ◦ qspi_dual_stacked
	  ◦ nand_8
	  ◦ nand_16
	  ◦ nor
	• For other devices
	Use the `-partlist` command line option to list all the flash types.

`-partlist <bpi|spi> <micron|spansion>`	Lists all the flash parts for other (non-Zynq) devices • `program_flash -partlist` - lists all flashes • `program_flash -partlist bpi micron` - lists all Micron BPI flashes • `program_flash -partlist spi spansion` - lists Spansion SPI flashes

Cable & Device Options

Option	Description
`-cable type <type of cable> esn <cable esn>` `url <URL>`	• `type <type of cable>` - Specify the cable type (xilinx_tcf) • `esn <cable esn>` - Specify the Electronic Serial Number (ESN) of the USB cable connected to the host machine. Use this option to uniquely identify a USB cable when multiple cables are connected to the host machine. • `url <URL>` - URL description of hw_server/TCF agent.
`-debugdevice deviceNr <device position in jtag chain>`	`-deviceNr` - Position in the JTAG chain of the device. The device position number starts from 1.

Other Notes

Supported Flash Parts for Non-Zynq Devices

The following table lists all the flash parts that are supported for non-Zynq devices. The part name information is passed using the `-flash_type` command line option. The list contains the flashes of type BPIx8, BPIx16 and SPI from Micron & Spansion. The `-partlist` command-line option can be used to filter out the flashes based on types (BPI/SPI) or manufacturer (Spansion/Micron).

Table 4-1: **Supported Flash parts for non-Zynq devices**

S.No.	Manufacturer	Part Name (-flash type)
1:	Spansion	s29gl128p-bpi-x16
2:	Spansion	s29gl256p-bpi-x16
3:	Spansion	s29gl512p-bpi-x16
4:	Spansion	s29gl01gp-bpi-x16
5:	Spansion	s29gl128s-bpi-x16
6:	Spansion	s29gl256s-bpi-x16
7:	Spansion	s29gl512s-bpi-x16
8:	Spansion	s29gl01gs-bpi-x16
9:	Spansion	s29gl128p-bpi-x8
10:	Spansion	s29gl256p-bpi-x8
11:	Spansion	s29gl512p-bpi-x8
12:	Spansion	s29gl01gp-bpi-x8
13:	Micron	28f640p30t-bpi-x16
14:	Micron	28f640p30b-bpi-x16
15:	Micron	28f128p30t-bpi-x16
16:	Micron	28f128p30b-bpi-x16
17:	Micron	28f256p30t-bpi-x16
18:	Micron	28f256p30b-bpi-x16
19:	Micron	28f512p30t-bpi-x16
20:	Micron	28f512p30e-bpi-x16
21:	Micron	28f512p30b-bpi-x16
22:	Micron	28f00ap30t-bpi-x16
23:	Micron	28f00ap30e-bpi-x16
24:	Micron	28f00ap30b-bpi-x16
25:	Micron	28f00bp30e-bpi-x16

Table 4-1: **Supported Flash parts for non-Zynq devices**

S.No.	Manufacturer	Part Name (-flash type)
26:	Micron	28f640p33t-bpi-x16
27:	Micron	28f640p33b-bpi-x16
28:	Micron	28f128p33t-bpi-x16
29:	Micron	28f128p33b-bpi-x16
30:	Micron	28f256p33t-bpi-x16
31:	Micron	28f256p33b-bpi-x16
32:	Micron	28f512p33t-bpi-x16
33:	Micron	28f512p33e-bpi-x16
34:	Micron	28f512p33b-bpi-x16
35:	Micron	28f00ap33t-bpi-x16
36:	Micron	28f00ap33e-bpi-x16
37:	Micron	28f00ap33b-bpi-x16
38:	Micron	28f128g18f-bpi-x16
39:	Micron	mt28gu256aax1e-bpi-x16
40:	Micron	mt28gu512aax1e-bpi-x16
41:	Micron	mt28gu01gaax1e-bpi-x16
42:	Micron	28f064m29ewh-bpi-x16
43:	Micron	28f064m29ewl-bpi-x16
44:	Micron	28f064m29ewt-bpi-x16
45:	Micron	28f064m29ewb-bpi-x16
46:	Micron	28f128m29ew-bpi-x16
47:	Micron	28f256m29ew-bpi-x16
48:	Micron	28f512m29ew-bpi-x16
49:	Micron	28f00am29ew-bpi-x16
50:	Micron	28f00bm29ew-bpi-x16
51:	Micron	28f064m29ewh-bpi-x8
52:	Micron	28f064m29ewl-bpi-x8
53:	Micron	28f064m29ewt-bpi-x8
54:	Micron	28f064m29ewb-bpi-x8
55:	Micron	28f128m29ew-bpi-x8
56:	Micron	28f256m29ew-bpi-x8
57:	Micron	28f512m29ew-bpi-x8
58:	Micron	28f00am29ew-bpi-x8
59:	Micron	28f00bm29ew-bpi-x8
60:	Spansion	s70gl02gp-bpi-x16

Table 4-1: **Supported Flash parts for non-Zynq devices**

S.No.	Manufacturer	Part Name (-flash type)
61:	Spansion	s70gl02gs-bpi-x16
62:	Spansion	s25fl032p-spi-x1_x2_x4
63:	Spansion	s25fl064p-spi-x1_x2_x4
64:	Spansion	s25fl132k-spi-x1_x2_x4
65:	Spansion	s25fl164k-spi-x1_x2_x4
66:	Spansion	s25fl128sxxxxxx0-spi-x1_x2_x4
67:	Spansion	s25fl128sxxxxxx1-spi-x1_x2_x4
68:	Spansion	s25fl256sxxxxxx0-spi-x1_x2_x4
69:	Spansion	s25fl256sxxxxxx1-spi-x1_x2_x4
70:	Spansion	s25fl512s-spi-x1_x2_x4
71:	Micron	mt25qu512-spi-x1_x2_x4
72:	Micron	mt25qu512-spi-x1_x2_x4_x8
73:	Micron	mt25ql512-spi-x1_x2_x4
74:	Micron	mt25ql512-spi-x1_x2_x4_x8
75:	Micron	mt25ql01g-spi-x1_x2_x4
76:	Micron	mt25ql01g-spi-x1_x2_x4_x8
77:	Micron	mt25ql02g-spi-x1_x2_x4
78:	Micron	mt25ql02g-spi-x1_x2_x4_x8
79:	Micron	mt25qu01g-spi-x1_x2_x4
80:	Micron	mt25qu01g-spi-x1_x2_x4_x8
81:	Micron	mt25qu02g-spi-x1_x2_x4
82:	Micron	mt25qu02g-spi-x1_x2_x4_x8
83:	Micron	n25q128-3.3v-spi-x1_x2_x4
84:	Micron	n25q128-1.8v-spi-x1_x2_x4
85:	Micron	n25q256-3.3v-spi-x1_x2_x4
86:	Micron	n25q256-1.8v-spi-x1_x2_x4_x8
87:	Micron	n25q256-1.8v-spi-x1_x2_x4
88:	Micron	n25q32-3.3v-spi-x1_x2_x4
89:	Micron	n25q32-1.8v-spi-x1_x2_x4
90:	Micron	n25q64-3.3v-spi-x1_x2_x4
91:	Micron	n25q64-1.8v-spi-x1_x2_x4

Conversion of ELF Files to SREC for Bootloader Applications

You can use the `mb-objcopy` utility to create SREC format files from ELF files. The SREC format applications can be stored in flash at some particular offsets. The SREC boot loader can read these applications, load them and execute. For example, navigate to the folder containing the `myexecutable.elf` file and execute the following:

```
mb-objcopy -O srec myexecutable.elf myexecutable.srec
```

This creates an SREC file that you can then use as appropriate. The `mb-objcopy` utility is a GNU binary that ships with the SDK.

Conversion of SREC/ELF/BIT files to BIN/MCS files for programming

You can use Xilinx Bootgen utility to create the BIN/MCS files from various other files.

```
bootgen -arch fpga -image <input.bif> -o <output.bin/mcs> -interface <options>
```

Bootgen Options

Option	Description
`-image <input.bif>`	Input boot image format file contains info regarding the input file.
`-o <output.bin/mcs>`	The output file path and format • -o output.bin - BIN file created with name output • -o output.mcs - MCS file created with name output
`-interface <options>`	Interface to program and boot from the flash • spi • bpix8 • bpix16 • smapx8 • smapx16 • smapx32

Examples

1. Converting ELF file to BIN file.

   ```
   bootgen -arch fpga -image elf_bin_all.bif -o boot.bin -interface spi
   ```

   Where the contents of the `elf_bin_all.bif file` are as follows:

   ```
   image:
    {
    hello.elf
    }
   ```

2. Converting SREC file to BIN file.

```
bootgen -arch fpga -image srec_bin_all.bif -o boot.bin -interface spi
```

Where the contents of the srec_bin_all.bif file are as follows:

```
image:
 {
 hello.elf.srec
 }
```

3. Converting BIT file to BIN file

```
bootgen -arch fpga -image bit_bin_all.bif -o boot.bin -interface spi
```

Where the contents of the bit_bin_all.bif are as follows:

```
image:
 {
 system.bit
}
```

Creating images for Zynq devices

Xilinx Bootgen is used to create images for Zynq devices. Various components are stitched together to create a boot image. Optionally, the components can be encrypted, authenticated, checksum. There are various options to create boot images.

For more information, refer to *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 8].

Appendix A

GNU Utilities

This appendix describes the GNU utilities available for use with the Vivado® Design Suite.

General Purpose Utility for MicroBlaze Processors

cpp

Pre-processor for C and C++ utilities. The preprocessor is invoked automatically by GNU Compiler Collection (GCC) 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.

Note: The `gcov` utility is not supported by IP integrator or SDK, but is provided as is for use if you want to roll your own coverage flows.

Utilities Specific to MicroBlaze Processors

Utilities specific to MicroBlaze™ Processors have the prefix "`mb-`," as shown in the following program names.

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`

Appendix B

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter: `docnav`

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

References

The following Vivado® Design Suite guides are referenced in this document.

1. *Software Development Kit (SDK) Help* (UG782)
2. *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585)
3. *MicroBlaze Processor User Guide* (UG081)
4. *Zynq-7000 SoC: Embedded Design Tutorial* (UG1165)

Other Xilinx Documentation

5. *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898) (UG898)
6. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940) (UG940)
7. *Generating Basic Software Platforms Reference Guide* (UG1138)
8. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) (UG821)
9. *Zynq UltraScale+ MPSoC Packaging and Pinouts Product Specification User Guide* (UG1075)
10. *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)
11. *Zynq UltraScale+ MPSoC Software Developer Guide* (UG1137)
12. *Zynq UltraScale+ MPSoC Quick Emulator User Guide* (UG1169)
13. Zynq UltraScale+ MPSoC OpenAMP Getting Started Guide (UG1186)

Additional Resources

14. GNU website: http://www.gnu.org
15. Red Hat Insight website: http://sources.redhat.com/insight.

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Zynq-7000 SoC: Development Tools Overview
2. Zynq-7000 SoC: System Performance Tools Overview
3. Zynq-7000 SoC: Hello World in 5 Minutes

4. Zynq-7000 SoC: Bare Metal Application Development

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2016-2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.