MicroBlaze™ Software Reference Guide

April 2002

XILINX®

MicroBlaze™ Software Reference Guide

The following table shows the revision history for this document.

	Version	Revision
10/15/01	1.9	Initial MDK (MicroBlaze Development Kit) release.
1/14/02	2.1	MDK 2.1 release
3/02	2.2	MDK 2.2 release

Contents

Figures

Device Drivers and Libraries

MicroBlaze Libraries

LibXil Standard C Libraries

LibXil File

LibXil Memory File System (MFS)

LibXil Net

LibXil Driver

Software Specification

Microprocessor Software Specification (MSS) Format

MicroBlaze Address Management

MicroBlaze Application Binary Interface

MicroBlaze Interrupt Management

Microblaze Instruction Set Architecture

MicroBlaze Instruction Set Architecture

Tables

MicroBlaze XMD

Device Drivers and Libraries

MicroBlaze Libraries

LibXil Standard C Libraries

LibXil File

LibXil Memory File System (MFS)

LibXil Net

LibXil Driver

Software Specification

Microprocessor Software Specification (MSS) Format

MicroBlaze Address Management

MicroBlaze Application Binary Interface

MicroBlaze Interrupt Management

Microblaze Instruction Set Architecture

MicroBlaze Instruction Set Architecture

Introduction to the MDT

Definitions

MDK - MicroBlaze Development Kit

MDT - Microprocessor Development Tools

MDT Overview

The Microprocessor Development Tools (MDT) included in the Xilinx MicroBlaze Development Kit (MDK) offer the embedded system designer a rich set of embedded processor system design tools. The MDT set of tools consists of

- Processor platform tailoring utilities
- Software application development tools
- A full featured debug tool chain
- Device drivers and libraries

Platform Tailoring Utilities

The MDT includes the Platform Generator and the Library Generator development platform tailoring utilities.

Using Platform Generator, it is possible to generate the hardware netlist for an entire user defined processor system. The user may specify the configuration of a MicroBlaze processor core, its bus interfaces, and the processor peripheral components that are to be associated with each given bus. Platform Generator tailors each bus components and generates a custom bus which ties the MicroBlaze processor core to its associated peripherals.

Library Generator produces customized device drivers and software function libraries for the given user defined hardware processor system generated by Platform Generator. The address range information of each specified processor peripheral is used to tailor the associated device drivers. In turn, standard C function libraries are tailored to work with the available set of customized device drivers.

Development Tools

Software application development support consists of a complete GNU C Compiler (GCC) and Binary Utilities (binutils) tool suites. These suites allow users to compile, assemble, and link their C code or MicroBlaze assembly language programs. The compiler's code optimizer and code generator have been customized to achieve the best possible performance for applications on the MicroBlaze ISA

The Xilinx Software IDE (XSI) provides an integrated Graphical User Interface (GUI) to create the software specification file for a Microprocessor system. It also provides an editor and a project management interface to create and edit source code.

The Flow Engine (XMF) is used to schedule tool flow (library generator, compiler tools and platform generator) through flow files and option files. XMF is an enhancement to the Xflow utility that is available as part of Xilinx Design Implementation (ISE) tools.

Debug Tools

The MicroBlaze debug tool chain consists of the GNU Debugger (GDB) software debug application, the XMD debug target interface utility, a hardware board interface, and a cycle-accurate Instruction Set Simulator (ISS).

The GDB debug utilities allows users to start the execution of a program, to set initial conditions, to set breakpoints, and to examine the state of the processor and the contents of memory. GDB communicates with XMD to obtain the current program execution information. XMD provides memory contents, processors state and cycle count information to GDB. XMD offers users a choice of two execution targets: a hardware board or a cycle-accurate Instruction Set Simulator. XMD also provides a Tcl interface that supports scripting.

Device Drivers and Libraries

The MDT includes a set of device drivers and library functions. The device drivers offer the user a default software interface to the hardware. Libraries further raise the abstraction level and offer a convenient reference implementation of frequently required functions.

The Xilinx Libraries include support for I/O operations, math computations, string manipulation, memory management functions, a stream based file system, a memory based file system and networking support.

Library Generator aids the user by automatically generating a customized set of device drivers and libraries for a given user defined processor system.

Other Documentation Material

The entire set of GNU manuals can be found online at:

http://www.gnu.org/manual

Document Organization

This MicroBlaze Software Reference Guide consists of the following documents:

- MDT Flow
- Processor Platform Tailoring Utilities
  - MicroBlaze Library Generator
  - MicroBlaze Platform Generator
- Software Application Development Tools
  - MicroBlaze GNU Compiler Tools (includes GNU Binary Utilities)
  - Microprocessor Software IDE (XSI)
  - Flow Engine (XMF)
- Debug Tool Chain
  - MicroBlaze Debug and Simulation
  - MicroBlaze GNU Debugger

  - MicroBlaze XMD
- Device Drivers and Libraries
  - MicroBlaze Libraries
  - LibXil Standard C Libraries
  - LibXil File
  - LibXil Memory File System (MFS)
  - LibXil Net
  - LibXil Driver
- Software Specification
  - Microprocessor Software Specification (MSS) Format
  - MicroBlaze Address Management
  - MicroBlaze Application Binary Interface
  - MicroBlaze Interrupt Management
- MicroBlaze Instruction Set Architecture

XILINX®

Microprocessor Development Tools Flow

Microprocessor Development Tools (MDT) Flow

Mar. 29, 2002

Summary

This document describes the Microprocessor Development Tools (MDT) flow for the 32-bit soft processor, MicroBlaze.

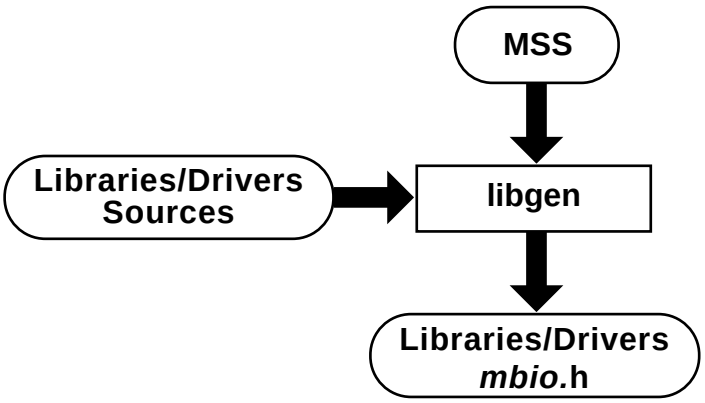
Overview

System design consists of tailoring of the software component, and the hardware component of the embedded processor.

The software component is defined by the MSS (Microprocessor Software Specification) file (see the Microprocessor Software Specification Format documentation for more information). The MSS file defines the standard input/output devices, interrupt handler routines, and other related software features. The MSS file is created by the user.

The hardware component is defined by the MHS (Microprocessor Hardware Specification) file (see the Microprocessor Hardware Specification Format documentation for more information). The MHS file defines the bus architecture, the peripherals, one of six configurations of the MicroBlaze bus interfaces (see the MicroBlaze Bus Interfaces documentation for more information), connectivity of the system, and the address space. The MHS file is created by the user.

Figure 1: **Library Generation**

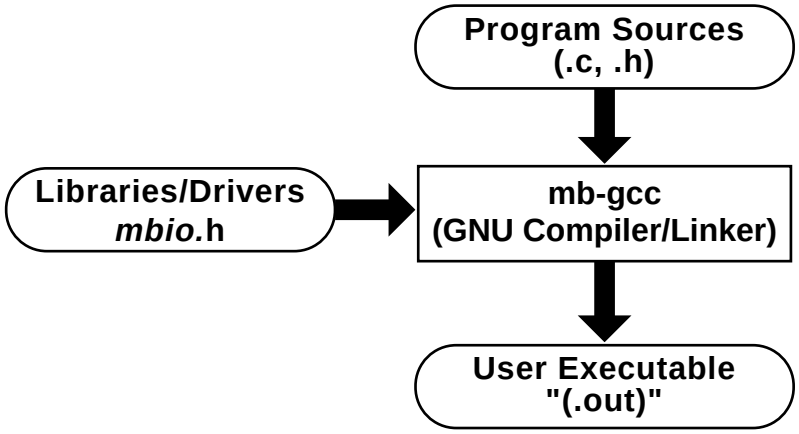


Software tailoring consists of library generation and executable file generation.

Library generation is done with the Library Generator (libgen) tool. Please refer to the MicroBlaze Library Generator and MicroBlaze Libraries documentation for more information. Library Generator will configure the libraries and device drivers with the base addresses of the peripherals of the embedded processor system from an MSS file. Refer to Figure 1 for a flow outline.

After the libraries and the device drivers are configured, an executable image can be generated using the GNU tools. The input into the GNU tools are the libraries/drives that are configured by Library Generator and the user input file. Refer to Figure 2 for a flow outline.

Figure 2: **Executable Generation**



Hardware generation is done with the Platform Generator (platgen) tool. Please refer to the MicroBlaze Platform Generator documentation for more information. Platform Generator will construct the system in the form of hardware netlists (HDL and EDIF files) from an MHS file. Refer to Figure 3 for a flow outline.

Verify Setup

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

MicroBlaze GNU

Ensure that the MicroBlaze GNU tools are in your path.

MicroBlaze on Solaris

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

- **`${MICROBLAZE}/bin/gnu`**
- **`${MICROBLAZE}/bin`**

MicroBlaze on PC

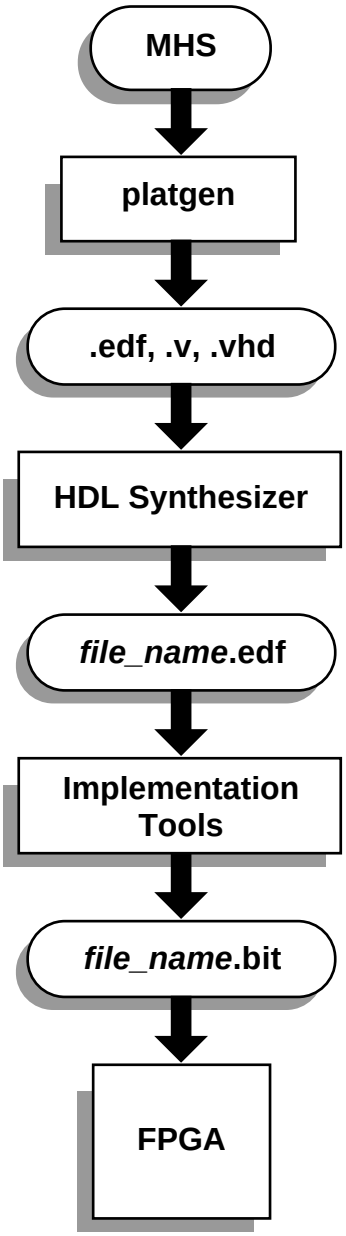
Check the executable search path.

- **`%MICROBLAZE%\bin\gnu`**
- **`%MICROBLAZE%\bin`**

Xilinx Alliance Software

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

Figure 3: **Hardware Flow**



Library Generator

The Library Generator (libgen) tool will configure the libraries and device drivers with the base addresses of the peripherals of the embedded processor system. Please see Figure 1 for a flow outline.

Run the Library Generator as follows:

```
libgen system.mss
```

Please see the MicroBlaze Library Generator and MicroBlaze Libraries documentation for more information.

Program Layout

Please see the MicroBlaze Program Layout documentation on details about address space restrictions.

Running GNU Tools

Given a set of C source files, a MicroBlaze executable is created as follows:

1. To compile and link, run the following:

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

   MicroBlaze compiler searches the **MICROBLAZE**, for libraries and include files.

   If the **MICROBLAZE** environment variable is not set, use the **-L** option to point to the directory containing the libc.a and libm.a libraries.

   ```
   mb-gcc -L<dir1-name> -I<dir2-name> -B<dir3-name> file1.c file2.c
   ```

   MicroBlaze compiler will look into dir1-name for libraries and dir2-name for header files and dir3-name for C runtime libraries.

   A MicroBlaze executable file, *a.out*, is created.

   For including any of the math function, include **-lm** in the command line for **mb-gcc**.

2. For further information on mb-gcc options, run one of the following:
   - mb-gcc --help
   - info gcc

   The latter will give you the info page of Solaris gcc, on which mb-gcc is based. You can also refer to the MicroBlaze GNU Compiler Tools documentation for more details on use of MicroBlaze GNU tools.

Debugging

You can debug your program in software (using a simulator), or on a board which has a Xilinx FPGA loaded with your hardware bitstream. Refer to the XMD documentation for more information.

Debugging Using Hardware: software intrusive

Create your application executable using the following command:

```
mb-gcc -g -xl-mode-xmdstub file1.c file2.c
```

This command creates the MicroBlaze executable *a.out,* linked with the C runtime library crt1.o and starting at physical address 0x400, and with debugging information that can be read by mb-gdb.

If you want to debug your code using a board, you must run Library Generator and Platform Generator with the **-mode xmdstub** option. This initializes the Local Memory (LM) with the **xmdstub** executable. Next, load the bitstream representing your design onto your FPGA. Refer to XMD documentation for more information.

Start xmd server in a new window with the following command:

```
xmd -t hw
```

Load the program in mb-gdb using the command:

```
mb-gdb a.out
```

Click on the “Run” icon and in the mb-gdb Target Selection dialog, choose

- Target: Remote/TCP
- Hostname: localhost
- Port: 1234

Now, mb-gdb’s Insight GUI can be used to debug the program.

Debugging Using A Simulator: non-intrusive

If you want to debug your code using a simulator, compile programs using the following command:

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

This command creates the MicroBlaze executable file, *a.out*, with debugging information that can be accessed by mb-gdb.

Xilinx provides two ways to debug programs in simulation.

1. **Cycle-accurate simulator in XMD:**

Start xmd server in a new window with the following command:

```
xmd -t sim
```

Loading and debugging the program in mb-gdb is done the same way as for xmd in hardware mode described above.

This is the preferred mechanism to debug user programs in simulation

2. **Simple ISA simulator in mb-gdb:**

The xmd server is not needed in this mode. After loading the program in mb-gdb, Click on the "Run" icon and in the mb-gdb Target Selection dialog, choose "**Simulator**".

Use this mechanism only if your program does not attempt to access any peripherals (not even via a print call).

Compiling with Optimization

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

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

Setting the Stack Size

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

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

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

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

Dumping an Object/Executable File

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

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

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

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

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

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

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

Platform Generator

Hardware generation is done with the Platform Generator (platgen) tool and an MHS file. This will construct the embedded processor system in the form of hardware netlists (HDL and EDIF files). Refer to Figure 3 for a flow outline.

Run Platform Generator as follows:

```
platgen system.mhs
```

Please refer to the MicroBlaze Platform Generator documentation for more information.

HDL Synthesis

Hierarchal EDIF netlists are generated in the default mode. This means that each instance of a defined peripheral in the MHS file is synthesized. The default mode leaves the top-level HDL

file untouched allowing you to synthesize it in any synthesizer of your choice. Currently, Platform Generator only supports XST and Synplify.

Platform Generator produces a synthesis vendor specific project file. This is done with **-s** option. The **-s** option builds the synthesis project file for you of the HDL files that were left untouched in default mode (i.e., not specifying the **-flat** option).

Platform Generator does not call the synthesizer for you. It just builds the synthesis project file for you.

If you did specify the **-flat** option, then skip the synthesis step since the top-level is synthesized for you.

The **-i** option disables IO insertion at the top-level, and also generates the HDL component stub for you of the name *system*_stub.vhd or *system*_stub.v. This allows the processor system to be included as a macro in a top-level HDL design. Otherwise, the output from Platform Generator is the top-level netlist.

iSE XST

If Platform Generator was run without the **-flat** option, then a synthesis script file for XST is written. This script can be executed under XST using the following command:

```
xst -ifn system.scr
```

Synplicity Synplify

If Platform Generator was run without the **-flat** option, then a synthesis project file for Synplify is written. This project can be executed under Synplify using the following command:

```
synplify system.prj
```

XILINX®

Processor Platform Tailoring Utilities

Jan. 10, 2002

MicroBlaze Library Generator

Summary

This document describes the Library Generator utility needed for the generation of libraries for the 32-bit soft processor, MicroBlaze. It also describes how the user can define peripherals and associated drivers.

Overview

The Library Generator (libgen) is generally the first tool to run to configure libraries and device drivers. An MSS file is created by the user and given to libgen as input. The MSS file defines the standard input/output devices, interrupt handler routines, and other related software features. Libgen configures libraries and drivers with the information in the MSS input file.

Tool Requirements

The library generator requires a valid MSS file as input. For more information on the MSS file format, please refer the Microprocessor Software Specification documentation.

Tool Usage

The Library Generator is run as follows:

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

Tool Options

The following options are supported in this version:

-h, -help (Help)

This option displays the usage menu and quits.

-v, -ver (Display version information)

This option displays the version number of libgen.

-a, -arch (Architecture family)

This option defines the target architecture family. The options are spartan2, spartan2e, virtex, virtexe or virtex2. The default option is virtex2.

-p, -proj (Specify project directory)

This option specifies the project directory. The default is the current directory. All output files and directories will be generated in the project directory. This project directory is also called **MICROBLAZE_PROJECT** for convenience in the documentation.

-P, -Per_Dir (Specify user peripherals and driver directory)

This option specifies the user peripherals and drivers directory. This is the replacement forthe **XIL_MYPERIPHERALS** environment variable in previous versions. Libgen looks for drivers in the directory **<PER_DIR>/drivers/**

Please refer to later sections for more information.

-m, -mode

This option allows libgen to be run in one of the following modes:

-mode executable: This is the default mode. This mode should be used if the user wants to generate a stand-alone executable program. The EXECUTABLE attribute in the MSS file is used in this mode. Note that in this mode, on board debug support is not available. The MSS file should have the line

```
SET attribute EXECUTABLE = dir/exec_file
```

where **dir** is the directory relative to **MICROBLAZE_PROJECT** directory.

-mode bootstrap: This mode is used when the user wants to use a bootstub executable to load the user program. The BOOTSTRAP attribute in the MSS file is used in this mode. This executable is created in the directory specified by the user in the MSS file. The MSS file should have the line

```
SET attribute BOOTSTRAP = dir/boot_exec_file
```

where **dir** is the directory (relative to **MICROBLAZE_PROJECT**) the bootstub file (**boot_exec_file**) should be created.

-mode xmdstub: This mode is used when user wants to use a debug stub for on-board debug. The XMDSTUB attribute in the MSS file is used in this mode. This attribute specifies that an xmdstub executable has to be created for on board debug by libgen. Libgen generates and overwrites the xmdstub executable if it already exists. The MSS file should have a line such as

```
SET attribute XMDSTUB = dir/debug_exec_file
```

where **dir** is the directory (relative to **MICROBLAZE_PROJECT**) the xmdstub file, **debug_exec_file**, should be created.

-d, -do_not_warn

This option disables printing of warning messages. By default, all warnings are printed.

Output Files

Libgen generates these directories and files in the **MICROBLAZE_PROJECT** directory:

include

The include directory contains C header files that are needed by drivers. The include file **mbio.h** is also created by libgen in this directory. This file defines base addresses of the peripherals in the system and also defines function prototypes.

lib

The lib directory contains **libc.a** and **libm.a** libraries. More information on the libraries can be found in the Libraries documentation.

libsrc

The libsrc directory contains intermediate files and makefiles that are needed to compile the libraries and drivers. The directory contains peripheral specific driver files that are copied from the MicroBlaze and user driver directories. Please refer the Drivers subsection of this document for more information. Note that this directory is overwritten each time libgen is run.

code

The code directory may be used as a repository for MicroBlaze executables.

MSS Attributes

Libgen is sensitive to these attributes in the MSS file (refer Microprocessor Software Specification documentation):

STDIN: This specifies that the peripheral is the standard input. Only peripherals whose MPD file specifies attribute INBYTE=true, can be standard input.

STDOUT: This specifies that the peripheral is the standard output. Only peripherals whose MPD file specifies OUTBYTE=true, can be standard output.

XMDSTUB: This attribute specifies the on board debug executable that needs to be generated.

BOOTSTRAP: This attribute specifies the bootstub executable that should be generated.

DEBUG_PERIPHERAL: This attribute specifies the peripheral in the MHS file to be the debug peripheral. This peripheral will be used for on board debug purposes.

BOOT_PERIPHERAL: This attribute specifies the peripheral in the MHS file to be the boot peripheral. This peripheral will be used for bootstrap purposes.

INT_HANDLER: This attribute defines the interrupt handler function for a peripheral's interrupt signal. This attribute is specified along with the interrupt signal (as defined in the MHS file) in the MSS as shown:

```
SET attribute INT_HANDLER = my_int_hand, Interrupt_signal
```

Please refer to the Interrupt Management documentation for more information on setting up interrupt handlers and handling peripheral interrupts.

DRIVER: This attribute specifies the name of the driver directory to be used for the peripheral requiring a driver.

DRIVER_VER: This attribute specifies the driver version to be used. This version is specified in the following format: **x.yz.a**, where **x,y** and **z** are digits, and **a** is a character. This is translated to the driver directory searched by libgen as follows:

```
MICROBLAZE_PROJECT/drivers/<DRIVER>_vx_yz_a
XIL_MYPERIPHERALS/drivers/<DRIVER>_vx_yz_a
MICROBLAZE/drivers/<DRIVER>_vx_yz_a
```

The **XIL_MYPERIPHERALS** path is specified using the command line option -P to libgen.

Please refer to the Drivers documentation for more information on device drivers and their usage.

MOUNT: This attribute specifies the mount name of a file system. Please see the Libraries documentation for more information.

LIBRARY: This attribute specifies that the file or device is accessed using Xilinx libraries. Please see the Libraries documentation.

Boot and Debug Peripherals

These are peripherals that are specifically used to download bootstub and xmdstub. The attributes BOOT_PERIPHERAL and DEBUG_PERIPHERAL are used for denoting the boot and debug peripheral instances. Libgen uses these attributes in xmdstub and bootstrap modes.

Drivers

Most peripherals require software drivers. The MDK peripherals are shipped with associated drivers. Refer to the Libraries and Device Drivers documentation for more information.

The attribute DRIVER must be used in the MSS file to specify the driver directory for a peripheral. There is no default value for this attribute. The driver directory contains C source and header files and a makefile for the driver. Libgen copies this directory over to the **MICROBLAZE_PROJECT/libsrc** directory and runs **make** for compiling the driver. Refer to the source codes and makefiles in the drivers directory for more information.

Libgen also creates an include file **mbio.h** in the **MICROBLAZE_PROJECT/include** directory. This header file must be included in the driver source files. This file contains peripheral base address definitions and interrupt masks for the peripherals. This file also contains function prototypes and useful defines.

Users can write their own drivers. These drivers must be in a specific directory under **MICROBLAZE_PROJECT/drivers** or **XIL_MYPERIPHERALS/drivers.** The DRIVER attribute allows the user to specify any name for their drivers.

Interrupts and Interrupt Controller

An interrupt controller peripheral must be instantiated if the MHS file has multiple interrupt signals defined. The MHS file should also have a PRIORITY attribute associated with each interrupt signal. In the MSS file, the INT_HANDLER attribute allows an interrupt handler routine to be associated with the interrupt signal. Libgen uses this attribute to configure the interrupt controller handler to call the appropriate peripheral handlers on an interrupt. The functionality of these handler routines is left to the user to implement. If INT_HANDLER attribute is not specified, libgen uses a default dummy handler routine for the peripheral.

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

Please reger to the Interrupt Management documentation for more information.

STDIN and STDOUT Peripherals

Peripherals that handle I/O need drivers to access data. Two files, **`inbyte.c, outbyte.c`** are required in the driver directory if the peripheral instance can be both standard input and standard output. The peripheral instance should be specified as STDIN or STDOUT in the MSS file. The attributes INBYTE=TRUE and OUTBYTE=TRUE have to be included in the MPD file for that peripheral.

If the peripheral is an input only or an output only peripheral, then either the inbyte.c or outbyte.c file must be present, and the corresponding attributes, INBYTE, STDIN or OUTBYTE, STDOUT need be specified.

MicroBlaze Platform Generator

Mar. 29, 2002

Summary

This document describes the Platform Generator utility usage for the 32-bit soft processor, MicroBlaze.

Overview

The hardware component is defined by the MHS (Microprocessor Hardware Specification) file (see the Microprocessor Hardware Specification Format documentation for more information). The MHS file defines the bus architecture, the peripherals, one of six configurations of the MicroBlaze bus interfaces (see the MicroBlaze Bus Interfaces documentation for more information), connectivity of the system, interrupt request priorities, and the address space. The MHS file is created by the user.

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

Tool Requirements

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

Tool Usage

Run Platform Generator as follows:

```
platgen system.mhs
```

Tool Options

The following are the options supported in the current version:

-a (Architecture family)

The **-a** option allows you to target a specific architecture family. The default family is virtex2.

-flat (Generate a flatten EDIF file)

The **-flat** option generates a flatten EDIF file. A synthesis project file is not created.

By default, Platform Generator runs in hierarchal mode. In hierarchal mode, Platform Generates hierarchal EDIF netlists. This means that each instance of a defined peripheral in the MHS file is synthesized. The default mode leaves the top-level HDL file untouched allowing you to synthesize it in any synthesizer of your choice. Currently, Platform Generator only supports XST and Synplify.

-h (Help)

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

-i (Do not insert IOs at top-level)

The **-i** option disables IO insertion at the top-level. This allows processor system to be included as a macro in a top-level design. Otherwise, the output from Platform Generator is the top-level design.

-l (Specify the HDL format)

The **-l** option allows you to specify the HDL format. The default value is vhdl.

Options: [vhdl, verilog]

-p (Specify the Project Directory)

The **-p** option allows you to specify the project directory path. The default is the current directory.

-P (Peripheral repository load path)

The **-P** option allows you to specify the peripheral repository load path.

-mode (Mode)

The -mode option selects the MicroBlaze executable image for memory initialization. The default is executable.

Options: [bootstrap, executable, xmdstub]

The bootstrap mode is used when the user wants to use a bootstub executable to load the user program. The BOOTSTRAP attribute in the MSS file is used in this mode.

The executable mode is used when the user wants to generate a stand-alone executable program. The EXECUTABLE attribute in the MSS file is used in this mode.

The xmdstub mode is used when the user wants to use a debug stub for on-board debug. The XMDSTUB attribute in the MSS file is used in this mode.

-mss (MSS file location)

The **-mss** option allows you to specify the path of the MSS file. The MSS is required if you wish to initialize the generated memories. The MSS file contains location pointers to the programming information.

-s (Generate synthesis vendor project file)

Platform Generator produces a synthesis vendor specific project file. This is done with -s option. The **-s** option builds the synthesis project file for you of the HDL files that were left untouched in default mode (i.e., not specifying the -flat option). The only supported values are 0, 2, and 4. The default value is 2.

Options: [0, 1, 2, 3, 4]

0 - None

1 - Exemplar - Leonardo

2 - iSE - XST - SCR/PRJ file

3 - Synopsys - FPGA Express

4 - Synplicity - Synplify - PRJ file

-sim (Generate simulation models and a simulation vendor project file)

The -sim option generates simulation models of the peripherals in use and a simulation vendor project file. The default value is 0.

Options: [0, 1]

0 - None

1 - ModelSim - DO file

-v (Display version)

The -v option displays the version and quits.

Load Path

By default, OPB peripherals reside at **$MICROBLAZE/hw/coregen** on an UNIX system or **%MICROBLAZE%\hw\coregen** on a PC system. If you want to specify additional directories, you have two options:

1. Current directory (where Platform Generator was launched; not where the MHS resides)
2. Set the Platform Generator **-P** option, or the **XIL_MYPERIPHERALS** environment variable

Platform Generator has a search priority mechanism to locate peripherals.

1. Search current directory
2. Search **$XIL_MYPERIPHERALS/opb_peripherals** (UNIX) or **%XIL_MYPERIPHERALS%\opb_peripherals** (PC)
3. Search **$MICROBLAZE/hw/coregen** (UNIX) or **%MICROBLAZE%\hw\coregen** (PC)

Search areas 1 and 2 have the same underlying directory structure. Search area 3 has the COREgen directory structure.

For search areas 1 and 2, the peripheral name is the name of the root directory. From the root directory, this is the underlying directory structure:

```
data
hdl
netlist
simmodels
```

Output Files

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

```
hdl
implementation
simulation
synthesis
```

HDL Directory

The hdl directory contains the following:

system.vhd

This is the top level HDL file of the processor and its peripherals.

Implementation Directory

The implementation directory contains the following:

system.edn

This is the top level EDIF of the processor and its peripherals. Only created if the **-flat** option is given.

*peripheral*_wrapper.edn

EDIF file of the peripheral. Only created if the **-flat** option is not given.

microblaze_*n*.edf

EDIF file of the MicroBlaze core.

Simulation Directory

The simulation directory contains the following:

system.vhd

This is the top level simulation file of the processor and its peripherals. Only created if the **-flat** option is given.

*peripheral*_wrapper.vhd

HDL simulation file of the peripheral made by **NGD2VHDL** or **NGD2VER**. Only created if the **-flat** option is not given.

`microblaze_n.vhd`

HDL simulation file of the MicroBlaze core.

`system.do`

Compile script for ModelSim.

Synthesis Directory

The synthesis directory contains the following:

`system.{prj|scr}`

Synthesis project file.

About Memory Generation

Platform Generator generates the necessary banks of memory and the initialization files for the Local Memory (LM) and OPB BRAM.

For the LM (lmb_lmb_bram) and OPB BRAM (opb_bram), the MHS options, C_BASEADDR and C_HIGHADDR (see the Microprocessor Hardware Specification Format documentation for more information), define the different depth sizes of memory.

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

Table 1: **Predefined Memory Sizes**

Architecture	Memory Size (kBytes)
Spartan-II	2, 4
Spartan-IIE	2, 4, 8
Virtex	2, 4, 8, 16
VirtexE	2, 4, 8, 16
Virtex2	8, 16, 32, 64

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

Platform Generator creates four blocks of memory. Each bank of memory is byte addressable (8 bits wide). Depending on the pre-defined memory size, each bank will contain one or more Select BlockRAMs.

For example, a memory size of 4 kBytes on a Virtex device, Platform Generator creates four banks of memory. Each bank is 8 bits wide and 1 kBytes deep. This configuration uses eight Select BlockRAMs, two Select BlockRAMs for each bank.

Use the **-mss** option to specify location of the MSS file. The MSS file contains location pointers to the programming information.

As of MDK 2.2, Platform Generator populates OPB BRAM and LM. One executable file can be distributed across multiple memories peripherals to cover the size of the program. For each

memory peripheral, Platform Generator will only request the required memory space. For example, if you define LM with a range of C_HIGHADDR=0x00001FFF and C_BASEADDR=0x00000000, Platform Generator will only request 8 kBytes of memory space from the executable file and populate the LM from hex 0 to hex 1FFF. If you define OPB BRAM with a range of C_HIGHADDR=0xFFFF2FFF and C_BASEADDR=0xFFFF2000, Platform Generator will request 4 kBytes from the executable file and populate OPB BRAM from the defined address range.

Reserved MHS Attributes

Platform Generator does automatic expansion on certain reserved attributes. These attributes will be populated by Platform Generator if encountered. This can prevent user error if your peripheral requires certain information on the platform being constructed. The following table lists the reserved attribute names:

Table 2: **Automatically Expanded Reserved Attributes**

Attribute	Description
C_FAMILY	FPGA Device Family
C_NUM_MASTERS	Number of masters
C_NUM_SLAVES	Number of slaves
C_NUM_INTR_INPUTS	Number of interrupt signals
C_OPB_AWIDTH	OPB Address width
C_OPB_DWIDTH	OPB Data width

Current Limitations

The current limitations of the Platform Generator flow are:

1. Only one MicroBlaze can be defined in the MHS file
2. Only one OPB can be defined in the MHS file
3. The OPB bus can be either a) a single D-OPB bus (Configurations 3 and 6 in which no I-OPB bus is present), or b) a single, unified I&D OPB bus (Configurations 1, 2, 4, and 5). Option b) requires users to declare an OPB arbiter peripheral in the MHS file. No error checking will be done to detect a missing OPB arbiter.
4. No error checking will be done to detect the address space requirements of the executable image.

Software Application Development Tools

Jan. 29, 2002

MicroBlaze GNU Compiler Tools

Summary

This document describes the various options supported by MicroBlaze GNU tools, such as **mb-gcc** compiler, **mb-as** assembler and **mb-ld** loader/linker. In this document, we discuss only those options, which have been added or enhanced for MicroBlaze. Standard libraries, provided as a part of the MicroBlaze GNU tools are also described briefly in this document.

Quick Reference

Table 1 briefly describes the commonly used compiler options.

Table 1: **Some commonly used compiler options**

Options	Explanation
-E	Preprocess only; Do not compile, assemble and link (Generates **.i** file)
-S	Compile only; Do not assemble and link (Generates **.s** file)
-c	Compile and Assemble only; Do not link (Generates **.o** file)
-g	Add debugging information for gdb to the final executable
-xl-mode-xmdstub	Intrusive hardware debugging on the board.Should be used only with xmdstub downloaded on to MicroBlaze
-xl-mode-bootstrap	Generate code, which can be downloaded using the boot strap loader
-xl-mode-bootstrap-reset	Same as bootstrap mode, but in this case, on reset, the control is transferred to user program instead of the boot stub.
-xl-mode-executable	Default mode for compilation.
-mxl-gp-opt	Use small data area anchors. Optimization for performance and size.
-mxl-soft-mul	Use software multiplier. Use this option when hardware multiplier is not present in the device. By default this option in turned ON.
-mno-xl-soft-mul	Do not use software multiplier. Compiler generates “mul” instructions.
-Wa,<*options*>	Pass comma-separated <*options*> on to the assembler
-Wp,<*options*>	Pass comma-separated <*options*> on to the preprocessor
-Wl,<*options*>	Pass comma-separated <*options*> on to the linker
-B <*directory*>	Add <*directory*> to the C-run time library search paths
-L <*directory*>	Add <*directory*> to library search path
-I <*directory*>	Add <*directory*> to header search path
-v	Display the programs invoked by the compiler
-o <*file*>	Place the output in <*file*>

Table 1: **Some commonly used compiler options**

Options	Explanation
-save-temps	Store intermediate files
--help	Display a short listing of options.
-O <*n*>	Specify Optimization level *n = 0,1,2,3*

Tool Usage

MicroBlaze GNU C compiler usage:

```
mb-gcc [options] file...
```

Compiler Options

The mb-gcc compiler for Xilinx's MicroBlaze soft processor introduces some new options as well as modifications to certain options supported by the gnu compiler tools. The new and modified options are summarized in this document.

-g

The -g option allows you to perform debugging at the source level. mb-gcc adds appropriate information to the executable file, which helps in debugging the code. mb-gdb provides debugging at source, assembly and mixed (both source and assembly) together.

-mxl-soft-mul

In some devices, a hardware multiplier is not present. In such cases, the user has the option to either build the multiplier in hardware or use the software multiplier library routine provided. MicroBlaze compiler mb-gcc assumes that the target device does not have a hardware multiplier and hence every multiply operation is replaced by a call to **mulsi3_proc** defined in library **libc.a**. Appropriate arguments are set before calling this routine.

-mno-xl-soft-mul

Certain devices such as VirtexII have a hardware multiplier integrated on the device. Hence the compiler can safely generate **mul** or **muli** instruction. Using a hardware multiplier gives better performance, but can be done only on devices with hardware multiplier such as Virtex 2.

-save-temps

This option saves the temporary files created while compiling a C code. The temporaries generated during the compilation phase are:

- Preprocessor output <*filename.i*>
- Assembler output <*filename.s*>
- Linker/Loader output <*filename.o*>

filename.c is the input C file.

-xl-mode-xmdstub

The mb-gcc compiler links certain initialization files along with the program being compiled. If the program is being compiled to work along with xmd, **crt1.o** initialization file is used, which returns the control of the program to the xmdstub after the execution of the user code is done. In other cases, **crt0.o** is linked to the output program, which jumps to halt after the execution of the program. Hence the option -xl-mode-xmdstub helps the compiler in deciding which initialization file is to be linked with the current program.

The code start address is set to **0x400** for programs compiled for a system with xmd. This ensures that the compiled program starts after the xmdstub. If the user intends to modify the default xmdstub, leading to increase in the size of the xmdstub, users should take care to change the start address of the text section. This option is described in the *Linker Loader Options* subsection.

Notes: -xl-mode-xmdstub is allowed only in hardware debugging mode and with xmdstub loaded in the memory. For software debugging (even with xmdstub), do not use this option. For more details on debugging with xmd, please refer to the XMD documentation.

-xl-mode-bootstrap

Certain programs are downloaded using the boot loader onto the device. This option links in crt2.o as the initialization file and starts the program at address location 0x100, leaving the first 100 words for the boot loader program. On a reset, the control is transferred back to the boot stub, which waits for loading a new program in the memory.

-xl-mode-bootstrap-reset

Same as the bootstrap mode above, but the reset location is overwritten to jump to the user code instead of the boot stub. Using this mode, the user does not have to reload the program on a reset, which is necessary in the previous mode.

-xl-mode-executable

This is the default mode used for compiling programs with mb-gcc. The final executable created starts from address location 0x0 and links in crt0.o. This option need not be provided on the command line for mb-gcc.

Notes: mb-gcc will signal fatal error, if more than one mode of execution is supplied on the command line.

-mxl-gp-opt

If the memory location requires more than 32K, the load/store operation would require two instructions. MicroBlaze ABI offers two global small data areas, which can contain up to 64K bytes of data each. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value. Hence needing only one instruction for load/store to the small data area.This optimization can be turned ON with the -mxl-gp-opt command line parameter. Variables of size lesser than a certain threshold value are stored in these areas. The value of the pointers is determined during linking. The threshold value can be changed using the **-Gn** option discussed below.

-Gn

The compiler stores certain data in the small data area of the code. Any global variable, which is equal to or lesser than 8 bytes will be stored in the small data area of the read-write or read-only section. This threshold value of 8 bytes could be changed using the above option in the command line.

Assembler Options

Assembler options can be used to aid in assembly level debugging.

-gstabs

The -gstabs option allows you to perform debugging at the assembly level. This should never be used with the -g compiler debug option. This option stores debugging information in a different format as compared to the information stored, while debugging with the -g option. Typically this option should be used along with -save-temps, since the debugger would need to refer to the assembler output file for debugging.

While using the mb-gcc flow, use -Wa, <option> to pass comma separated options to the assembler.

Linker/Loader Options

-defsym _STACK_SIZE=<value>

The total memory allocated for the stack and the heap can be modified by using the above linker option. The variable STACK_SIZE is the total space allocated for heap as well as the stack. The variable STACK_SIZE is given the default value of 100 words (i.e 400 bytes). If any user program is expected to need more than 400 bytes for stack and heap together, it is

recommended that the user should increase the value of STACK_SIZE using the above option. This option expects value in **bytes**.

In certain cases, a program might need a bigger stack. If the stack size required by the program is greater than the stack size available, the program will try to write in other forbidden section of the code, leading to wrong execution of the code.

Notes: Minimum stack size of 16 bytes (0x0010) is required for programs linked with the C runtime routines (crt0.o and crt1.o).

-defsym _TEXT_START_ADDR=<value>

By default, the text section of the output code starts with the base address 0x0. This can be overridden by using the above options. If the option -defsym _TEXT_START_ADDR=<value> is supplied to mb-gcc, the text section of the output code will now start from the given <value>. When the compiler is invoked with **-xl-mode-xmdstub**, the user program starts at 0x400 by default. The user does not have to use **-defsym _TEXT_START_ADDR**, if they wish to use the default start address set by the compiler.

-N

When the text start address of a particular program is modified using the option described above, an additional option -N has to be provided to the linker. For more details on this option, please refer to the GNU documentation.

-Wl, <option>

While using the mb-gcc flow, use -Wl, <option> to pass comma separated options to the assembler. For options with spaces, each part of the option needs to be prefixed by -Wl

The following is an example.

If you were using the “defsym_TEXT_START_ADDR=<value>” option in the mb-gcc command line, the option has to be given as:

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

For more information, type `mb-gcc --help` or consult the GCC manual (available online at http://www.gnu.org/manual)

Standard Libraries

MicroBlaze compiler tools provide a range of libraries for better performance and code size. These libraries are described in the MicroBlaze Libraries documentation. Certain libraries are optimized to give a better performance on the MicroBlaze processor. These libraries are *strcmp, strcpy, malloc, memset, memcpy* and *exit*. More details on the libraries provided with MicroBlaze are available in the libraries documentation.

In addition to the libraries, certain operations like divide are implemented in software. Multipliers might not be available on all the devices. Keeping this in mind, an option is also provided to use a software based multiplier instead of a hardware multiplier. To use software multiplier, use compiler options described in this document.

Division and Mod operations in MicroBlaze

Our devices do not support a divide operation in hardware. A divider in hardware would be extremely expensive and hence is not a good solution. Hence in addition to standard libraries, the compiler also generates a procedure for divide. Every divide operation is replaced by a call to this **divsi3_proc.** The divisor and the dividend are passed as parameters to the **divsi3_proc** and the **quotient** is returned in the **integer return register r3**.

Modulo operation is also carried out in similar fashion, except that the remainder is return in **integer return register r3**.

Software multiply

On devices with no hardware multiplier, such as Virtex and Spartan, multiply operation is done using a software multiply routine. The compiler will not generate a **mul** instruction, instead it will generate code to call a software routine **mulsi3_proc** to do the multiplication. This multiply routine is used to multiply two 32 bit integers and get a 32 bit output.

For long long operations, i.e for inputs of size 64 bits, another software routine is provided, which takes in two 64 bit numbers and returns a 64 bit result.

The compiler assumes no hardware muliplier on the device and always generates a subroutine call to the multiply routines for multiply operation. If the system is targetted for devices with hardware multiplier such as Virtex II, invoke the compiler with **-mno-xl-soft-mul** option on the command line to mb-gcc.

Psuedo-Ops

MicroBlaze supports a certain pseudo-ops making assembly programming easier for assembly writers. The supported pseudo-ops are listed in Table 2.

Table 2: **Psuedo-Opcodes supported by Assembler**

Psuedo Opcodes	Explanation
nop	No operation. Replaced by instruction: **or** R0, R0, R0
la Rd, Ra, Imm	Replaced by instruction: **addi** 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
lmi Rx, Ra, Imm [1]	Replace by (31-x+1) number of instructions: **lwi** Rx, Ra, Imm **lwi** R(x+1), Ra, Imm + 4 **lwi** R(x+2), Ra, Imm + 8 ... **lwi** R31, Ra, Imm + (31-x) * 4
smi Rd, Ra, Imm [1]	Replace by (31-x+1) number of instructions: **swi** Rx, Ra, Imm **swi** R(x+1), Ra, Imm + 4 **swi** R(x+2), Ra, Imm + 8 ... **swi** R31, Ra, Imm + (31-x) * 4

1.opcode not supported in the current version of the assembler

Operating Instructions

The gnu software tools for MicroBlaze can either be used to compile, assemble and link the input C file in one step by using the **mb-gcc** command or perform each step separately.

Entire Gnu Tool Flow

The mb-gcc can be used either to generate the final executable file in the elf-format. Alternatively, one of the two options below could be used to stop the compilation at a stage prior to the production of the final elf-formatted executable.

-S: Compile the input file only, without assembling or linking. In this case, the compiler generates <*filename.s*>,where input file is <*filename.c*>.

-c: Compile and assemble the input file, without linking. The compiler generates <*filename.o*>, where input file is <*filename.c*>

Environment Variable

mb-gcc refers to one environment variable for finding the appropriate initialization files, libraries and include files.

MICROBLAZE

This environment variable points to the base directory, where MicroBlaze system is installed and is set during the installation of MicroBlaze software tools.

This variable is represented as **$MICROBLAZE** on Unix shells and as **%MICROBLAZE%** on Windows command prompt.

Search Paths

On UNIX shells

MicroBlaze compiler (mb-gcc) searches certain paths for libraries and header files.

Libraries are searched in the following order:

1. Directories passed to the compiler with the **-L <dir name>** option.
2. Directories passed to the compiler with the **-B <dir name>** option.
3. **${MICROBLAZE}/lib**

Header files are searched in the following order:

1. Directories passed to the compiler with the **-I <dir name>** option.
2. **${MICROBLAZE}/include**

Initialization files are searched in the following order:

1. Directories passed to the compiler with the **-B <dir name>** option.
2. **${MICROBLAZE}/lib**

On Windows command prompt

MicroBlaze compiler (mb-gcc) searches certain paths for libraries and header files.

Libraries are searched in the following order:

1. Directories passed to the compiler with the **-L <dir name>** option.
2. Directories passed to the compiler with the **-B <dir name>** option.
3. **%MICROBLAZE%\lib**

Header files are searched in the following order:

1. Directories passed to the compiler with the **-I <dir name>** option.
2. **%MICROBLAZE%\include**

Initialization files are searched in the following order:

1. Directories passed to the compiler with the **-B <dir name>** option.

```
2. %MICROBLAZE%\lib
```

Initialization Files

The final executable needs certain registers such as the small data area anchors (R2 and R13) and the stack pointer (R1) to be initialized. These initialization files are distributed with the MicroBlaze Development Kit. In addition to the precompiled object files, source files are also distributed in order to help user make their own changes as per their requirements. Initialization can be done using one of the four C runtime routines:

crt0.o

This initialization file is to be used for programs which are to be executed standalone, i.e without **xmd**.

crt1.o

This file is located in the same directory and should be used when the **xmd** debugger is to be present in the system.

crt2.o

In case of programs used with the boot-loader, crt2.o is used as the initialization file. The boot loader is used to load the program at runtime using the boot stub.

crt3.o

The source for crt2.o and crt3.o is the same as the functionality is the same except for the behavior on a reset. In crt3.o, address location 0x0 is overwritten, such that on a reset, the control is transferred to the user program instead of the boot stub.

These files are described in detail in the MicroBlaze ABI documentation. The source for initialization file can be changed as per the requirements of the project. These changed files have to be then assembled to generate an object file (.o format). To refer to the newly created object files instead of the standard files, use the **`-B <directory-name>`** command line option while invoking **`mb-gcc.`**

Command Line Arguments

MicroBlaze currently does not support an operating system. Hence command line arguments cannot be used in programs compiled with MicroBlaze compiler. The command line arguments argc and argv are initialized to 0 by the C runtime routines.

Interrupt Handlers

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

In order to distinguish an interrupt handler from a sub-routine, mb-gcc looks for an attribute (interrupt_handler) in the declaration of the code. This attribute is defined as follows:

```
void int_handler_func () __attribute__ ((interrupt_handler));
```

Attribute for interrupt handler is to be given only in the prototype and not the definition.

Interrupt handlers might also call other functions, which might use volatile registers which were not saved by the interrupt handler routine. These functions are defined with **`save_volatiles`** attribute as

```
void int_call_func () __attribute__((save_volatiles));
```

For correct code, all the sub-routines called from the interrupt handler routine should be declared with **`save_volatiles`** attribute.

The attributes with their functions are tabulated in Table 3.

Interrupt handlers can also be defined in the MicroBlaze Hardware Specification (MHS) and the MicroBlaze Software Specification (MSS) file. These definitions would automatically add the

attributes to the interrupt handler functions. For more information please refer MicroBlaze Interrupt Management document.

Table 3: **Use of attributes**

Attributes	Functions
`interrupt_handler`	This attribute saves the machine status register and all the volatiles being used in the function in addition to the non-volatile registers. **`rtid`** is used for returning from the interrupt handler
`save_volatiles`	This attribute is used for sub-routines called from interrupt handlers. This attribute saves the volatile registers being used by the current sub-routine in addition to the other non-volatile registers.

Microprocessor Software IDE (XSI)

Mar. 11, 2002

Summary

This document describes the Microprocessor Software IDE (XSI) utility used for customizing software flow of the 32-bit soft processor, MicroBlaze.

Overview

Xilinx Software IDE (XSI) provides an integrated GUI for creating the software specification file for the Microprocessor system. It also provides an editor and a project management interface to create and edit source code. The IDE offers software tool flow customization options.

Processes Supported

XSI supports the creation of the MSS file (refer Microprocessor Software Specification documentation) and software tool flows associated with this software specification. This version of XSI does not support complete HW/SW tool flows. It only supports customization of SW libraries, drivers, interrupt handlers and compilation of user programs. It is assumed that a hardware specification exists in the form of an MHS file.

Tools Supported

Table 1 describes the tools that are supported in the IDE.

Table 1: **Tools supported in XSI**

Tool	Function	Reference
Library Generator (libgen)	Customizes software libraries, drivers and interrupt handlers	The Library Generator Documentation
GNU Compiler Tools	Preprocess, compile, assemble and link programs	GNU tools Documentation

Features

XSI has the following features

1. Creation of Software Specification (MSS) for a given hardware specification (MHS)
2. Support for all the tools described in Table 1.
3. Viewing and editing of C source and header files
4. Project Management

Project Management

A **project** consists of the Microprocessor Software Specification (MSS) and the C source and header files that need to be compiled into an executable. The MSS file also includes a reference to the MHS file. The project also includes the FPGA architecture family for which the system is created.

Creating New Project

A New Project is created using the **New Project** menu option in the Project submenu of the main menu. The **New Project** toolbar button can also be used. A new project requires an MHS file (refer the Microprocessor Hardware Specification documentation) and a project directory where flow tools create output files and directories. Source and Header files required for user

application development are created and added as described in the Source Code Management section. Project options are written into an **xmp** (Xilinx Microprocessor Project) file.

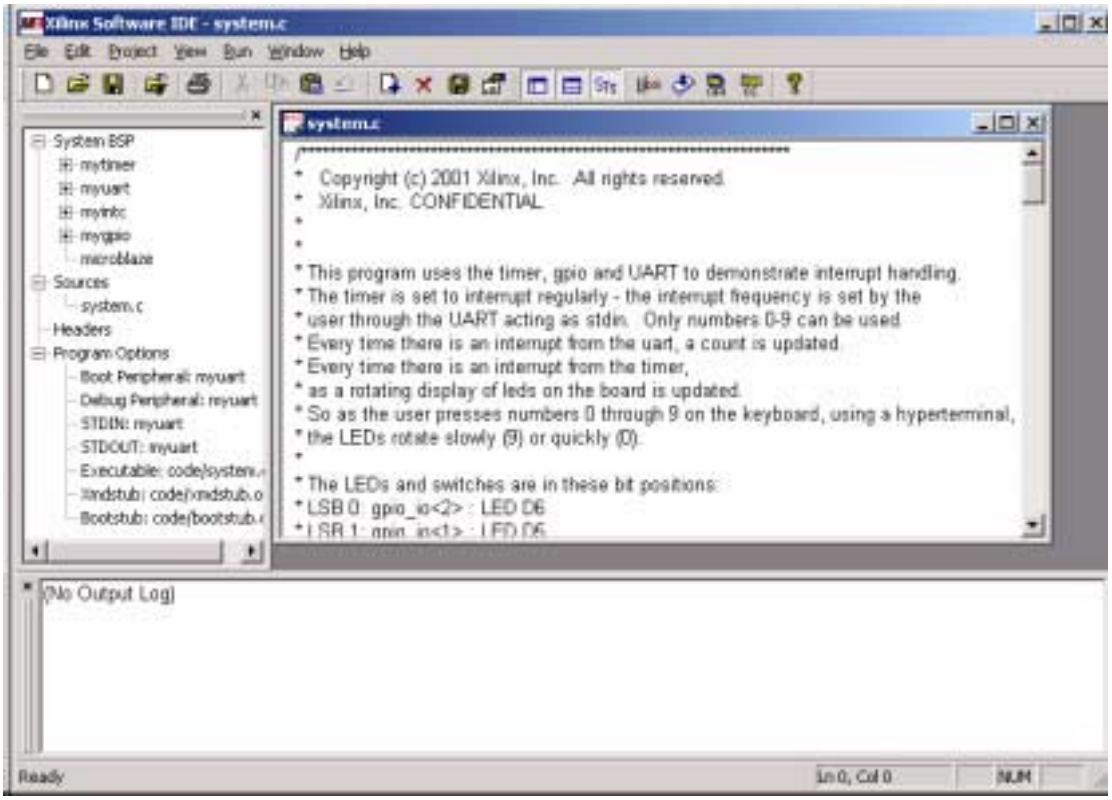
Opening Existing Project

An existing **xmp** file should be opened and worked on using the **Open Project** menu option (**Project** submenu of Main menu). New source files and header files can be created and added as described in the Source Code Management section of this documentation.

A new project can be created or opened only if the current project is closed. XSI does not allow multiple projects to be open simultaneously.

XSI Interface

The figure below shows a screenshot of XSI. The main window is to the right, the project view window is to the left, and the transcript window is at the bottom.



XSI Main Window

All source and header file editing is performed in the main window of XSI. Any number of source and header files can be open simultaneously.

Project View Window

The project view window shows system components (processor and peripherals), source and header files of the project and software specification options.

Transcript Window (Console)

The transcript window is the bottom window when XSI is started. This window acts as a console for output, warning and error messages from tools invoked.

Software Platform Management

In the Project View Window, the system BSP and the program options are displayed as a tree structure.

System BSP Tree

The System BSP tree displays all the peripherals in the system that can be customized for the software flow.

Double clicking on each peripheral opens a dialog window displaying user settable software options.

Interrupt Handler Routines - The name of the interrupt handling routine is specified for the peripheral interrupt signal.

Driver and Driver Version Option - This option sets the driver name and version number used for the peripheral instance.

Program Options

Table 2 shows the options that are displayed in the Program Options Category.

Table 2: **Program Options for Software Tools**

Option	Value	Description
Boot Peripheral	Instance Name	Designates the peripheral instance as Boot peripheral
Debug Peripheral	Instance Name	Designates the peripheral instance as debug peripheral. Here the peripheral will be used to download the debug stub (xmdstub)
STDIN	Instance Name	Peripheral designated as standard input
STDOUT	Instance Name	Peripheral designated as standard output
Bootstrap	Dir/Filename	Specifies the bootstub file to be created
Xmdstub	Dir/Filename	Specifies the debug stub (xmdstub) to be created
Executable	Dir/Filename	Name of the executable.

For more information on the options, please refer Library Generator Documentation and Microprocessor Software Specification documentation.

Source Code Management

XSI has an integrated editor for viewing and editing C source and header files of the user program.

Adding Files to Project

Files can be added to the project by clicking the right mouse button on the **Sources** or **Headers** Tree Item in the Project View Window. The same operation can be accomplished by using the Project submenu in the Main menu. Multiple files are added by pressing the control key and using arrow keys (or the mouse) to select in the file selection dialog.

Deleting Files from Project

Any file can be deleted from the project by selecting the file in the Project View window and pressing the DEL key, or by clicking the right mouse button on the item and choosing **Delete File from Project**. Note that the file does not get physically deleted from the system.

Editing Files

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

Flow Tool Settings

XSI supports tool flows as shown in Table 1. The Main menu has a **Run** submenu. In this submenu, the **Set Options** pull down menu can be used to set various Tool options.

Set Compiler Options

This menu item allows the user to set various compiler options as shown in Table 3. Each option is dealt in detail in the GNU Compiler Tools documentation. Only the Microblaze GNU compiler specific options can be set in this version of XSI.

Table 3: **Compiler Options that can be set using XSI**

Option Tab	Option	Description
General Options	Flow Option	Runs the compiler flow till preprocessor, compile, assemble or link stage.
Compiler Options	Optimization Level	Choose the level of compiler optimization. Equivalent to -O option in gcc.
	Global Pointer Optimization	This option enables global pointer optimization in the compiler
	Hardware Multiply	Enables the use of hardware multiplier on Virtex II.
	Debug Options	-g option to generate debug symbols or -gstabs option to generate stabs information.
Directories	Search Paths	Compiler, Library and Include paths. Equivalent to -B, -L and -I option to gcc.
	Output File	Sets the name of the executable file. Equivalent to -o option of gcc.
Other	Program Start Address	Specifies the start address (in hex) of the text segment of the executable. This is the address at which MicroBlaze starts execution.
	Stack Size	Specifies the stack size in bytes for the program. This is specified as a hex value.
	Pass Options	Options can also be passed to the compiler, assembler and linker. The options have to be comma separated. For eg. to pass

Set Libgen Options

A libgen options dialog is presented to the user when this menu item is selected. The following options can be set.

Peripheral Repository Directory - Specifies the directory that contains user peripherals and associated files for the peripherals such as MPD files, PAO files etc. This repository should also contain driver directories for the user peripheral. See libgen documentation for more information.

Tool Invocation

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

The current version of XSI does not check for flow dependencies. Hence, the user is responsible for invoking the tools in the proper order.

When libgen is invoked, an MSS file is created for the software specification. When the user exits the application, a prompt to save the current project appears. The user can also save the project in another name by using **Save Project As** in the Project submenu of the Main menu.

Flow Engine (XMF)

Apr. 04, 2002

Summary

This document describes the flow engine utility **XMF** used for MicroBlaze tool flow. The flow engine is an extension to the **Xflow** utility available as part of Xilinx ISE tools. Please refer to the Xilinx tools documentation for more information.

Overview

The flow engine can be used to schedule tool flows (library generator, compiler tools, platform generator) using flow files and option files. XMF is an enhancement to the **Xflow** utility that is available as part of Xilinx Design Implementation (ISE) tools.

Tool Requirements

The flow engine for the embedded flow uses a flow file **processor.flw** that is provided in the MDK distribution. The flow file specifies the various tools that are executed in a particular order, and the control options for the programs. This flow file supports execution of the library generator, compiler tools and the platform generator. For backend synthesis tool flow, the **xflow** utility can be used. Three option files (**xmdstub.opt**, **bootstrap.opt** and **executable.opt**) are also provided with some common options to the individual tools in the flow. These option files can be editted to change or add new options to the tools in the tool flow.

Tool Usage

The tool is run as follows:

```
xmf -p <architecture> -processor <option file> -g mode:<mode> <design>
```

-p Option

Specifies the FPGA architecture. Valid options are virtex, virtexe, spartan2, spartan2e and virtex2.

-processor Option

Specifies one of *xmdstub.opt*, *bootstrap.opt* or *executable.opt* option files.

Other options that can be used are detailed in the **Xflow** documentation. Some of the options that may be useful apart from **-processor** are **-implement, -config, -fit, -assemble** and so on for backend synthesis and configuration tool flow. Please refer the Xflow documentation for more information.

-g mode:<mode> Option

Specifies the mode corresponding to the option files. Valid options are *xmdstub*, *bootstrap* and *executable*.

<design> specifies the MSS file that is input to the tool flow.

Debug Tool Chain

MicroBlaze Debug and Simulation

Jan. 9, 2002

Summary

This document describes debug and simulation options for the 32-bit soft processor, MicroBlaze.

Overview

Debug and simulation are integral parts of developing programs for embedded systems. MicroBlaze tools offer various options to support both software and hardware debug and simulation. This document explains the options available based on the functionality needed.

Terms and Definitions

Software Debug

Software debug is the process of debugging an application program only. It is assumed that the available hardware is functionally correct. An example of a software debugger is GNU's GDB.

Hardware Debug

Hardware debug is the process of debugging hardware. Tools available for debugging hardware design targeted at FPGAs include hardware simulators, such as *ModelSim* by Model Technology Incorporated or an in-circuit logic analyzer such as Xilinx's *ChipScope ILA*.

Hardware Simulation

Hardware simulation involves exercising a design under test in an execution engine. The engine simulates the design's behavior. Designs are usually specified by means of an HDL model of the hardware.

Co-Simulation

Co-simulation is the process of debugging both hardware and software concurrently through simulation.

Intrusive Software Debug

Intrusive software debug involves the introduction of a software control and monitoring agent (a debug stub). The agent provides functions for setting breakpoints, accessing registers and memory, etc. The agent is compiled along with the application program under test. It should be noted that the introduction of the agent may cause side effects which can affect the behavior of the test program.

Non-Intrusive Software Debug

Software Debug is non-intrusive if *no* software monitoring agent is compiled with the test program. The monitor does not affect the order of execution of program events.

Instruction Set Simulator (ISS)

An ISS is a software which uses a model of the instruction set of a given processor to execute the application program under test. The execution time step is that of a single program instruction. An ISS provides functionally correct program execution data and can be used in intrusive or in non-intrusive configurations.

Cycle Accurate ISS

A cycle accurate ISS uses detailed information of the processor's pipeline behavior to model the instruction execution and provide cycle count information of the executed instructions. The execution time step is that of a single clock cycle. Any given processor instruction may take one or more clock cycles to execute.

Hardware Board

A board contains an FPGA device such as a VirtexE-100, and several other components such as displays (LCD/LED), connectors (serial/parallel), memory, etc.

MicroBlaze System

A MicroBlaze system is a configuration of the FPGA on a particular board that contains a customized MicroBlaze processor core. It also contains additional logic, possibly in the form of IP cores that control one or more components on the board.

Software Debug

Overview

The MicroBlaze development tools (MDT) provide options for using an ISS, a Cycle-Accurate ISS or a hardware board for debugging software.

Simulators provide non-intrusive debugging, while hardware boards allow debugging in an intrusive manner. If the hardware board is used to debug, a debug stub called **xmdstub** is used to control the execution of the test program and provide communication between the debugging host and the board executing the test program.

For source level debugging, programs should be compiled with **`-g`** option. This will add debugging information for **`mb-gdb`**. While initially verifying the functional correctness of a C program, it is also advisable to not use any **`mb-gcc`** optimization option like -O2 or -O3 as **`mb-gcc`** does aggressive code motion optimizations which may make debugging difficult to follow.

Using a Simulator

The test program is compiled using the command:

```
mb-gcc -g program.c
```

This command creates a MicroBlaze executable **`a.out`** with the debugging information needed by mb-gdb.

Simple ISS

This method of debug can be used to determine if the program is functionally correct. An ISS does not model the behavior of any peripheral, hence the program must not access any peripherals or any memory beyond the local memory address space. See the Program Layout documentation on more details on address space restrictions.

There is an ISS integrated in mb-gdb. To use it, load the program **`a.out`** in **`mb-gdb`**. Select **Target Settings** from the **File** menu. In the mb-gdb **Target Selection** dialog, choose **simulator** and click **OK**. Now, the program can be downloaded to the ISS and executed in it. See the GNU Debugger documentation for more information on debugging using mb-gdb.

Cycle Accurate ISS

The MDT includes the Xilinx Microprocessor Debug (XMD) engine that integrates debug and simulation in both hardware and software. It provides a consistent user interface through mb-gdb. The cycle accurate ISS is a part of this engine. See the MicroBlaze XMD documentation for more information.

In order to debug using the cycle accurate ISS, the program should be compiled with **`mb-gcc`** as shown above. XMD is a separate program that must be started in simulator mode using the following command:

```
xmd -t sim
```

To use the cycle accurate ISS in XMD, load the program **a.out** in **mb-gdb**. Select **Target Settings** from the **File** menu. In the mb-gdb **Target Selection** dialog, choose:

- **Target**: Remote/TCP
- **Hostname**: localhost
- **Port**: 1234

Select **Connect to target** from the **Run** menu. This will attempt a connection from mb-gdb to the XMD engine. If successfully connected, the mb-gdb interface can be used to debug the program. For further information, refer to the MicroBlaze XMD documentation and MicroBlaze GNU Debugger documentation.

Please note that the XMD cycle accurate ISS does not simulate peripherals in this release.

Using Hardware

The application program can also be debugged using the hardware board. In this case, an **xmdstub** executable must be generated and compiled with the program. To do that, the MSS file must contain an XMDSTUB attribute that specifies the xmdstub file location. Library Generator is then run with the **-mode xmdstub** option.

The Library Generator creates an xmdstub executable in the location specified. Libgen also configures libraries for the system. Please see the Library Generation documentation for more information.

The application program is compiled using mb-gcc as follows:

```
mb-gcc -g -xl-mode-xmdstub program.c
```

This command creates the test program executable **a.out.**

Platgen is now run with **-mode xmdstub** option. The Local Memory (LM) is initialized with the **xmdstub** executable and a netlist for the system is created. Please see the Platform Generator documentation for more information.

Start the XMD engine in hardware mode in a new window with the following command:

```
xmd -t hw
```

To use the hardware board through XMD, load the program **a.out** in **mb-gdb**. Select **Target Settings** from the **File** menu. In the mb-gdb **Target Selection** dialog, choose:

- **Target**: Remote/TCP
- **Hostname**: localhost
- **Port**: 1234

Select **Connect to target** from the **Run** menu. This will attempt a connection from mb-gdb to the XMD engine. If successfully connected, the mb-gdb interface can be used to debug the program. For further information, refer to the MicroBlaze XMD documentation and MicroBlaze GNU Debugger documentation.

When using the hardware for debugging, any memory mapped peripherals can be accessed as regular memory. It is assumed that the hardware is functionally correct.

Hardware Simulation

Hardware Simulation Overview

Hardware simulation models for MicroBlaze are provided as VHDL files that can be used in any VHDL simulator. For peripherals that are shipped with the MicroBlaze Development Kit (MDK), the simulation models are generated by Platform Generator (platgen). All simulation files are created in the **simulation** subdirectory by platgen.

Platgen generates simulation models for the system when the **-sim** option is specified. Platgen generates a hierarchal netlist and simulation models by default. Platgen can also generate a flattened netlist and simulation models if the **-flat** option is specified. See the Platform Generator documentation for more information.

Output Files

Flatten Mode

In the flatten mode, two HDL simulation files that represent the entire system are generated. One is an HDL file that represents the system without the processor core (e.g. **`mysystem.vhd`**), and the other is the processor core (e.g. **`microblaze_3.vhd`**) simulation file.

Hierarchical Mode

In the hierarchal mode, multiple simulation HDL files are generated: one for the processor configuration and one each for the peripherals, bus and memory defined in the system.

Setup Script and Signals

Any HDL simulator can be used to compile and simulate the models generated. Platgen supports an option to generate a *ModelSim* specific script file for compilation of the simulation models.

Important Signals

The signals **sys_clk** and **sys_rst** are the system clock and reset signals. These signals can be controlled as stimuli during a simulation.

Some of the other important signals that the MicroBlaze simulation file includes are:

- **register_file** - All the register file registers in the processor core.
- **rpc_fetch** - Register with the program counter value at the fetch stage
- **rpc_decode** - Register with the program counter value at the decode stage
- **rpc_execute** - Register with the program counter value at the execute stage
- **rimm** - Register that holds the value of an imm instruction
- **fetch_stage** - Instruction opcode at the fetch stage of the pipeline.
- **decode_stage** - Instruction opcode at the decode stage of the pipeline.
- **execute_stage** - Instruction opcode at the execute stage of the pipeline.

Requirements

ModelSim Libraries

Unisims and *Simprims* libraries are required for hardware simulation. These can be obtained from the Xilinx support web site at **http://support.xilinx.com**

Co-Simulation and Debug

MicroBlaze System Debug

Software debug and hardware simulation can be used to find bugs in software and hardware respectively. Co-simulation support is necessary for debug of both hardware and software together without assuming correctness of the other. An HDL simulator (e.g. ModelSim) can be used for this purpose.

Using ModelSim

The simulation files that are generated by Platgen contain simulation files and configuration files for local memory. If the application program is included in the platgen run, then the memory configuration files contain the executable information. When the HDL simulator is used as described above, interaction between the program and hardware can be seen by tracing important signal waveforms, registers and memory contents. The local memory simulation model is integrated into the system model and this allows for a powerful co-simulation support.

Program Monitoring

The Xilinx Microprocessor Debug Terminal (XMD terminal) is a program that provides a simple text based interface for monitoring a MicroBlaze system. It communicates with a hardware board running the XMD stub through a JTAG download cable or serial cable. The XMD terminal program allows a user to load programs into the memory of a MicroBlaze system, execute them and, probe and modify the contents of memory and registers.

MicroBlaze GNU Debugger

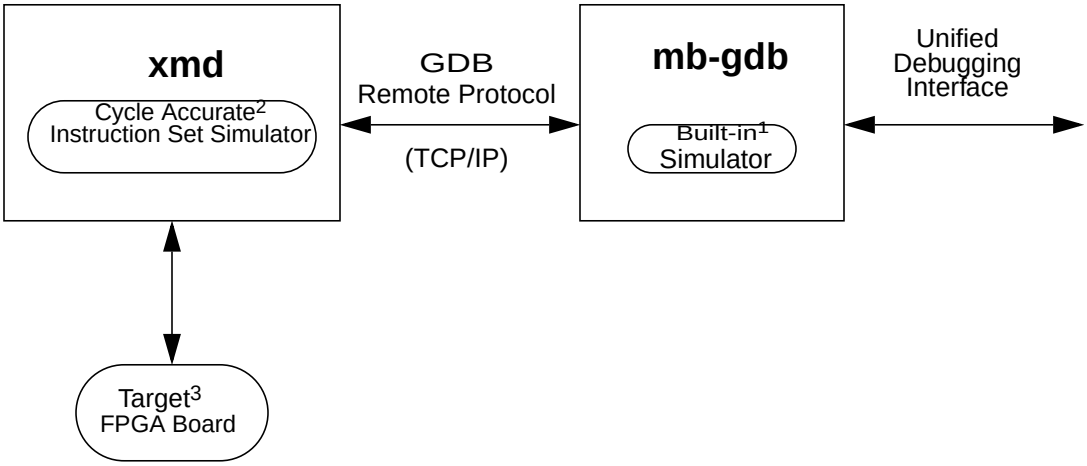
Jan. 11, 2001

Summary

This document describes the general usage of the Xilinx GNU debugger for MicroBlaze.

Overview

MicroBlaze GDB is a powerful yet flexible tool which provides a unified interface for debugging/verifying a MicroBlaze system during various development phases.



Tool Usage

MicroBlaze GDB usage:

```
mb-gdb [options] [executable-file [core-file or process-id]]
```

Tool Options

The most common options in the MicroBlaze GNU debugger are:

--command=FILE

Execute GDB commands from FILE. Used for debugging in batch/script mode.

--batch

Exit after processing options. Used for debugging in batch/script mode.

--nw

Do not use a GUI interface.

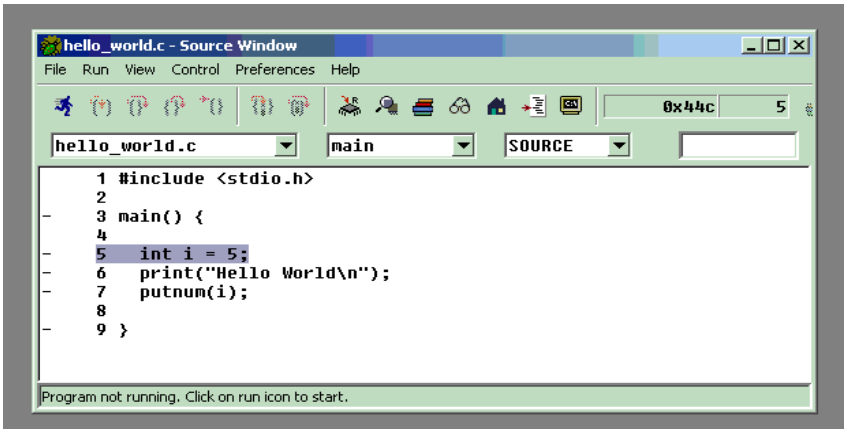
-w

Use a GUI interface. (Default)

MicroBlaze GDB Targets

Currently, there are three possible targets that are supported by the MicroBlaze GNU Debugger and XMD tools - a built-in simulator target and two remote targets (XMD):

```
xilinx > mb-gdb hello_world.out
```

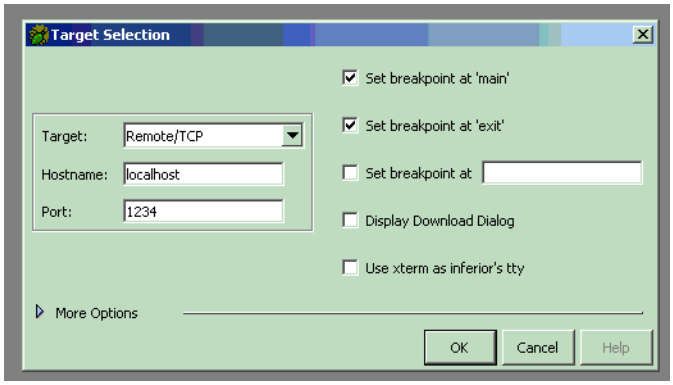


From the ***Run*** pull-down menu, select ***Connect to target*** in the mb-gdb window. In the Target Selection dialog, you can choose between the **`Simulator`** (built-in) and **`Remote/TCP`** (for XMD) targets.

In the target selection dialog, choose:

- Target: Remote/TCP
- Hostname: localhost
- Port: 1234

Click *OK* and mb-gdb will attempt to make a connection to XMD. If successful, a message will be printed in the shell window where XMD was started.



At this point, **`mb-gdb`** is connected to XMD and controls the debugging. The simple but powerful GUI can be used to debug the program, probe memory, registers, etc.

GDB Built-in Simulator

The MicroBlaze debugger provides an instruction set simulator, which can be used to debug programs that do not access any peripherals. This simulator makes certain assumption about the executable being debugged:

- The size of the application being debugged determines the maximum memory location

which can be accessed by the simulator.

- The simulator assumes that the accesses are made only to the fast local memory (LMB).

When using the command `info target`, the number of cycles reported by the simulator are under the assumptions that memory access are done only into local memory (LMB).Any access to the peripherals will result in the simulator indicating an error.This target does not require **xmd** to be started up. This target should be used for basic verification of functional correctness of programs which do not access any peripherals or OPB or external memory.

Remote

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

Simulator Target

The XMD simulator is a Cycle-Accurate Instruction Set Simulator of the MicroBlaze system which presents the simulated MicroBlaze system state to GDB..

Hardware Target

With the hardware target, XMD communicates with an xmdstub program running on a hardware board through the serial cable or JTAG cable, and presents the running MicroBlaze system state to GDB.

For more information about XMD refer to the XMD Documentation.

Note

1. The simulators provide a non-intrusive method of debugging a program. Debugging using the hardware target is intrusive because it needs an xmdstub to be running on the board.
2. If the program has any I/O functions like print() or putnum(), that write output onto the UART or JTAG Uart, it will be printed on the console/terminal where the xmd server was started. (Refer to the MicroBlaze Libraries documentation for libraries and I/O functions information).

GDB Command Reference

For help on using mb-gdb, click on **Help->Help Topics** in the GUI mode

or type **"help**" in the console mode.

In the GUI mode, to open a console window, click on **View->Console**

For a comprehensive online documentation on using GDB, refer to
http://www.gnu.org/manual/gdb/

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

Table 1 briefly describes the commonly used mb-gdb console commands. The equivalent GUI

Table 1: **Commonly Used GDB Console Commands**

Command	Description
b main	Set a breakpoint in function main
r	Run the program (for the built-in simulator only)
c	Continue after a breakpoint, or Run the program (for the xmd simulator only)
l	View a listing of the program at the current point
n	Steps one line (stepping over function calls)
s	Step one line (stepping into function calls)
info reg	View register values
info target	View the number of instructions and cycles executed (for the built-in simulator only)
monitor info	View the number of instructions and cycles executed (for the xmd simulator only)
p *xyz*	Print the value of *xyz* data

versions can be easily identified in the mb-gdb GUI window icons. Some of the commands like info target, monitor info, may be available only in the console mode.

Compiling for Debugging

In order to debug a program, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code. The mb-gcc compiler for Xilinx's MicroBlaze soft processor will include this information when the appropriate modifier is specified.

The **`-g`** option in **`mb-gcc`** allows you to perform debugging at the source level. mb-gcc adds appropriate information to the executable file, which helps in debugging the code. mb-gdb provides debugging at source, assembly and mixed (both source and assembly) together. While initially verifying the functional correctness of a C program, it is also advisable to not use any mb-gcc optimization option like -O2 or -O3 as mb-gcc does aggressive code motion optimizations which may make debugging difficult to follow. For debugging with **`xmd`** in hardware mode, the **`mb-gcc`** option **`-xl-mode-xmdstub`** must be specified. Refer to the XMD documentation for more information about compiling for specific targets.

MicroBlaze XMD

Mar. 22, 2002

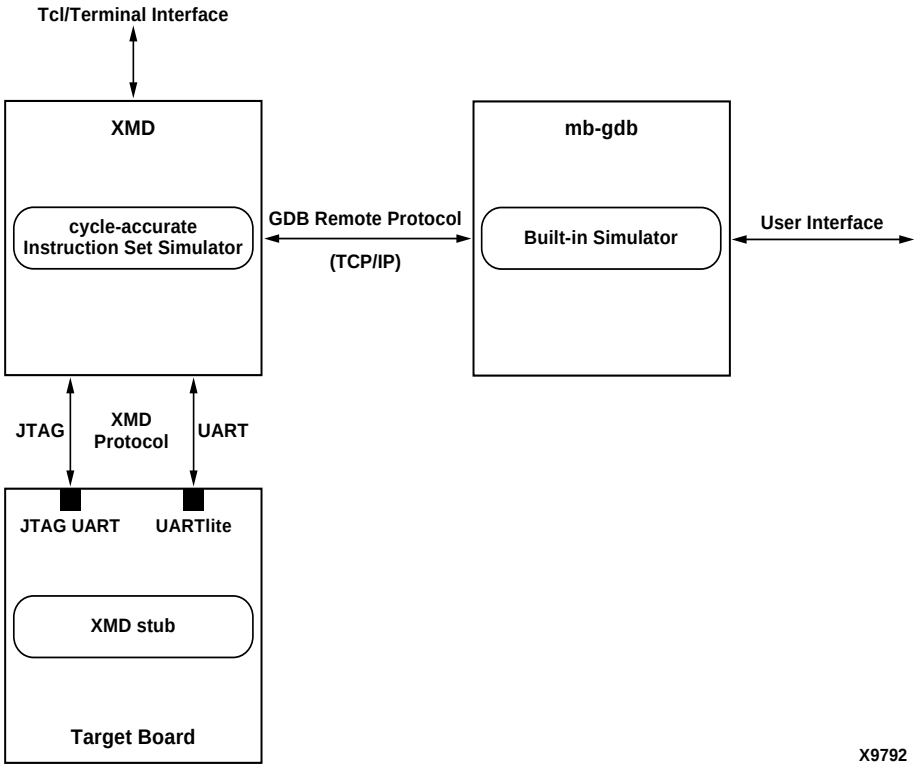
Summary

This document describes usage of the Xilinx Microprocessor Debug (XMD) tool.

Overview

The Xilinx Microprocessor Debug (XMD) Engine is a program that facilitates a unified GDB interface for debugging programs on a MicroBlaze system as well as a Tcl interface for system verification. It supports non-intrusive debugging of programs running on the cycle accurate MicroBlaze simulator or intrusive debugging on a remote hardware board. In the hardware mode, a small stub program executing on the target MicroBlaze board provides communication between **xmd** and the board.

The XMD Engine is used along with MicroBlaze GDB (**mb-gdb**) for debugging. **xmd** implements the GDB Remote Protocol and accepts connections from **mb-gdb** on a TCP port. **mb-gdb** can connect to **xmd** running on the same computer or any computer on the Internet. **mb-gdb** is the user interface for debugging programs either in simulation or in hardware. The Tcl interface can be used either for low-level debugging as well as running verification test scripts to test the complete system.



XMD Usage

To start the XMD engine, execute xmd from a shell as follows.

```
xmd [options]
```

XMD Options

-u [tcl | gdb]

Specify the user interface for the current session. With the Tcl interface, xmd starts a Tcl shell augmented with XMD commands. See the "XMD Commands" section for a list of commands. With the gdb interface, **xmd** starts the simulator or connects to the target board and then listens for TCP connections from **mb-gdb**. Default interface if the Tcl interface.

-t [sim | hw]

Specify the debug target. **xmd** supports non-intrusive debugging on the MicroBlaze simulator or intrusive debugging on remote hardware. Use **sim** for simulator or **hw** for remote hardware. The default target is the simulator.

-c [s | j]

Specify the **xmd** communication. Debugging is supported over JTAG (using opb_jtag_uart peripheral) or serial cable (using opb_uart peripheral). Use **s** for serial or **j** for JTAG. Default is JTAG communication.

-j <FPGA_device_position>

Specify the position of the FPGA device in the JTAG chain that contains the MicroBlaze system to be debugged. The JTAG chain positions are auto detected and displayed by **xmd** on startup.

-J <list of BSDL files>

Specify the configuration of the JTAG chain on the target board by providing the BSDL files for all the devices that make up the JTAG chain in the same order as they occur in the chain.

By default, xmd autodetects the JTAG chain. But if it fails to do so, then this option can be used to connect to the target board.

-d <time-out>

Specify the time-out delay for **xmd** communication in seconds.

-p <tcp_port>

Specify the TCP port to accept **mb-gdb** connections on. The default TCP port is *1234*.

-s <serial_port>

Specify the serial port where the remote hardware is connected. The default serial port is */dev/ttya* on Solaris and *Com1* on Windows.

-b <baud>

Specify the serial port baud rate in bps. The default value is *19200* bps.

-h

Display help.

-V

Run in Verbose mode. Prints detailed messages about **xmd** operations.

-v

Display the version number

Hardware target

With a hardware target, user programs can be downloaded from **mb-gdb** directly onto a remote hardware board and be executed with support of the xmd stub running on the board. A sample session of XMD with a hardware target is shown below.

```
xilinx > xmd -t hw -u gdb
MicroBlaze XMD Engine
Using Hardware board debugging through XMD stub
Connecting to XMD stub at baud rate: 19200 bps
XMD stub initialized. Version No: 2
Use the following command in GDB to connect:
   target remote <hostname>:1234
```

Now XMD is connected with the hardware target and is waiting for a connection from **mb-gdb**. Refer the MicroBlaze GNU Debugger document to see how to start **mb-gdb**, make a remote connection from **mb-gdb** to **xmd**, download a program onto the target and debug the program.

To debug a program by downloading on the remote hardware board, the program has to be compiled with **-g -xl-mode-xmdstub** options to mb-gcc.

User Program Outputs

If the program has any I/O functions like print() or putnum(), that write output onto the UART or JTAG Uart, it will be printed on the console/terminal where the **xmd** was started. (Refer to the MicroBlaze Libraries documentation for libraries and I/O functions information).

Hardware Target Requirements

To debug programs on the hardware board using XMD, the following requirements have to be met.

1. **xmd** uses a JTAG or serial connection to communicate with **xmdstub** on the board. Hence a JTAG Uart or a Uart designated as DEBUG_PERIPHERAL in the mss file is needed on the target MicroBlaze system.

Platform Generator can create a system that includes a JTAG Uart or a Uart, if specified in the system's mhs file. For more information on creating a system with a Uart or a JTAG Uart, refer to the MicroBlaze Hardware Specification Format documentation.

2. **xmdstub** on the board uses the JTAG Uart or Uart to communicate with the host computer. Hence, it has to be configured to use the JTAG Uart or Uart in the MicroBlaze system.

Library Generator can configure the **xmdstub** to use the DEBUG_PERIPHERAL in the system. When **libgen** is run with **-mode xmdstub** option, it will generate a **xmdstub** configured for the DEBUG_PERIPHERAL and put it in **code/xmdstub.out** as specified by the XMDSTUB attribute in the mss file. For more information, refer to the Library Generator documentation.

3. **xmdstub** executable must be included in the MicroBlaze local memory at system startup. To have the **xmdstub** included in the MicroBlaze local memory, the `xmdstub.out` file should be specified in the user's mss file as follows:

```
SET attribute XMDSTUB=code/xmdstub.out
```

Platform Generator can populate the MicroBlaze LMB when it is run with the **-mode xmdstub** option. It will use the executable specified in the XMDSTUB attribute to initialize the MicroBlaze local memory while generating the system netlist.

4. Any user program that has to be downloaded on the board for debugging should have a program start address higher than 0x400 and the program should be linked with the startup code in crt1.o

mb-gcc can compile programs satisfying the above two conditions when it is run with the option **-xl-mode-xmdstub**. For source level debugging, programs should also be compiled with **-g** option. While initially verifying the functional correctness of a C program, it is advisable to not use any mb-gcc optimization option like -O2 or -O3 as mb-gcc does aggressive code motion optimizations which may make debugging difficult to follow.

Simulator target

You can use **mb-gdb** and **xmd** to debug programs on the cycle-accurate simulator built in XMD. A sample session of XMD and GDB is shown below.

To startup the XMD engine with the Simulator target, you need to specify the target sim.

```
xilinx > xmd -t sim
MicroBlaze XMD Engine
Using Simulator
Use the following command in GDB to connect:
   target remote <hostname>:1234
```

Now XMD is running with the simulator target and waiting for a connection from mb-gdb.

Refer the MicroBlaze GNU Debugger document to see how to start **mb-gdb**, make a remote connection from **mb-gdb** to **xmd**, download a program onto the target and debug the program. With **xmd** and **mb-gdb**, the debugging user interface is uniform with simulation or hardware targets.

Simulation Statistics

While **mb-gdb** is connected to XMD with the simulator target, the statistics of the cycle-accurate simulator can be viewed from mb-gdb as follows:

- In the **mb-gdb** GUI menu, select **View->Console**.
- In the console window, type **monitor info**
- To reset the simulation statistics, type **monitor reset**

Simulator Target Requirements

To debug programs on the Cycle-Accurate Instruction Set Simulator using XMD, the following requirements have to be met.

1. Programs should be compiled for debugging and should be linked with the startup code in crt0.o

mb-gcc can compile programs with debugging information when it is run with the option **-g** and by default, mb-gcc links crt0.o with all programs. (Explicit option: **-xl-mode-executable**)

2. Programs can have a maximum size of 64Kbytes only.
3. Currently, XMD with simulator target does not support the simulation of OPB peripherals.

XMD Tcl commands

In the Tcl interface mode, **xmd** starts a Tcl shell augmented with xmd commands. All **xmd** Tcl commands start with an 'x' and can be listed from xmd by typing "xhelp". These commands may be used in verification scripts. **xmdterm.tcl** in the MicroBlaze **bin/** directory may be used as an example. For interactive debugging without mb-gdb, use the commands in the **XMDTERM** section below.

xrmem <addr> [num]

Read num bytes or 1 byte from memory address <addr>

xwmem <addr> <value>

Write a 8-bit byte *value* at the specified memory *addr*.

xrreg [<reg>]

Read all registers or only register number **reg.**

xwreg <reg> <value>

Write a 32-bit *value* into register number *reg*

xdownload <filename>

Download the given ELF file onto the current target's memory. Note that NO Bounds checking is done by xmd, except preventing writes into xmdstub area (address 0x0 to 0x400).

xcontinue [<addr>]

Continue execution from the current PC or from the optional address argument. While the program is running, users could send Break or Reset signals to MicroBlaze by pressing 'b' for External Break signal, 'r' for Processor Reset, 's' for System Reset, 'n' for a Non-maskable Break signal or 'q' to quit xmd. Break signal is expecially useful while debugging interrupts, as this will freeze the running program and let the user see the status and debug it. See below for more details about signals.

xstep

Single step one MicroBlaze instruction. If the PC is at an IMM instruction the next instruction is executed as well. During a single step, interrupts are disabled by keeping the BIP flag set. Use xcontinue with breakpoints to enable interrupts while debugging.

xbreakpoint <addr>

Set a breakpoint at the given address. Note - Breakpoints on instructions immediately following `imm` instruction can lead to undefined results.

xremove <addr>

Remove breakpoint at given address.

xlist

List all the breakpoint addresses.

xdisassemble <inst>

Disassemble and display one 32-bit instruction.

xsignal <signal>

Send a signal to a hardware target. This is only supported by the JTAG UART when the debug signals for Processor Break, Reset and System reset are connected to MicroBlaze and the OPB bus. Platform Generator automatically connects these signals by default. Supported signals are listed in the following table.

Table 1: **XMD Hardware target signals**

Signal Name (value)	Description
Processor Break (0x20)	Raises the Brk signal on MicroBlaze using the JTAG UART Ext_Brk signal. It sets the Break-in-Progress (BIP) flag on MicroBlaze and jumps to addr 0x18. From xmd, this will allow a user to stop a running program (while in continue) and examine the processor status.
Non-maskable Break (0x10)	Similar to the Break signal but works even while the BIP flag is already set. Refer the MicroBlaze ISA documentation for more information about the BIP flag.
System Reset (0x40)	Resets the entire system by sending an OPB Rst using the JTAG UART Debug_SYS_Rst signal.
Processor Reset (0x80)	Resets MicroBlaze using the JTAG UART Debug_Rst signal.

xstats [options]

Display the simulation statistics for the current session. 'reset' option can be provided to reset the simulation statistics.

xhelp

List all xmd Tcl commands.

xmdterm commands

xmdterm.tcl script in the installation directory provides commands for doing assembly level debugging using the low level xmd commands. **xmdterm.tcl** is automatically loaded by xmd on startup. Powerful verification scripts can be written in Tcl based on the xmdterm script. User scripts with helper commands can be loaded into xmd by using the Tcl command "*source script.tcl*". Refer the Tcl documentation at the Tcl Developer site for more information on writing Tcl scripts and custom commands.

The debugging commands provided by xmdterm.tcl are listed below

Table 2: **Assembly level debugging commands**

command [options]	Description
rrd	Read Registers
rwr <reg_num> <word>	Write Register
mrd <address> <num_words>	Read Memory
mwr <address> <word>	Write Memory
dis [<address>]]<num_words>]	Disassemble
con [<addr>]	Continue from current PC
stp [<n instrns>]	Single Step one Instruction
bps <addr>	Set Breakpoint
bpr <addr>	Remote Breakpoint
bpl	List Breakpoints
dow <filename>	Download Elf File
help	List all commands

XILINX®

Device Drivers and Libraries

MicroBlaze Libraries

April 4, 2002

Summary

This document describes the organization of Xilinx Libraries and the interaction of its components with the user application. Xilinx provides two libraries, one for math functions **(libm)** and the other for C language support **(LibXil)**.

Overview

The C support library consists of the following components:

- Standard C libraries: **newlib libc**
- Xilinx file support functions **LibXil File**
- Xilinx memory file system **LibXil Mfs**
- Xilinx networking support **LibXil Net**
- Xilinx device drivers **LibXil Driver**

Most of the routines in the library are written in C and can be ported to any platform. The Library Generator (libgen) configures the libraries for a MicroBlaze, using the attributes defined in the Microprocessor Software Specification (MSS) file.

The math library is an enhancement over the newlib math library **libm.a .**

Library Organization

The structure of **LibXil** is outlined in Figure 1. The user application calls routines implemented in **LibXil** and/or **libm**. In addition to the standard C routines supported by **libc.a**, Xilinx library LibXil contains the following modules:

1. Stream based file system and device access (LibXil File)
2. Memory based file system (LibXil Mfs)
3. Networking application support (LibXil Net)
4. Device drivers (LibXil Driver)

Components such as LibXil Mfs can be accessed directly by the user. These routines can also be accessed through LibXil File.

Some of the library modules interact with drivers. These drivers are provided in the MicroBlaze Development Kit and are configured by libgen. These drivers form the Driver module of the LibXil library.

These libraries and include files are created in the current project's **`lib`** and **`include`** directories respectively. The **-I** and **-L** options of mb-gcc should be used to add these directories to its library search paths. Please refer to Microprocessor Software Specification Documentation and Library Generator documentation for more information.

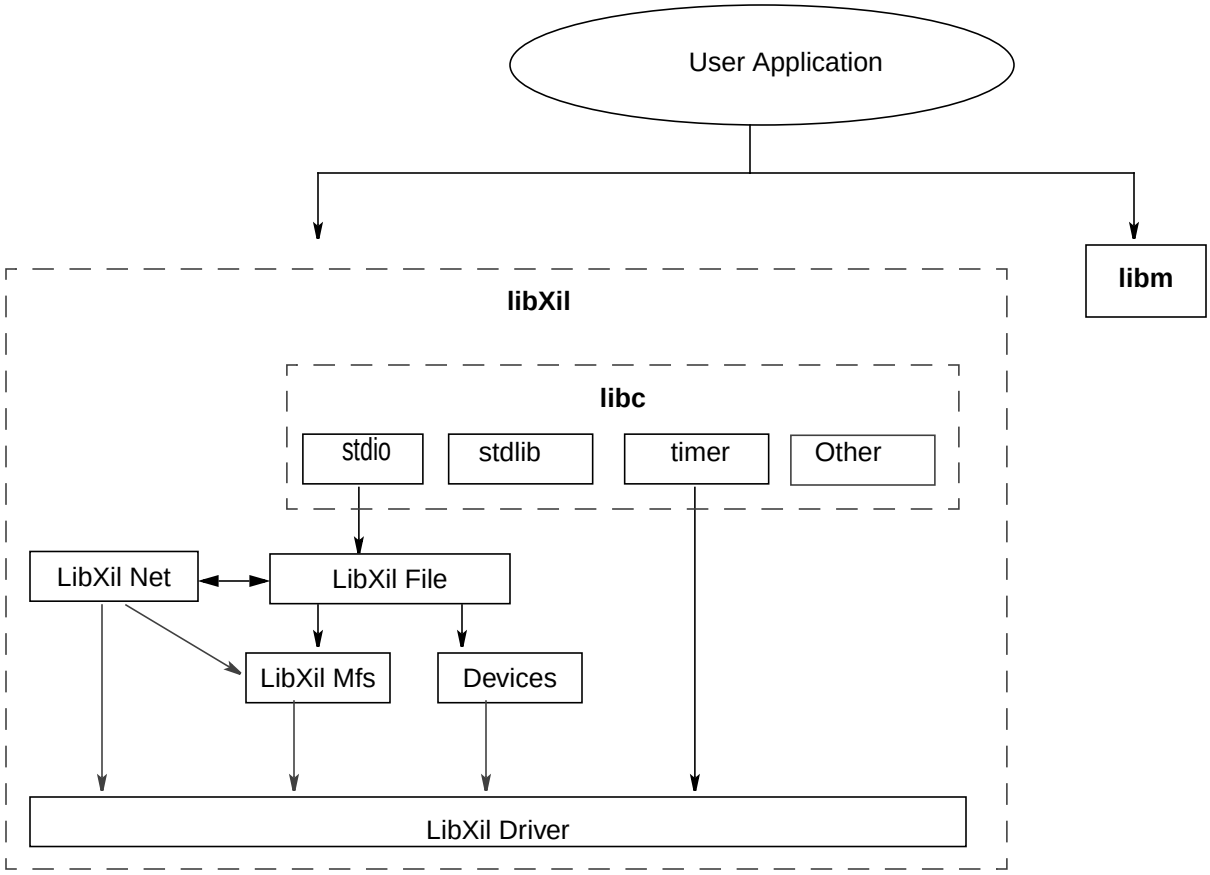


Figure 1: **Structure of LibXil library**

Library Customization

The standard newlib libc contains dummy functions for most of the operating system specific function calls such as **open, close, read, write etc**. These routines are included in the **libgloss** component of the standard libc library. The LibXil File module contains routines to overwrite these dummy functions. The routines interact with file systems such as Xilinx Memory File System[1] and peripheral devices [2] such as UART, UARTLITE and GPIO.

LibXil Net routines provide support for networking applications via the ethernet. This module is discussed more in details in the LibXil Net document. The module LibXil Net needs some support from the file system and hence calls other routines from the LibXil File and/or the LibXil Mfs modules. On the other hand, if an application requires opening files over the network, routines from the LibXil File module will need the support of the LibXil Net.

This highlights the need of a complete library solution, which is provided by Xilinx Libraries LibXil.

Libgen is used to tailor the library compilation for a particular project using attributes in the MSS. These attributes are described in the LibXil File and LibXil Mfs documents.

1. For more information on Memory File System, please refer to the document on LibXil Mfs
2. For more information on Device Drivers, please refer to the document on LibXil Driver

LibXil Standard C Libraries

April 1, 2002

Summary

This document describes the standard C libraries provided with the MicroBlaze Distribution Kit. These standard libraries are enhancements to the newlib libc and libm libraries to generate better code for MicroBlaze applications.

Standard C Functions (libc)

The MicroBlaze libraries provide standard C library functions, while the MicroBlaze drivers implement functions to access the peripherals.

List of Standard C Library (libc.a) Files

The standard C library libc.a contains the standard C functions compiled for MicroBlaze. For a list of all the supported functions refer to the following files in **`MICROBLAZE/include`**

`_ansi.h`	`fastmath.h`	`machine/`	`reent.h`	`stdlib.h`	`utime.h`
`_syslist.h`	`fcntl.h`	`malloc.h`	`regdef.h`	`string.h`	`utmp.h`
`ar.h`	`float.h`	`math.h`	`setjmp.h`	`sys/`	
`assert.h`	`grp.h`	`paths.h`	`signal.h`	`termios.h`	
`ctype.h`	`ieeefp.h`	`process.h`	`stdarg.h`	`time.h`	
`dirent.h`	`limits.h`	`pthread.h`	`stddef.h`	`unctrl.h`	
`errno.h`	`locale.h`	`pwd.h`	`stdio.h`	`unistd.h`	

Programs accessing standard C library functions must be compiled as follows:

```
mb-gcc <C files>
```

The libc library is included automatically.

The **`-lm`** option should be specified for programs that access libm math functions.

Refer to the MicroBlaze ABI documentation for information on the C Runtime Library.

Input/Output Functions

The MicroBlaze libraries contains standard C functions for I/O;such as printf and scanf. These are large and may not be suitable for embedded processors. In addition, the MicroBlaze library provides the following smaller I/O functions:

void print (char *)

This function prints a string to the peripheral designated as standard output in the MSS file.

void putnum (int)

This function converts an integer to a hexadecimal string and prints it to the peripheral designated as standard output in the MSS file.

void xil_printf (const *char ctrl1, ...)

This funtion is similar to printf but much smaller in size (only 1KB). It does not have support for floating point numbers. xil_printf also does not support printing of long long (i.e 64 bit numbers).

The prototypes for these functions are in **`stdio.h`**.

Please refer the Microprocessor Software Specification documentation for information on setting the standard input and standard output devices for a system.

Memory Management Functions

Memory management routines such as **`malloc, calloc`** and **`free`** can run the gamut of high functionality (with associated large size) to low functionality (and small size). This version of MicroBlaze only supports a simple, small `malloc`, and a dummy `free`. Hence when memory is allocated using `malloc`, this memory can not be reused.

The **`_STACK_SIZE_`** option to mb-gcc specifies the total memory allocated to stack and heap. The stack is used for function calls, register saves and local variables. All calls to malloc allocate memory from heap. The stack pointer initially points to the bottom (high end) of memory, and grows toward low memory while the heap pointer starts at low memory and grows towards high memory. The size of the heap cannot be increased at runtime. The return value of `malloc` must always be checked to ensure that it could actually allocate the memory requested.

Please note that whereas `malloc` checks that the memory it allocates does not overwrite the current stack pointer, updates to the stack pointer do not check if the heap is being overwritten.

Increasing the **`_STACK_SIZE_`** may be one way to solve unexpected program behavior. Refer to the Linker/Loader Options section of the MicroBlaze GNU Compiler Tools document for more information on increasing the stack size.

Arithmetic Operations

Integer Arithmetic

Integer addition and subtraction operations are provided in hardware. By default, integer multiplication is done in software using the library function **`mulsi3_proc`**. Integer multiplication is done in hardware if the mb-gcc option **`-mno-xl-soft-mul`** is specified.

Integer divide and mod operations are done in software using the library functions **`divsi3_proc`** and **`modsi3_proc`**.

Double precision multiplication, division and mod functions are carried out by the library functions **`muldi3_proc, divdi3_proc`** and **`moddi3_proc`** respectively.

Floating Point Arithmetic

All floating point addition, subtraction, multiplication and division operations are also implemented using software functions in the C library.

LibXil File

April 1, 2002

Summary

Xilinx libraries provide block access to file systems and devices using standard calls such as **open,close, read,write etc**. These routines form the **LibXil File Module** of the Libraries.

A system can be configured to use LibXil File module through the Library Generator (libgen)

Overview

The LibXil library provides block access to files and devices through the **LibXil File** module. This module provides standard routines such as **open, close, read, write etc** to access file systems and devices.

The module **LibXil File** can also be easily modified to incorporate additional file systems and devices. LibXil File implements a subset of operating system level functions.

Module Usage

A file or a device is opened for read and write using the open call in the library. The library maintains a list of open files and devices. Read and write commands can be issued to access blocks of data from the open files and devices.

Module Routines

LibXil File includes various routines for file and device access as shown in Table 1.

Table 1: **Routines Provided by LibXil File Module**

Functions
int open (const char *name, int flags, int mode)
int close (int fd)
int read (int fd, char* buf, int nbytes)
int write (int fd, char* buf, int nbytes)
int lseek (int fd, long offset, int whence)
int chdir (const char *buf)
const char* getcwd (void)

int open (const char *name, int flags, int mode)

Parameters	name refers to the name of the device/file
	flags refers to the permissions of the file. This field is not meaningful for a device.
	mode indicates read, write or append mode.
	Returns file/device descriptor (fd) assigned by LibXil File
Description	The routine registers the device or the file in the local device table and calls the underlying open function for that particular file or device.
Includes	xilfile.h
	mbio.h

int close (int fd)

Parameters	fd refers to the file descriptor assigned by open()
Description	Close the file/device associated with fd.
Includes	xilfile.h
	mbio.h

int read (int fd, char* buf, int nbytes)

Parameters	fd refers to the file descriptor assigned by open()
	buf refers to the destination buffer where the contents of the stream should be copied
	nbytes: Number of bytes to be copied
	Returns the number of bytes read.
Description	Read nbytes from the file/device pointed by the file descriptor fd and store it in the destination pointed by buf.
Includes	xilfile.h
	mbio.h

int write (int fd, char* buf, int nbytes)

Parameters	fd refers to the file descriptor assigned by open()
	buf refers to the source buffer
	nbytes: Number of bytes to be copied
	Returns the number of bytes written to the file.
Description	Write nbytes from the buffer, buf to the file pointed by the file descriptor fd
Includes	xilfile.h
	mbio.h

int lseek (int fd, long offset, int whence)

Parameters	fd: file descriptor returned by open
	offset: Number of bytes to seek
	whence: Location to seek from. This parameter depends on the underlying File System being used.
	Returns : New file pointer location
Description	The lseek() system call moves the file pointer for fd by offset bytes from whence.
Includes	xilfile.h
	mbio.h

int chdir (char* newdir)

Parameters	newdir: Destination directory
	Returns the same value as returned by the underlying file system. -1 for failure.
Description	Change the current directory to newdir
Includes	xilfile.h
	mbio.h

const char* getcwd (void)

Parameters	Returns the current working directory.
Description	Get the absolute path for the current working directory.
Includes	xilfile.h
	mbio.h

Libgen Support

LibXil File Instantiation

User applications can access underlying file systems and devices or make use of the **LibXil File** module to integrate with file systems and devices.

The Libgen attribute **LIBRARY** indicates that **LibXil File** module should be compiled into the project specific Libraries.

To use Memory File System with LibXil File component, the following code is used in the MSS file.

```
SELECT FILESYS XilMfs
CSET attribute MOUNT= "/home/"
CSET attribute LIBRARY = XilFile
END
```

To access a device through Xilfile the following snippet is used in the mss file.

```
SELECT INSTANCE myuart
CSET attribute DRIVER = drv_uartlite
CSET attribute DRIVER_VER = 1.00.b
CSET attribute LIBRARY = XilFile
END
```

Table 2: **List of peripherals supported by LibXil File**

Peripheral Name
OPB_UARTLITE
OPB_UART
OPB_GPIO

System Initialization

Libgen generates a file system initialization file (**xilfile_init.c**) is compiled into the LibXil library. The file contains configuration information and data structures required by the **LibXil File** module, such as the Device tables and the File System table. STDIN, STDOUT and STDERR devices are also configured for use with the module.

Limitations

LibXil File module currently enforces the following restrictions :

- Only one instance of a File System can be mounted. This file system and the mount point has to be indicated in the Microprocessor Software Specification (MSS) file.
- Files cannot have names starting with **/dev**, since it is a reserved word to be used only for accessing devices
- Currently LibXil File has support only for 1 filesystem (LibXil Memory File System) and 3 devices (UART, UARTlite and GPIO). Others can be added easily.
- Only devices can be assigned as STDIN, STDOUT and STDERR

LibXil Memory File System (MFS)

April 1, 2002

Summary

This document describes the Memory File System (MFS). This file system resides on the memory and can be accessed throught LibXil File module or directly. Memory File System is integrated with a system using the Library Generator.

Overview

The Memory File System (MFS) component, **LibXil MFS**, provides users the capability to manage platform's memory in the form of file handles. Users can create directories, and can have files within each directory. The file system can be accessed from the high level C-language through function calls specific to the file system. Alternatively, the users can also manage files through the standard C language functions like **open** provided in **XilFile.**

MFS Functions

This section presents a list of functions provided by the MFS. Table 1 provides the function names with signature at a glance. C-like access.

Table 1: **MFS functions at a glance**

Functions
void mfs_init_fs (void)
int mfs_change_dir (char *newdir)
int mfs_delete_file (char *filename)
int mfs_create_dir (char *newdir)
int mfs_delete_dir (char *newdir)
int mfs_rename_file (char *from_file, char *to_file)
int mfs_exists_file (char *filename)
int mfs_get_current_dir_name (char *dirname)
int mfs_get_usage(int *num_blocks_used, int *num_blocks_free)
int mfs_file_open (char *filename, int mode)
int mfs_file_read (int fd, char *buf, int buflen)
int mfs_file_write (int fd, char *buf, int buflen)
int mfs_file_close(int fd)
int mfs_file_lseek (int fd, int offset, int whence)
int mfs_ls (void)
int mfs_cat (char *filename)
int mfs_copy_stdin_to_file (char *filename)
int mfs_file_copy (char *from_file, char *to_file)

Detailed summary of MFS Functions

int mfs_init_fs (void)

Parameters	None
	Return 1 for success and 0 for failure
Description	Initialize the memory file system. This function must be called before any file system operation.
Includes	xilmfs.h
	mbio.h

int mfs_change_dir (char *newdir)

Parameters	newdir is the chdir destination.
	Return 1 for success and 0 for failure
Description	If newdir exists, make it the current directory of MFS. Current directory is not modified in case of failure.
Includes	xilmfs.h
	mbio.h

int mfs_delete_file (char *filename)

Parameters	filename: file to be deleted
	Return 1 for success and 0 for failure
Description	Delete filename from its directory.
Includes	xilmfs.h
	mbio.h

int mfs_create_dir (char *newdir)

Parameters	newdir: Directory name to be created
	On success, return index of new directory in the file system
	On failure, return 0
Description	Create a new empty directory called newdir inside the current directory.
Includes	xilmfs.h
	mbio.h

int mfs_delete_dir (char *dirname)

Parameters	dirname: Directory to be deleted On success, return 1 On failure, return 0
Description	Delete the directory dirname, if it exists and is empty,
Includes	xilmfs.h mbio.h

int mfs_rename_file (char *from_file, char *to_file)

Parameters	from_file: Original filename to_file: New file name On success, return 1 On failure, return 0
Description	Rename from_file to to_file. Rename works for directories as well as files. Function fails if to_file already exists.
Includes	xilmfs.h mbio.h

int mfs_exists_file (char *filename)

Parameters	filename: file/directory to be checked for existance Return 0 : if filename doesnot exist Return 1 : if filename is a file Return 2 : if filename is a directory
Description	Check if the file/directory is present in current directory.
Includes	xilmfs.h mbio.h

int mfs_get_current_dir_name (char *dirname)

Parameters	dirname: Current directory name is returned in this pointer If Success return 0 If Failure return 1
Description	Return the name of the current directory in a pre allocated buffer, dirname, of at least 16 chars.Note that it does not return the absolute path name of the current directory, but just the name of the current directory
Includes	xilmfs.h mbio.h

int mfs_get_usage (int *num_blocks_used, int *num_blocks_free)

Parameters	num_blocks_used: Number of blocks_used num_blocks_free : Number of free blocks If Success return 0 If Failure return 1
Description	Get the number of used blocks and the number of free blocks in the file system through pointers.
Includes	xilmfs.h mbio.h

int mfs_file_open (char *filename, int mode)

Parameters	filename : file to be opened mode: Read/Write or Create mode. Return the index of filename in the array of open files or -1 on failure.
Description	Open file filename with given mode. The function should be used for files and not directories: MODE_READ, no error checking is done(if file or directory). MODE_CREATE creates a file and not a directory. MODE_WRITE fails if the specified file is a DIR.
Includes	xilmfs.h mbio.h

int mfs_file_read (int fd, char *buf, int buflen)

Parameters	fd: File descriptor return by open buf: Destination buffer for the read buflen: Length of the buffer If Success return number of bytes read. If Failure return 1
Description	Read buflen number bytes and place it in buf. fd should be a valid index in open_files array, pointing to a file, not a directory. buf should be a pre-allocated buffer of size buflen or more. If fewer than buflen chars are available then only that many chars are read.
Includes	xilmfs.h mbio.h

int mfs_file_write (int fd, char *buf, int buflen)

Parameters	fd: File descriptor return by open buf: Source buffer from where data is read buflen: Length of the buffer If Success return 1 If Failure return 1
Description	Write buflen number of bytes from buf to the file. Rfd should be a valid index in open_files array. buf should be a pre-allocated buffer of size buflen or more.
Includes	xilmfs.h mbio.h

int mfs_file_close (int fd)

Parameters	fd: File descriptor return by open If Success return 1 If Failure return 1
Description	Close the file pointed by fd. The filesystem regains the fd and uses it for new files.
Includes	xilmfs.h mbio.h

int mfs_file_lseek (int fd, int offset, int whence)

Parameters	fd: File descriptor return by open offset : Number of bytes to seek whence: File system dependent mode: If whence is MFS_SEEK_END, the offset can be either 0 or negative, otherwise offset should be non-negative. If whence is MFS_SEEK_CURR : the offset is calculated from the current location If whence is MFS_SEEK_SET: the offset is calculated from the start of the file Return 1 on success and 0 on failure.
Description	Seek to a given offset within the file at location fd in open_files array. It is an error to seek before beginning of file or after the end of file.
Includes	xilmfs.h mbio.h

int mfs_ls (void)

Parameters	Return 1 on success and 0 on failure.
Description	List contents of current directory on STDOUT.
Includes	xilmfs.h mbio.h

int mfs_cat (char *filename)

Parameters	filename: File to be displayed Return 1 on success and 0 on failure.
Description	Print the file to STDOUT.
Includes	xilmfs.h mbio.h

int mfs_copy_stdin_to_file (char *filename)

Parameters	filename: Destination file. Return 1 on success and 0 on failure.
Description	Copy from STDIN to named file.
Includes	xilmfs.h mbio.h

int mfs_file_copy (char *from_file, char *to_file)

Parameters	from_file Source file to_file : Desitnation file Return 1 on success and 0 on failure.
Description	Copy from_file to to_file. It fails if to_file already exists, or if either could not be opened.
Includes	xilmfs.h mbio.h

C-like access

The user can choose not to deal with the details of the file system by using the standard C-like interface provided by **Xil File.** It provides the basic C stdio functions like **open**, **close**, **read**, **write**, and **seek**. These functions have identical signature as those in the standard ANSI-C. Thus any program with file operations performed using these functions can be easily ported to MFS by interfacing the MFS in conjunction with libXfiles.

LibGen Customization

Memory file system can be integrated with a system using the following snippet in the mss file. The memory file system should be instantiated with the name **XilMfs**

```
SELECT FILESYS XilMfs
CSET attribute MOUNT = /
CSET attribute LIBRARY = XilFile
END
```

Table 2: **Attributes for including Memory File System**

Attributes	Description
MOUNT	Mount name for the file system.
LIBRARY	Set this attribute to **XilFile** if the file system is accessed through XilFile component of MicroBlaze Libraries

LibXil Net

April 1, 2002

Summary

This document describes the network library for MicroBlaze, libXilNet. The library includes functions to support the TCP/P stack and the higher level application programming interface(APIs).

Overview

The MicroBlaze networking library, **libXilNet**, allows MicroBlaze to connect to the internet. LibXilNet includes functions for handling the TCP/IP stack protocols. It also provides a simple set of Application Programming Interface (APIs) functions enabling network programming. This document describes the various functions of LibXilNet.

Protocols Supported

LibXilNet supports drivers and functions for the Media Access layer and protocols of TCP/IP stack. The following list enumerates them.

- Media Access Layer Drivers (MAC)
- Ethernet Encapsulation (RFC 894)
- Address Resolution Protocol (ARP - RFC 826)
- Internet Protocol (IP - RFC 791)
- Internet Control Management Protocol (ICMP - RFC 792)
- Transmission Control Protocol (TCP - RFC 793)
- User Datagram Protocol (UDP - RFC 768)

Footprint

LibXilNet is a small library which is geared for embedded systems. Table 1 shows the output from `mb-size` on libXilNet. It gives the code + data footprint of memory requirements.

Table 1: **mb-size output for libXilNet**

text	data	bss	dec	filename
704	0	0	704	mac.o
404	0	0	404	eth.o
432	0	0	432	arp.o
1140	4	0	1144	ip.o
200	0	0	200	icmp.o
420	0	0	420	udp.o
1880	13	0	1893	tcp.o
1332	16	1128	2476	packet.o

The total memory requirements for libXilNet is 7673 ~ 8k bytes.

Library Architecture

Figure 1 gives the architecture of libXilNet. Higher Level applications like HTTP server, TFTP (Trivial File Transfer Protocol), PING etc., uses API functions to use the libXilNet library.

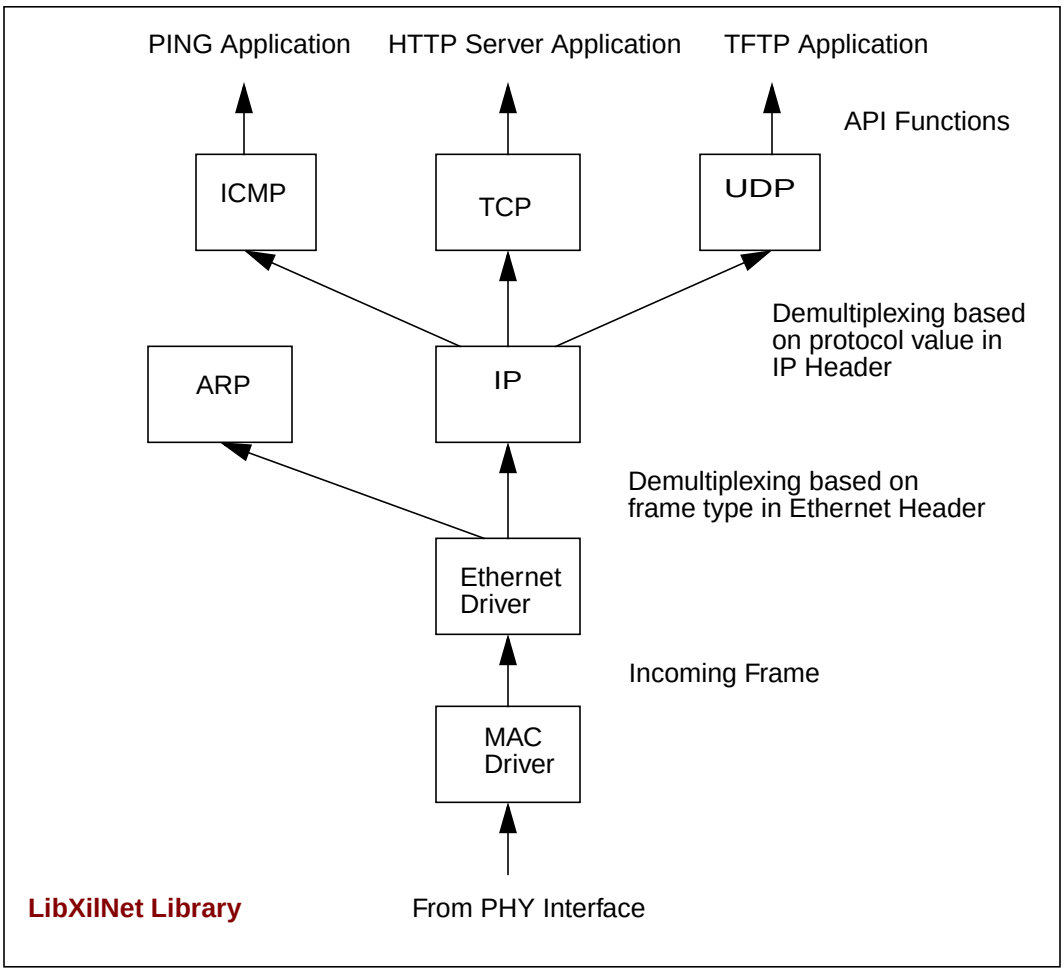


Figure 1: **Schematic diagram of LibXilNet Architecture**

Protocol Function Description

A detailed description of the drivers and the protocols supported is given below.

Media Access Layer (MAC) Drivers

MAC drivers are provided for receiving and sending the ethernet frames over the PHY interface. The MAC hardware provides the Cyclic Redundancy Check (CRC) on the frames received/sent on the PHY layer. The 48-bit hardware address needs to be programmed onto the MAC before it can be used. A MAC initializing function is provided to achieve the same.

Ethernet Drivers

Ethernet drivers performs the encapsulation/removal of ethernet headers on the payload in accordance with the RFC 894. Based on the type of payload (IP or ARP), the drivers call the corresponding protocol callback function.

ARP (RFC 826)

Functions are provided for handling ARP requests. An ARP request (for the 48-bit hardware address) would be responded with the 48-bit ethernet address in the ARP reply. Currently, ARP request generation for a desired IP address is not supported.

IP (RFC 791)

IPv4 datagrams are used by the level protocols like ICMP, TCP, UDP for receiving/sending data. A callback function is provided for ethernet drivers which would be invoked whenever there is a IP datagram as a payload in ethernet frame. Minimal processing of source IP address check is performed before the corresponding higher level protocol's (ICMP, TCP, UDP) is called. Checksum would be calculated on all the outgoing IP datagrams before calling the ethernet callback function for sending the data. IP address for MicroBlaze needs to be programmed before using it for communication. An IP address initializing function is provided. Refer to the table describing the various routines for further details on the function. Currently no IP fragmentation would be performed on the outgoing datagrams.

ICMP (RFC 792)

ICMP functions handling only the echo requests (ping requests) are provided. Echo requests would be responded with the appropriate requirements as per the RFC.

UDP (RFC 768)

UDP is a connectionless protocol. The UDP callback function, called from IP layer, would perform the minimal check of source port and strip off the UDP header. Checksum calculation would be performed on the outgoing UDP datagram. Currently, only one UDP connection is supported.

TCP (RFC 793)

TCP is a connection-oriented protocol. Callback functions are provided for sending and receiving TCP packets. TCP maintains connections as a finite state machine. On receiving a TCP packet, minimal check of source port correctness is done before taking the necessary action for the connection based on the current machine state. Checksum is calculated on all outgoing TCP packets. Currently only one TCP connection is supported.

API

Functions for sending and receiving UDP and TCP packets are provided. High level network applications need to use these functions for performing data communication. Refer to the table describing the routines in libXilNet for further details.

Current Restrictions

Certain restrictions apply to the MicroBlaze libXilNet library software. These are

- Only one connection for TCP and UDP supported currently. This means the applicaiton cannot open more than one TCP or UDP port at any instance..
- Only one protocol port can operate at any instance - LibXilNet currently supports either a TCP port or a UDP port based connection at any instance.
- Only server functionalities for ARP - This means ARP requests are not being generated from MicroBlaze
- No timers in TCP - Since there are no timers used, every "send" over a TCP connection waits for an "ack" before performing the next "send".

Functions of LibXilNet

The following table gives the list of functions in libXilNet and their descriptions.

Table 2: **Functions in LibXilNet**

Functions	Description
`int xilnet_recvfrom (unsigned char* buf, int len)`	Receives data(maximum length of `len`) from the UDP port in `buf` and returns the number of bytes received. This function is used by the application.
`int xilnet_sendto (unsigned char* buf, int len)`	Sends data of length `len` in `buf` on the UDP port and returns the number of bytes sent. This function is used by the application.
`int xilnet_recv (unsigned char* buf, int len)`	Receives data(maximum length of `len`) from the TCP port in `buf`and returns the number of bytes received. This function is used by the application.
`int xilnet_send(unsigned char* buf, int len)`	Sends data of length `len` in `buf` on the TCP port and returns the number of bytes sent. After sending in data, it waits for an acknowledgement of the data sent before returning. This function is used by the application.
`void xilnet_reset_peer (void)`	Resets the connection to the peer for accetping a new connection. This function has to be called in the application after the entire transaction on the connection is over for enabling future connections to peer.
`void xilnet_close (void)`	Closes the current TCP connection. This function has to be called from the application for a smooth termination of the connection after a connection is done with the communication.
`void xilnet_mac_init (unsigned char* mac_addr)`	This function has to be called in the application before starting any communicaton. It sets the 48-bit MAC address to the value in `mac_addr`. Note that each element of `mac_addr` array has a byte of the 48-bit MAC address
`void xilnet_mac_recv_frame (unsigned char* frame, int len)`	Receives ethernet frame (maximum length `len`) on the PHY interface in buffer `frame`. This function is called from the ethernet drivers.
`void xilnet_mac_send_frame (unsigned char* frame, int len)`	Sends ethernet frame, `frame`, of length `len` onto the PHY interface. This function is called from the ethernet drivers.

Table 2: **Functions in LibXilNet**

Functions	Description
`void xilnet_eth_recv_frame (unsigned char* frame, int len)`	Receives ethernet frame from the MAC, strips ethernet header and calls either `ip` or `arp` callback function based on frame type. This function is called from `receive/send` functions of API. The function receives frame of maximum length `len` in buffer `frame`.
`void xilnet_eth_send_frame (unsigned char* frame, int len, void *daddr, unsigned short type)`	Creates ethernet header for payload `frame` of length len, with destination ethernet address `daddr`, and frame type, `type`. Sends the ethernet frame to the MAC. This function is called from `receive/send` (both versions) functions.
`int xilnet_arp (unsigned char* buf, int len)`	This is the `arp` callback function. It gets called by the ethernet driver for `arp` frame type. The `arp` packet is copied onto the `buf` of length `len`.
`void xilnet_arp_reply (unsigned char* buf, int len)`	This function sends the `arp` reply, present in `buf` of length `len`, for `arp` requests. It gets called from the `arp` callback function for `arp` requests.
`void xilnet_ip_init (unsigned char* ip_addr)`	This function initializes the `ip` address for MicroBlaze to the address represented in `ip_addr` as a dotted decimal string. This function has to be called in the application before any communication.
`int xilnet_ip (unsigned char* buf, int len)`	This is the `ip` callback function. It gets called by the ethernet driver for `ip` frame type. The `ip` packet is copied onto the `buf` of length `len`. This function calls in the appropriate protocol callback function based on the protocol type.
`int xilnet_ip_header (unsigned char* buf, int len, int proto)`	This function fills in the `ip` header from start of `buf`. `ip` packet is of length `len`. `proto` is used to fill in the protocol field of `ip` header. This function is called from the `receive/send` (both versions) functions.
`int xilnet_ip_calc_chksum (unsigned char* buf, int len)`	This function calculates and fills the `checksum` for the `ip` packet `buf` of length `len`. This function is called from the `ip header` creation function.
`int xilnet_udp (unsigned char* buf, int len)`	This is the `udp` callback function. This is called when `ip` receives a `udp` packet. This function checks for valid `udp` port and strips the `udp` header.
`void xilnet_udp_header (unsigned char* buf, int len)`	This function fills in the `udp` header from start of `buf`. `udp` packet is of length `len`. This function is called from the `receivefrom/sendto` functions.

Table 2: **Functions in LibXilNet**

Functions	Description
`unsigned short xilnet_udp_tcp_calc_chksum (unsigned char* buf, int len, unsigned char* saddr, unsigned char* daddr, unsigned short proto)`	This function calculates and fills the `checksum` for the `udp/tcp` packet `buf` of length `len`. source ip address (`saddr`), destination ip address(`daddr`) and protocol (`proto`) are used in the checksum calculation for creating the pseudo header. This function is called from either `udp header` or `tcp header` creation function.
`int xilnet_tcp (unsigned char* buf, int len)`	This is the `tcp` callback function. This is called when `ip` receives a `tcp` packet. This function checks for valid `tcp` port and strips the `tcp` header. It maintains a finite state machine for the connection.
`void xilnet_tcp_sendack (unsigned char* buf, int len)`	This function sends an `ack` for a received `tcp` packet. This is called from the `tcp` callback function when it receives a connection termination packet.
`void xilnet_tcp_header (unsigned char* buf, int len, unsigned char flags)`	This function fills in the `tcp` header from start of `buf`. `tcp` packet is of length `len`. It sets the `flags` in tcp header. This function is called from the `receive/send` functions.
`int xilnet_icmp (unsigned char* buf, int len)`	This is the `icmp` callback function. This is called when `ip` receives a `icmp` echo request packet (ping request). This function checks only for a echo request and sends in a `icmp` echo reply.
`void xilnet_icmp_echo_reply (unsigned char* buf, int len)`	This functions fills in the `icmp` header from start of `buf`. `icmp` packet is of length `len`. It sends the `icmp` echo reply by calling the `ip, ethernet` send functions and resets the peer for a new connection. This function is called from the `icmp` callback function.

LibXil Driver

April 4, 2002

Summary

This document describes the drivers available for various peripheral devices. The drivers provide mechanism to communicate with the device from the software program.

Overview

The drivers provide functions to access peripheral devices. These drivers are automatically configured by libgen for every project based upon the Microprocessor Software Specification (MSS) file. These tailored driver files are compiled and archived into LibXil and the sources are saved in the current project's libsrc directory.

Avaiable Device Drivers

Simple device drivers are provided with all Xilinx supplied OPB Peripherals. These drivers are present in the **`drivers`** directory in the installation area. This section describes the functions available for each device driver. The Library Generator tool automatically configures the peripheral drivers included in the MSS file. The source code for the configured drivers can be found in the user project's **libsrc** directory. Table 1 shows the driver name and thier latest available versions.

Table 1: **List of Drivers**

Peripheral	Driver Directory	Version
General Purpose I/O	drv_gpio	v1.00.b
IIC Bus	drv_iic	v1.00.b
Interrupt Controller	drv_intc	v1.00.b
JTAG UART	drv_jtag_uart	v1.00.b
OPB Arbiter	drv_opb_arbiter	v1.02.b
Serial Port Interface	drv_spi	v1.00.b
Timebase/Watchdog Timer	drv_timebase_wdt	v1.00.b
Timer/Counter	drv_timer	v1.00.b
UART Lite	drv_uartlite	v1.00.b

Data Types

All the driver software use a standard data types which have been defined in a file called globaltypes.h. Table 2 presents what each data type means. A ‘U’ at the beginning of the data type means it if of type unsigned. A number at the end of the data type represents the number of bits needed to store that data type. In this document, a word stands for a 32-bit data, either signed or unsigned. Similarly, half-word stands for 16-bit and a byte stands for 8-bit data.

Table 2: **Global Typedefs**

Typedef	Data Type
UINT8	unsigned char
UINT16	unsigned short
UINT32	unsigned int
UINT64	unsigned long
INT8	char
INT16	short
INT32	int
INT64	long
BOOL	enum {FALSE=0, TRUE=1}

[1]Driver Usage

A peripheral that is instantiated in the MHS file must have the driver specified in the MSS file using the DRIVER keyword. For e.g., if the MHS file has the GPIO peripheral instantiated in this manner:

```
SELECT SLAVE opb_gpio
CSET attribute INSTANCE = mygpio
CSET attribute HW_VER = 1.00.a
....
```

Then the MSS file should associate the driver with this instance as:

```
SELECT INSTANCE mygpio
CSET attribute DRIVER = drv_gpio
CSET attribute DRIVER_VER = 1.00.a
....
```

Libgen then automatically configures this version of the driver for that peripheral.

The user can then include the files **gpio.h** and **mbio.h** to access the driver functions and other useful definitions such as base address of the peripheral, standard input and output base addresses and so on. The include files are specified in the sections on each peripheral driver function.

Please note that MicroBlaze also has interrupt enable routines that are needed if interrupt handling is required. These are present in the **drv_microblaze_v1_00_a** directory and can be accessed by setting the DRIVER and DRIVER_VER attributes.

1. This is the way to include the drivers available in your application.

Driver Functions

The driver functions that are provided are enumerated below for quick reference.

Table 3:

General Purpose I/O
UINT32 gpio_read_INT32 (**UINT32** base_addr)
void gpio_write_INT32 (**UINT32** base_addr, **UINT32** data)
void gpio_write_ctrl (**UINT32** base_addr, **UINT32** data)
IIC Bus
The IIC hardware core is a paid peripheral. Please contact Xilinx to obtain the same.
Interrupt Controller
void intc_disable_all_interrupts (**UINT32** base_addr)
void intc_disable_interrupt (**UINT32** base_addr, **UINT32** periph_int_priority_mask)
void intc_enable_interrupt (**UINT32** base_addr, **UINT32** periph_int_priority_mask)
void intc_interrupt_handler (**void**)
void intc_start (**UINT32** base_addr)
JTAG UART
Functions are similar to UART Lite peripheral, except the names of the functions are jtag_uart_<function name> instead of uartlite_<function name>
Serial Port Interface (SPI)
INT8 inbyte (**void**)
void outbyte **(INT8)**
void spi_write_control_reg(**UINT32** base_addr, **INT8** val)
UINT8 spi_read_control_reg(**UINT32** base_addr)
void spi_enable_device **(UINT32** base_addr)
void spi_disable_device (**UINT32** base_addr)
void spi_write_intr_enable_reg(**UINT32** base_addr, **UINT8** mask)
void spi_read_intr_enable_reg(**UINT32** base_addr)
void spi_write_intr_reg (**UINT32** base_addr, **UINT8** mask)
UINT8 spi_read_intr_reg(**UINT32** base_addr)
void spi_clear_all_intr (**UINT32** base_addr)
void spi_enable_all_intr(**UINT32** base_addr)
void spi_disable_all_intr(**UINT32** base_addr)
void spi_reset_fifo(**UINT32** base_addr)
INT8 spi_read_byte(**UINT32** base_addr)
void spi_write_byte(**UINT32** base_addr, **INT8** data)
void spi_set_options(**UINT32** base_addr, **UINT32** options)
UINT32 spi_get_options(**UINT32** base_addr)

Table 3:

void spi_set_slave_select(**UINT32** base_addr, **UINT32** slave_bit_pos)
UINT32 spi_get_slave_select(**UINT32** base_addr)
Timebase Watchdog Timer
void timebase_wdt_disable (**UINT32** base_addr)
void timebase_wdt_enable (**UINT32** base_addr)
UINT32 timebase_wdt_get_status (**UINT32** base_addr)
UINT32 timebase_wdt_get_timebase (**UINT32** base_addr)
void timebase_wdt_kick (**UINT32** base_addr)
void timebase_wdt_set_status0 (**UINT32** base_addr, **UINT32** status)
void timebase_wdt_set_status1(**UINT32** base_addr, **UINT32** status)
Timer/Counter
UINT32 get_elapsed_time (**UINT32** base_addr, **UINT32** timer_number)
void start_timer (**UINT32** base_addr, **UINT32** timer_number)
UINT32 timer_get_capture (**UINT32** base_addr, **UINT32** timer_number)
UINT32 timer_get_csr (**UINT32** base_addr, **UINT32** timer_number)
UINT32 timer_get_time (**UINT32** base_addr, **UINT32** timer_number)
void timer_set_compare (**UINT32** base_addr, **UINT32** timer_number, **UINT32** compare_value)
void timer_set_csr (**UINT32** base_addr, **UINT32** timer_number, **UINT32** status_value)
UART Lite
INT8 inbyte (**void**)
void outbyte (**INT8**)
void uartlite_disable_intr (**UINT32** base_addr)
INT32 uartlite_empty(**UINT32** base_addr)
void uartlite_enable_intr (**UINT32** base_addr)
INT32 uartlite_full (**UINT32** base_addr)
UINT32 uartlite_get_status (**UINT32** base_addr)
INT8 uartlite_read_byte (**UINT32** base_addr)
INT32 uartlite_is_intr_enabled (**UINT32** base_addr)
void uartlite_write_byte (**UINT32** base_addr, **INT8** ch)
void uartlite_set_control (**UINT32** base_addr, **UINT32** data)

General Purpose I/O Driver (gpio)

Gpio can be used to hook up I/O such as LEDs, seven segment displays and switches on the development board. The following driver functions are available for the gpio:

UINT32 gpio_read_INT32 (UINT32 base_addr)

Parameters	base_addr: Base address of gpio peripheral
Description	Read the GPIO data pins.
Includes	gpio.h mbio.h

void gpio_write_INT32 (UINT32 base_addr, UINT32 data)

Parameters	base_addr: Base address of gpio peripheral data: Word to be written to the GPIO data pins
Description	Write data to the GPIO data pins. This write has no effect on pins that have been programmed as input pins.
Includes	gpio.h mbio.h

void gpio_write_ctrl (UINT32 base_addr, UINT32 data)

Parameters	base_addr: Base address of gpio peripheral data: Word to be written to GPIO_TRI register
Description	Write data to the GPIO_TRI register of the gpio. Each I/O pin of the GPIO is individually programmable as an input or as an output. Writing a 0 to a bit of the GPIO_TRI register configures that bit as an output, and writing a 1 to a bit of the GPIO_TRI register configures that bit as an input.
Includes	gpio.h mbio.h

Example MHS File Entry

```
SELECT SLAVE opb_gpio
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = mygpio
CSET attribute C_BASEADDR = 0xFFFF0100
CSET attribute C_HIGHADDR = 0xFFFF01ff
CSET attribute C_AWIDTH   = 32
CSET attribute C_DWIDTH   = 32
CSET attribute C_GPIO_WIDTH = 16
CSET attribute C_ALL_INPUTS = 0
CSET signal GPIO_IO = gpio_io
END
```

Example MSS File Entry

```
SELECT INSTANCE mygpio
CSET attribute DRIVER = drv_gpio
CSET attribute DRIVER_VER = 1.00.a
END
```

Example C Program

```
#include <gpio.h>
#include <mbio.h>
void main() {
  UINT32 gpio_data;
   /* Set the gpio as output on low 8 bits (LEDs), and input on high 8 bits
(DIP switches) */
  /* Libgen sets up all base addresses in mbio.h The format is <peripheral
instance name (in caps) in mss file>_BASEADDR */
  gpio_write_ctrl(MYGPIO_BASEADDR, 0xff00);
  /* Read from the DIP switches */
  gpio_data = gpio_read_INT32(MYGPIO_BASEADDR);
  /* Write this value to the the LEDs */
  gpio_data >>= 8;
  gpio_write_INT32(MYGPIO_BASEADDR, gpio_data);
}
```

Interrupt Controller Driver

The following driver routines are available for the interrupt controller.

void intc_disable_all_interrupts (UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Disable interrupts for all peripherals. This is done by clearing the Master IRQ Enable bit of the Master Enable Register (MIE). Interrupts can be enabled again by calling intc_start.
Includes	mb_interface.h

void intc_disable_interrupt (UINT32 base_addr, UINT32 periph_int_priority_mask)

Parameters	base_addr: Base address of the interrupt controller periph_int_priority_mask: Peripheral interrupt priority mask
Description	Disable interrupts for peripherals whose interrupt priority is set in the interrupt priority mask. If periph_int_priority_mask has a 1 in bit position i, then the peripheral whose interrupt priority is i+1 has its interrupts disabled. At startup, all peripherals have their interrupts disabled.
Includes	mb_interface.h

void intc_enable_interrupt (UINT32 base_addr, UINT32 periph_int_priority_mask)

Parameters	base_addr: Base address of the interrupt controller periph_int_priority_mask: Peripheral interrupt priority mask
Description	Enable interrupts for peripherals whose interrupt priority is set in the interrupt priority mask. If periph_int_priority_mask has a 1 in bit position i, then the peripheral whose interrupt priority is i+1 has its interrupts enabled. At startup, all peripherals have their interrupts disabled.
Includes	mb_interface.h

void intc_interrupt_handler (void)

Parameters	None
Description	If an interrupt controller is included in the system, then this routine is made the default interrupt handler and is called whenever the MicroBlaze IRQ input is raised. It locates the highest priority interrupting device that has interrupts enabled, and calls its interrupt handler. On return from the handler, the interrupt is acknowledged by writing to the Interrupt Acknowledge Register (IAR). If any lower priority devices have also raised an interrupt their interrupt handlers are also called, and their interrupts acknowledged.
Includes	None. This routine must not be called directly by the user.

void intc_start (UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Start the interrupt controller. This enables hardware interrupts by setting the HardWare Interrupt Enable (HIE) bit of the Master Enable Register (MER). It also sets the Master IRQ Enable (MIE) bit of the Master Enable Register (MER).
Includes	mb_interface.h

JTAG UART Driver

The driver is similar to the UART lite driver routines except that the function names are changed to jtag_uart_<function name>. For e.g., instead of uartlite_inbyte, use jtag_uart_inbyte.

SPI Driver

Serial Bus Interface core allows to connect multiple serial devices and facilitates communication between them. The following functions are available.

INT8 inbyte (void)

Parameters	None
Description	Read a byte from the SPI device. This call will block until a byte is actually available on the SPI Data Receive Register (DRR) or the associated FIFO. This function is added to the library only if a SPI device is configured to be the standard input to the system. Functions such as scanf call inbyte to read a single byte. Libgen automatically configures inbyte with the correct base address.
Includes	This function should not be called directly. Use spi_read_byte to read a character from the SPI device.

void outbyte (INT8 ch)

Parameters	ch: character to be written to the UART Lite
Description	Write a byte to the SPI device. This call will block until the byte is actually written to the SPI Data Transmit Register (DTR) or the associated FIFO, if any. This function is added to the library only if a SPI device is configured to be the standard output to the system. Functions such as printf call outbyte to write a single byte. Libgen automatically configures outbyte with the correct base address.
Includes	This function should not be called directly. Use spi_write_byte to write a character to the SPI device.

void spi_write_control_reg(UINT32 base_addr, INT8 val)

Parameters	base_addr: Base address of the interrupt controller val: The value to be written.
Description	Write the val to the Control Register (CR)
Includes	spi.h mbio.h

UINT8 spi_read_control_reg(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Read the return the contents of the Control Register
Includes	spi.h mbio.h

void spi_enable_device (UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Enable the SPI device to perform data transfer operations
Includes	spi.h mbio.h

void spi_disable_device (UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Disable the SPI Device. It can not perform data transfer now
Includes	spi.h mbio.h

void spi_write_intr_enable_reg(UINT32 base_addr, UINT8 mask)

Parameters	base_addr: Base address of the interrupt controller mask: The value to be written.
Description	Enable chosen interrupts by writing '1' on specific locations in the bit mask to be written.
Includes	spi.h mbio.h

void spi_read_intr_enable_reg(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Read and return the current value of the interrupt enable register to find out which interrupts are currently enabled
Includes	spi.h mbio.h

void spi_write_intr_reg (UINT32 base_addr, UINT8 mask)

Parameters	base_addr: Base address of the interrupt controller mask: The value to be written.
Description	Write a perticular mask to the Interrupt Register, thus generating the corresponding interrupt(s) from software
Includes	spi.h mbio.h

UINT8 spi_read_intr_reg(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Read and return the value of the interrupt register to find out which interrupts are pending.
Includes	spi.h mbio.h

void spi_clear_all_intr (UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Clear all pending interrupts by writing 0 to the interrupt register
Includes	spi.h mbio.h

void spi_enable_all_intr(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Write that value to the interrupt enable register so that all interrupts are enabled
Includes	spi.h mbio.h

void spi_disable_all_intr(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Write that value to the interrupt enable register so that all interrupts are disabled and the SPI device does not respond to any interrupts
Includes	spi.h mbio.h

void spi_reset_fifo(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Reset receive and transmit FIFOs, if any, attached to the SPI device
Includes	spi.h mbio.h

INT8 spi_read_byte(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Read the Data Receive Register (DRR) and return a byte of data from the SPI device. This function blocks if DDR or the attached FIFO is empty.
Includes	spi.h mbio.h

void spi_write_byte(UINT32 base_addr, INT8 data)

Parameters	base_addr: Base address of the interrupt controller data: The data to be sent out by the SPI device.
Description	Write the data to the Data Transmit Register (DTR) for tranmitting it to other device. This function blocks if the attached FIFO is full or if previous data written to DTR is not yet sent.
Includes	spi.h mbio.h

void spi_set_options(UINT32 base_addr, UINT32 options)

Parameters	base_addr: Base address of the interrupt controller options: The value corresponding to the option to be set
Description	Configure the device in one of the following available option: XSP_MASTER_OPTION: configure as a master (default is slave), option value is 1. XSP_CLK_ACTIVE_LOW_OPTION: configure clock to be active low (default is active high), option value is 2 XSP_CLK_PHASE_1_OPTION: use clock phase 1 (default is phase 0), option value is 4 XSP_LOOPBACK_OPTION: enable loopback mode (default is disabled), option value is 8 For a set of options, option value is sum of the corresponding option values
Includes	spi.h mbio.h

UINT32 spi_get_options(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Return the current option settings for the SPI device
Includes	spi.h mbio.h

void spi_set_slave_select(UINT32 base_addr, UINT32 slave_num)

Parameters	base_addr: Base address of the interrupt controller slave_num: The slave to be selected for this device
Description	This function should only be used when device is configured as a master. Select the slave to which the device will communicate. slave_nm varies from 0 to (number_of_slaves - 1)
Includes	spi.h mbio.h

UINT32 spi_get_slave_select(UINT32 base_addr)

Parameters	base_addr: Base address of the interrupt controller
Description	Return the slave currently selected for this master device. Initially no slave is selected.
Includes	spi.h mbio.h

Timebase/ WatchDog Timer Driver

The following driver functions are available for the Timebase WDT.

void timebase_wdt_disable (UINT32 base_addr)

Parameters	base_addr: Base address of the timebase WDT
Description	Disable the watchdog timer. Clear the EWDT1 bit in TCSR0 and the EWDT2 bit in TCSR1.
Includes	timebase_wdt.h mbio.h

void timebase_wdt_enable (UINT32 base_addr)

Parameters	base_addr: Base address of the timebase WDT
Description	Enable the timebase WDT. Write EWDT2 in Control/Status register TCSR1.
Includes	timebase_wdt.h mbio.h

UINT32 timebase_wdt_get_status (UINT32 base_addr)

Parameters	base_addr: Base address of the timebase WDT
Description	Read the Control/Status Register (TCSR0) of the timebase WDT. The following masks may be used to decode the returned value: TIMEBASE_WDT_WRS : Watchdog reset status TIMEBASE_WDT_WDS : Watchdog timer state TIMEBASE_WDT_EWDT1 : Enable Watchdog Timer (Enable 1) TIMEBASE_WDT_EWDT2 : Enable Watchdog Timer (Enable 2)
Includes	timebase_wdt.h mbio.h

UINT32 timebase_wdt_get_timebase (UINT32 base_addr)

Parameters	base_addr: Base address of the timebase WDT
Description	Read the Timebase Register (TBR) to return the free-running counter value of the timebase WDT.
Includes	timebase_wdt.h mbio.h

void timebase_wdt_kick (UINT32 base_addr)

Parameters	base_addr: Base address of the timebase WDT
Description	Kick the WDT. Clear the Watchdog Timer State (WDS) bit of the Control/Status register TCSR0.
Includes	timebase_wdt.h mbio.h

void timebase_wdt_set_status0 (UINT32 base_addr, UINT32 status)

Parameters	base_addr: Base address of the timebase WDT status: Value to be written to control status register
Description	Write to the Control/Status Register (TCSR0) of the timebase WDT. The following masks may be used to set up the input: TIMEBASE_WDT_WRS : Watchdog reset status TIMEBASE_WDT_WDS : Watchdog timer state TIMEBASE_WDT_EWDT1 : Enable Watchdog Timer (Enable 1)
Includes	timebase_wdt.h mbio.h

void timebase_wdt_set_status1 (UINT32 base_addr, UINT32 status)

Parameters	base_addr: Base address of the timebase WDT status: Value to be written to control status register
Description	Write to the Control/Status Register (TCSR1) of the timebase WDT. The following mask may be used to set up the input: TIMEBASE_WDT_EWDT2 : Enable Watchdog Timer (Enable 2)
Includes	timebase_wdt.h mbio.h

Timer/Counter Driver

The following driver functions are available for the timer/counter.

UINT32 get_elapsed_time(UINT32 base_addr, UINT32 timer_number)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1.
Description	Get the number of clock ticks since function start_timer was last called. Used to find the number of clock ticks required to execute a piece of code. This function will not return the correct value if the timer has rolled over, i.e., if the code being timed takes more than 0xFFFFFFFF clock ticks to execute.
Includes	timer.h mbio.h

void start_timer (UINT32 base_addr, UINT32 timer_number)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1.
Description	Start timing a piece of code. This resets the timer, and enables it in compare mode. Use function get_elapsed_time to read the current time, and thus find the number of clock ticks between the two function calls.
Includes	timer.h mbio.h

UINT32 timer_get_capture (UINT32 base_addr, UINT32 timer_number)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1.
Description	Read the capture register. If the value of timer_number is 0, then read TCCR0, else read TCCR1.
Includes	timer.h mbio.h

UINT32 timer_get_csr (UINT32 base_addr, UINT32 timer_number)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1.
Description	Read from the control/status register. If timer_number is 0, read TCSR0, else read TCSR1. The following masks may be used to decode the status register. TIMER_ENABLE_ALL : Enable all timers. TIMER_PWM : Enable Pulse Width Modulation TIMER_INTERRUPT : Timer Interrupt. TIMER_ENABLE : Enable Timer TIMER_ENABLE_INTR : Enable Interrupts TIMER_RESET : Reset Timer TIMER_RELOAD : Auto Reload/Hold Timer TIMER_EXT_CAPTURE : Enable External Capture Trigger TIMER_EXT_COMPARE : Enable External Compare Signal TIMER_DOWN_COUNT : Up/Down Count TIMER_CAPTURE_MODE : Timer Mode
Includes	timer.h mbio.h

UINT32 timer_get_time (UINT32 base_addr, UINT32 timer_number)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1.
Description	Read the time value from the timer/counter register. If the value of timer_number is 0, then read TCR0, else read TCR1.
Includes	timer.h mbio.h

void timer_set_compare (UINT32 base_addr, UINT32 timer_number, UINT32 compare_value)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1. compare_value: Value to be written to the compare register
Description	Write to the compare register. If the value of timer_number is 0, then write to TCCR0, else write to TCCR1.
Includes	timer.h mbio.h

void timer_set_csr (UINT32 base_addr, UINT32 timer_number, UINT32 status_value)

Parameters	base_addr: Base address of the timer/counter timer_number: Identifies which timer is being accessed. May be 0 or 1. status_value: Value to be written to the status register
Description	Write to the control/status register. If timer_number is 0, write to TCSR0, else write to TCSR1. The following masks may be used when setting up the status value. TIMER_ENABLE_ALL : Enable all timers. TIMER_PWM : Enable Pulse Width Modulation TIMER_INTERRUPT : Timer Interrupt. TIMER_ENABLE : Enable Timer TIMER_ENABLE_INTR : Enable Interrupts TIMER_RESET : Reset Timer TIMER_RELOAD : Auto Reload/Hold Timer TIMER_EXT_CAPTURE : Enable External Capture Trigger TIMER_EXT_COMPARE : Enable External Compare Signal TIMER_DOWN_COUNT : Up/Down Count TIMER_CAPTURE_MODE : Timer Mode
Includes	timer.h mbio.h

Example MHS File snippet

```
SELECT SLAVE opb_timer
CSET attribute HW_VER = 1.00.a
CSET attribute INSTANCE = mytimer
CSET attribute C_BASEADDR = 0xFFFF0000
CSET attribute C_HIGHADDR = 0xFFFF00ff
CSET attribute C_AWIDTH   = 32
CSET attribute C_DWIDTH   = 32
CSET signal Interrupt = interrupt, PRIORITY = 1
CSET signal CaptureTrig0 = net_gnd
CSET signal CaptureTrig1 = reset_gpio
```

```
END
```

Example MSS File snippet

```
SELECT INSTANCE mytimer
CSET attribute DRIVER = drv_timer
CSET attribute INT_HANDLER = timer_int_handler, Interrupt
END
```

Example C Program

```
#include <timer.h>
#include <mbio.h>
#include <mb_interface.h>

/* Use two timers in compare mode to count down from different values.
   Print 'a' when timer 0 interrupts, and 'b' when timer 1 interrupts.
*/

#define TIMER_0_INT_VAL 1000000
#define TIMER_1_INT_VAL 2000000

/* Timer interrupt handler */
void
timer_int_handler() {
  unsigned int csr;

  /* Read timer 0 CSR to see if it raised the interrupt */
  csr = timer_get_csr(MYTIMER_BASEADDR, 0);
  if (csr & TIMER_INTERRUPT) {
   print("a");
    /* Clear the interrupt */
    timer_set_csr(MYTIMER_BASEADDR, 0, csr);
  } else {
    /* Read timer 1 CSR to see if it raised the interrupt */
    csr = timer_get_csr(MYTIMER_BASEADDR, 1);
    if (csr & TIMER_INTERRUPT) {
      print("b");
      /* Clear the interrupt */
      timer_set_csr(MYTIMER_BASEADDR, 1, csr);
    } else {
      /* Error */
      print("Error\n");
    }
  }
}

void
main() {
  int i;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

  /* set the number of cycles each timer must count */
  timer_set_compare(MYTIMER_BASEADDR, 0, TIMER_0_INT_VAL);
  timer_set_compare(MYTIMER_BASEADDR, 1, TIMER_1_INT_VAL);

  /* reset the timers, and clear interrupts */
  timer_set_csr(MYTIMER_BASEADDR, 0, TIMER_INTERRUPT | TIMER_RESET );
```

```
  timer_set_csr(MYTIMER_BASEADDR, 1, TIMER_INTERRUPT | TIMER_RESET );

  /* start the timers */
  timer_set_csr(MYTIMER_BASEADDR, 0, TIMER_ENABLE | TIMER_ENABLE_INTR |
TIMER_RELOAD | TIMER_DOWN_COUNT);
  timer_set_csr(MYTIMER_BASEADDR, 1, TIMER_ENABLE | TIMER_ENABLE_INTR |
TIMER_RELOAD | TIMER_DOWN_COUNT);

  /* Wait for interrupts to occur */
  for (i=0; i<0xFFFFFF; i++) {
      ;
  }
}
```

UART Lite Driver

The following driver functions are available for the UART Lite.

INT8 inbyte (void)

Parameters	None
Description	Read a byte from the UART Lite. This call will block until a byte is actually available on the UART Lite Receive FIFO. This function is added to the library only if a UART Lite is configured to be the standard input to the system. Functions such as scanf call inbyte to read a single byte. Libgen automatically configures inbyte with the correct base address.
Includes	This function should not be called directly. Use uartlite_read_byte to read a character from the UART Lite.

void outbyte (INT8 ch)

Parameters	ch: character to be written to the UART Lite
Description	Write a byte to the UART Lite. This call will block until the byte is actually written to the UART Lite Transmit FIFO. This function is added to the library only if a UART Lite is configured to be the standard output to the system. Functions such as printf call outbyte to write a single byte. Libgen automatically configures outbyte with the correct base address.
Includes	This function should not be called directly. Use uartlite_write_byte to write a character to the UART Lite.

void uartlite_disable_intr (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Disable interrupts on the UART Lite.
Includes	uartlite.h mbio.h

INT32 uartlite_empty (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Check if any data is available on the UART Lite Receive FIFO. Return nonzero if the UART Lite Receive FIFO is empty. Return zero if the UART Lite Receive FIFO is not empty, i.e., if there are characters waiting to be read.
Includes	uartlite.h mbio.h

void uartlite_enable_intr (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Enable interrupts on the UART Lite.
Includes	uartlite.h mbio.h

INT32 uartlite_full (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Check if the UART Lite Transmit FIFO is full. Return nonzero if the UART Lite Transmit FIFO is full. Return zero if the UART Lite Transmit FIFO is not full, i.e., characters can continue to be transmitted to the UART Lite.
Includes	uartlite.h mbio.h

UINT32 uartlite_get_status (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Read and return the UART Lite Status register. The status register contains the following bits: UARTLITE_PAR_ERROR : 1 if Parity error occured UARTLITE_FRAME_ERROR : 1 if Frame error occured UARTLITE_OVERRUN_ERROR : 1 if Overrun error occured UARTLITE_INTR_ENABLED : 1 if interrupts are enabled UARTLITE_TX_FIFO_FULL : 1 if the transmit FIFO is full UARTLITE_TX_FIFO_EMPTY : 1 if the transmit FIFO is empty UARTLITE_RX_FIFO_FULL : 1 if the receive FIFO is full UARTLITE_RX_FIFO_VALID_DATA : 1 if the receive FIFO has valid data
Includes	uartlite.h mbio.h

INT8 uartlite_read_byte (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Read a byte from the UART Lite. This call will block until a byte is actually available on the UART Lite Receive FIFO.
Includes	uartlite.h mbio.h

INT32 uartlite_is_intr_enabled (UINT32 base_addr)

Parameters	base_addr: Base address of uartlite peripheral
Description	Check if interrupts are enabled on the UART Lite. Return nonzero if interrupts are enabled on the UART Lite. Return zero if interrupts are not enabled on the UART Lite.
Includes	uartlite.h mbio.h

void uartlite_write_byte (UINT32 base_addr, INT8 ch)

Parameters	base_addr: Base address of uartlite peripheral ch: character to be written to the UART Lite
Description	Write a byte to the UART Lite. This call will block until the byte is actually written to the UART Lite Transmit FIFO.
Includes	uartlite.h mbio.h

void uartlite_set_control (UINT32 base_addr, UINT32 data)

Parameters	base_addr: Base address of uartlite peripheral data: word to be written to the control register
Description	Write a word to the UART Lite control register. The following masks may be used to write to the control register: UARTLITE_INTR_ENABLE : Enable interrupts on the UART Lite UARTLITE_RST_RX_FIFO : Reset/Clear the receive FIFO UARTLITE_RST_TX_FIFO : Reset/Clear the transmit FIFO
Includes	uartlite.h mbio.h

MicroBlaze Interrupt Routines

The following routines are available to support interrupts on the MicroBlaze. These routines should be used to disable or enable interrupts on MicroBlaze. These routines are included by libgen in the same manner as the drivers, using the DRIVER and DRIVER_VER attributes. Please refer to the Interrupt Management documentation for the use of these routines.

void microblaze_disable_interrupts (void)

Parameters	None
Description	Disable interrupts on the MicroBlaze. This clears the interrupt enable bit of the MSR.
Includes	mb_interface.h

void microblaze_enable_interrupts (void)

Parameters	None
Description	Enable interrupts on the MicroBlaze. Interrupts are disabled when the system starts up. Therefore this function must be called if the MicroBlaze is expected to handle interrupts. This sets the interrupt enable bit of the MSR.
Includes	mb_interface.h

Software Specification

Microprocessor Software Specification (MSS) Format

Jan. 8, 2002

Summary

This document describes the Microprocessor Software Specification (MSS) format for the 32-bit soft processor, MicroBlaze.

Overview

The MSS file specifies the software configuration of the platform. The MSS file defines the standard input/output devices, interrupt handler routines, and other related software features. The MSS file is created by the user. Please also refer to the Microprocessor Hardware Specification documentation for more information on the related MHS file.

Microprocessor Software Specification (MSS) Format

An MSS file is supplied by the user as an input to the Library Generator. The MSS file contains directives for customizing the microblaze executable and the software flow.

The MSS file has a dependency on the MHS file. This dependency has to be specified in the MSS file as **SET attribute HW_SPEC_FILE = <file_name.mhs>**. Hence, a hardware platform has to be defined in order to configure the software flow. Refer the Microprocessor Hardware Specification documentation for more information on hardware configuration.

Keywords

The MSS file consists of **SET** statements that assign values to global attributes. It also consists of **CSET** statements that assign values to instance specific attributes. An instance of a peripheral is selected using the **SELECT** statement. Every **CSET** statement between the **SELECT** and the **END** statement refers to the selected instance. The MSS syntax is not case sensitive. However, attribute and instance names are case sensitive.

Format

The format for assigning global attributes:

```
SET ATTRIBUTE name = value
```

The format for selecting a peripheral instance:

```
SELECT INSTANCE instance_name
```

The format for selecting a file system:

```
SELECT FILESYS filesys_name
```

The format for instance specific assignment statements:

```
CSET ATTRIBUTE name = value
```

The format for ending a peripheral instance definition:

```
END
```

Comments can be specified anywhere in the file. A # character denotes the beginning of a comment and all characters after the # till the end of the line are ignored. White spaces are also ignored.

MSS example

An example MSS file is given below:

```
SET attribute HW_SPEC_FILE = system.mhs
```

```
SET attribute BOOT_PERIPHERAL = my_jtag
SET attribute DEBUG_PERIPHERAL = my_jtag
SET attribute XMDSTUB = code/xmdstub.out
SET attribute BOOTSTRAP = code/bootstub.out
SET attribute EXECUTABLE = code/hello_world.out
SET attribute STDIN = my_uartlite
SET attribute STDOUT = my_uartlite

SELECT INSTANCE my_microblaze
CSET attribute DRIVER = drv_microblaze
CSET attribute DRIVER_VER = 1.00.a
END

SELECT INSTANCE my_intc
CSET attribute DRIVER = drv_intc
CSET attribute DRIVER_VER = 1.00.a
END

SELECT INSTANCE my_uartlite
CSET attribute DRIVER_VER = 1.00.a
CSET attribute DRIVER = uartlite
CSET attribute LIBRARY = XilFile
CSET attribute INT_HANDLER = my_uartlite_hndl, Interrupt
END

SELECT INSTANCE my_timebase_wdt
CSET attribute DRIVER_VER = 1.00.a
CSET attribute DRIVER = drv_timebase_wdt
CSET attribute INT_HANDLER=my_timebase_hndl, Timebase_Interrupt
CSET attribute INT_HANDLER=my_timebase_hndl, WDT_Interrupt
END

SELECT FILESYS XilMfs
CSET attribute MOUNT = /home/mine
CSET attribute LIBRARY = XilFile
END
```

Global Options

These options are specified with a SET keyword. These options define the software specification and are not specific to a peripheral instance.

HW_SPEC_FILE Option

This option points to the MHS file. The path can be a relative path from the <USER_PROJECT> directory or can be an absolute path. This option is mandatory.

Format

```
SET attribute HW_SPEC_FILE = system.mhs
```

BOOTSTRAP Option

The bootstrap image is set using the BOOTSTRAP option.

Format

```
SET attribute BOOTSTRAP = code/bootstub.out
```

Library Generator creates this executable in the <USER_PROJECT>/code directory. Please see the Library Generator document for more information.

The path to the file is a relative path from the <USER_PROJECT> directory.

BOOT_PERIPHERAL Option

Identify boot peripheral with the BOOT_PERIPHERAL option.

Format

```
SET attribute BOOT_PERIPHERAL = <instance_name>
```

STDIN Option

Identify standard input device with the STDIN option.

Format

```
SET attribute STDIN = <instance_name>
```

STDOUT Option

Identify standard output device with the STDOUT option.

Format

```
SET attribute STDOUT = <instance_name>
```

EXECUTABLE Option

The executable image is set using the EXECUTABLE option.

Format

```
SET attribute EXECUTABLE = code/a.out
```

Currently, the executable programming information is used only for initializing the LMB memory.

Hardware designers have the option of defining the system before an executable file is completed. In the MSS file, you can simply leave the value to EXECUTABLE blank.

Format

```
SET attribute EXECUTABLE = “”
```

XMDSTUB Option

The on-board debug image is set using the XMDSTUB option.

Format

```
SET attribute XMDSTUB = code/xmdstub.out
```

Library Generator creates this executable in the <USER_PROJECT>/code directory.

Please see the Library Generator document for more information.

The path to the file is a relative path from the <USER_PROJECT> directory.

DEBUG_PERIPHERAL Option

The peripheral that is used to handle the xmdstub should be specified in the DEBUG_PERIPHERAL option:

```
SET attribute DEBUG_PERIPHERAL = <instance_name>
```

Instance Specific Options

Peripheral instances defined in the MSS are allowed the following list of options:

Table 1: **MSS Peripheral Options**

Option	Values	Default	Definition
DRIVER		Must be specified	Driver directory name
DRIVER_VER	1.00.a	No Version	Driver version
INT_HANDLER	name of interrupt handler	default_int_handler	Interrupt handler function
LIBRARY	library name	none	The peripheral is accessed as a block peripheral through Xilinx library calls

DRIVER Option

This option is needed for peripherals that have drivers associated with them.

Format

```
CSET attribute DRIVER = drv_uartlite
```

Library Generator copies the driver directory specified to <USER_PROJECT>/libsrc directory and creates the driver using the makefile specified. Please see the Library Generator document for more information.

DRIVER_VER Option

The driver version is set using the DRIVER_VER option.

Format

```
CSET attribute DRIVER_VER = 1.00.a
```

The version is specified as a literal of the form 1.00.a.

INT_HANDLER Option

This option defines the interrupt handler software routine for an interrupt signal of the peripheral.

Format

```
CSET attribute INT_HANDLER = my_int_handl, Interrupt
```

The interrupt signal which is handled by the interrupt handler is specified after the attribute as shown above. This signal should match the signal name specified in the MHS file for that peripheral instance.

LIBRARY Option

This option specifies that the peripheral is also accessed through high level routines in Xilinx libraries such as LibXil File and LibXil Mfs.

Format

```
CSET attribute LIBRARY = XilFile
```

This attribute should be used only for UARTLITE, JTAG_UART and GPIO peripherals. This option enables customization of Xilinx Libraries for block access such as device **open, read, write, seek, etc.**

File System Specific Options

A supported file system can be selected by using the SELECT FILESYS command in the MSS.:

Table 2: **MSS FileSys Options**

Option	Values	Default	Definition
MOUNT	string	None	Specifies the mount name
LIBRARY	library name	none	The file system is accessed as through a Xilinx library.

MOUNT Option

This option defines the mount name for the file system instance. Every file system instance should have a unique mount name. The name is a directory name.

Format

```
CSET attribute MOUNT = /home/
```

The above command specifies that the mount name for the file system instance is /home/.

LIBRARY Option

This option specifies that the file system is accessed through high level routines in Xilinx libraries such as LibXil File.

Format

```
CSET attribute LIBRARY = XilFile
```

This option enables customization of Xilinx Libraries for block access through generic file **open, read, write, seek, etc.,** irrespective of the file system implementation.

Jan. 10, 2002

MicroBlaze Address Management

Summary

This document describes the MicroBlaze program address management techniques. For advanced address space management, a discussion on linker scripts is also included in this document.

Programs and Memory

MicroBlaze users can write either C or Assembly programs, and use the MicroBlaze Development Kit to transform their source code into bit patterns stored in the physical memory of a MicroBlaze System. User programs typically access local/on-chip memory, external memory and memory mapped peripherals. Memory requirements for user programs are specified in terms of how much memory is required for storing the instructions, and how much memory is required for storing the data associated with the program.

MicroBlaze address space is divided between the system address space and the user address space. In certain examples, users would need advanced address space management, which can be done with the help of linker script, described in this document.

Current Address Space Restrictions

Memory and Peripherals Overview

MicroBlaze uses 32-bit addresses, and as a result it can address memory in the range zero through 0xFFFFFFFF. MicroBlaze can access memory either through its Local Memory Bus (LMB) port or through the On-chip Peripheral Bus (OPB). The LMB is designed to be a fast access, on-chip block RAM (BRAM) memories only bus. The OPB represents a general purpose bus interface to on-chip or off-chip memories as well as other non-memory peripherals.

LMB vs OPB Address Space

Currently, the ninth address bit is used to distinguish between LMB and OPB addresses. OPB addresses can be used for on-chip memory, external memory, on-chip memory mapped peripherals or off-chip memory mapped peripherals.

Notation: MicroBlaze is a 32-bit big endian processor

Bit0 Bit1 Bit2 ... Bit31

LMB addresses look like this:

xxxx xxxx **0**xxx xxxx xxxx xxxx xxxx xxxx

OPB addresses look like this:

xxxx xxxx **1**xxx xxxx xxxx xxxx xxxx xxxx

In hex notation, OPB addresses are **`0xXX800000-0xXXffffff`**, representing 256 address ranges with 23 bits of address space per range (2^23 addresses per range).

BRAM Size Limits

The amount of BRAM memory that can be assigned to the LMB address space or to each instance of an OPB mapped BRAM peripheral is limited. The largest supported BRAM memory size for Virtex/VirtexE is 16 kilobytes and for Virtex2 it is 64 kilobytes. It is important to understand that these limits apply to each separately decoded on-chip memory region only. The total amount of on-chip memory available to a MicroBlaze system may exceed these limits. The total amount of memory available in form of BRAMs is also FPGA device specific. Smaller

devices of a given device family provide less BRAM than larger devices in the same device family.

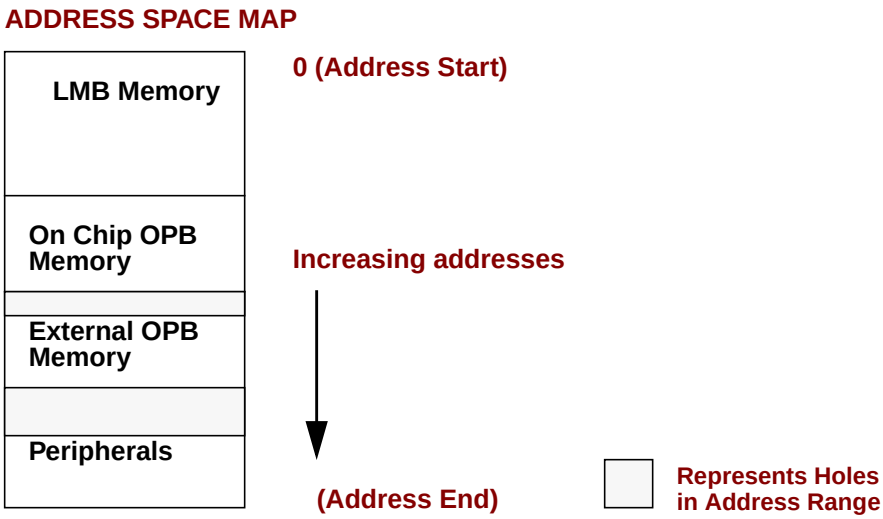


Figure 1: **Sample Address Map**

Special Addresses

Every MicroBlaze system must have user writable memory present in addresses 0x00000000 through 0x00000018. These memory locations contain the addresses MicroBlaze jumps to after a reset, interrupt, or exception event occurs. It follows that the LMB address space must start at memory location zero. Please refer to the *MicroBlaze Application Binary Interface* (ABI) documentation for further details.

OPB Address Range Details

Within the OPB address space, the user can arbitrarily assign address space to on/off-chip memory peripherals and to on/off-chip non-memory peripherals. The OPB address space may contain holes representing regions that are not associated with any OPB peripheral. Special linker scripts and directives may be required to control the assignment of object file sections to address space regions.

Address Map

Figure 1 shows a possible address map for a MicroBlaze System. The actual address map is defined in the MicroBlaze Hardware Specification (MHS) file. It contains an address map specifying the addresses of LMB memory, OPB memory, External memory and peripherals.

The address range grows from 0. At the lowest range is the LMB memory. This is followed by the OPB memory, External Memory and the Peripherals. Some addresses in this address space have predefined meaning. The processor jumps to address 0x0 on reset, to address 0x8 on exception, and to address 0x10 on interrupt.

Memory Speeds and Latencies

MicroBlaze requires 2 clock cycles to access on-chip Block RAM connected to the LMB for *write* and 2 clock cycles for *read*. On chip memory connected to the OPB bus requires 3 cycles for *write* and 4 cycles for *read*. External memory access is further limited by off-chip memory access delays for read access, resulting in 5-7 clock cycles for *read*. Furthermore, memory accesses over the OPB bus may incur further latencies due to bus arbitration overheads. As a

result, instructions or data that need to be accessed quickly should be stored in LMB memory when possible.

For more information on memory access times, see the *MicroBlaze Hardware Reference* documentation.

System Address Space

MicroBlaze programs can be executed in different scenarios. Each scenario needs a different set of system address space. The system address space is occupied by the xmdstub or the bootstub, when debug or boot support is required. System address space is also need by the C-runtime routines.

System with only an executable [No debug, No Bootstrap]

The scenario is depicted in Figure 2(a). The C-runtime file crt0.o is linked with the user program. The system file, crt0.o starts at address location 0x0, immediately followed by user's program.

System with debugging support

With systems requiring debug support, **xmdstub** has to be downloaded at address location 0x0. The C-runtime file crt1.o is bundled with the user program and is place at a default location. This scenario is shown in Figure 2(b).

System with bootstrap support

The user can also bootstrap their program by using the bootstub. This bootstub occupies the system address space starting at address location 0x0. In addition to this system space, every user program is pre-pended with another C-runtime routine crt2.o or crt3.o depending on the compilation switch used. This scenario is shown in Figure 2(c).

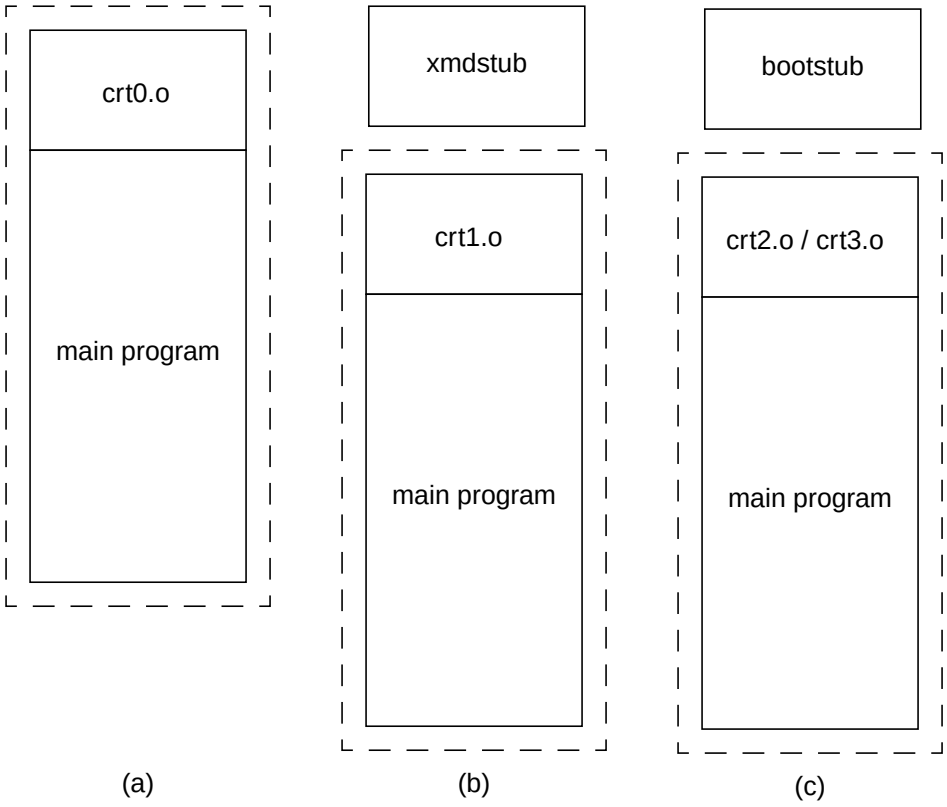


Figure 2: **Execution Scenarios**

Default User Address Space

The default usage of the compiler **mb-gcc** will place the users program immediately after the system address space. The user does not have to give any additional options in order to make space for the system files. The default start address for user programs is described in Table 1

Table 1: **Start address for different compilation switches**

Compile Option	Start Address
-xl-mode-executable	0x0
-xl-mode-xmdstub	0x400
-xl-mode-bootstrap	0x100
-xl-mode-bootstrap-reset	0x100

If the user needs to start the program at a location other than the default start address or if non-contiguous address space is required, advanced address space management is required.

Advanced User Address Space

Different Base Address, Contiguous User Address Space

The user program can run from any memory [i.e LMB memory or OPB memory]. By default, the compiler will place the user program at location defined in Table 1. To execute program from OPB memory or any address location other than default, users would have to provide the compiler **mb-gcc** with additional option.

The option required is

`-Wl,-defsym -Wl,_TEXT_START_ADDR=<start_address>`

where **`<start_address>`** is the new base address required for the user program.

Different Base Address, Non-contiguous User Address Space

The users can place different components of their program on different memories. For example, on MicroBlaze systems with holes non-contiguous LMB and OPB memories, users can keep their code on LMB memory and the data on OPB memory. The users can also create systems which have contiguous address space for LMB and OPB memory, but having holes in the OPB address space.

All such user program need creation of a non-contiguous executables. To facilitate creation of non-contiguous executable, linker scripts have to be modified. The default linker script provided with the MicroBlaze Distribution Kit will place all user code and data in one contiguous address space.

Linker scripts are defined in later sections in this document.

For more details on linker options see the *MicroBlaze GNU Compiler* documentation.

Object-file Sections

The sections of an executable file are created by concatenating the corresponding sections in an object (.o) file. The various sections in the object file are given in Figure 3.:

.text

This section contains executable code. This section has the x (executable), r (read-only) and i (initialized) flags.

.rodata

This section contains read-only data of size more than 8 bytes (default). The size of the data put into this section can be changed with an mb-gcc -G option. All data in this section is accessed using absolute addresses. This section has the r (read-only) and the i (initialized) flags. For more details refer to the *MicroBlaze ABI* documentation.

.sdata2

This section contains small read-only data (size less than 8 bytes). The size of the data going into this section can be changed with an mb-gcc -G option. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all data in the .sdata2 section can be accessed using a single instruction (A preceding imm instruction will never be necessary). This section has the r (read-only) and the i (initialized) flags. For more details refer to the *MicroBlaze ABI* documentation.

.data

This section contains read-write data of size more than 8 bytes (default). The size of the data going into this section can be changed with an mb-gcc -G option. All data in this section is accesses using absolute addresses. This section has the w (read-write) and the i (initialized) flags.

Sectional Layout of an Object or an Executable File

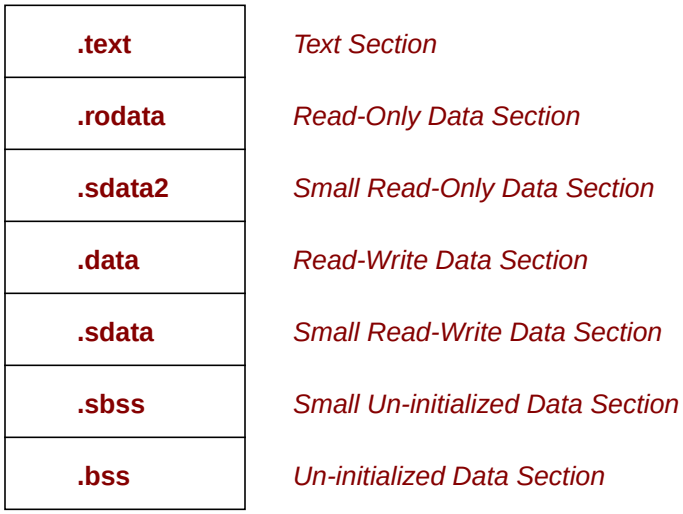


Figure 3: **Sectional layout of an object or executable file**

.sdata

This section contains small read-write data of size less than 8 bytes (default). The size of the data going into this section can be changed with an mb-gcc -G option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all data in the .sdata section using a single instruction. (A preceding imm instruction will never be necessary). This section has the w (read-write) and the i (initialized) flags.

.sbss

This section contains small un-initialized data of size less than 8 bytes (default). The size of the data going into this section can be changed with an mb-gcc -G option. This section has the w (read-write) flag.

.bss

This section contains un-initialized data of size more than 8 bytes (default). The size of the data going into this section can be changed with an mb-gcc -G option. All data in this section is accessed using absolute addresses. The stack and the heap are also allocated to this section. This section has the w (read-write) flag.

The linker script describes the mapping between all the sections in all the input object files, and the output executable file.

If your address map specifies that the LMB, OPB and External Memory occupy contiguous areas of memory, you can use the default (built-in) linker script to generate your executable. This is done by invoking mb-gcc as follows:

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

Note that using the built-in linker script implies that you have no control over which parts of your program are mapped to the different kinds of memory.

Minimal Linker Script

If your LMB, OPB and External Memory do not occupy contiguous areas of memory, you can use a minimal linker script to define your memory layout. Here is a minimal linker script that describes the memory regions only, and uses the default (built-in) linker script for everything else.

```
/*
* Define the memory layout, specifying the start address and size of the
* different memory regions. The ILMB will contain only executable code (x),
* the DLMB will contain only initialized data (i), and the DOPB will contain
* all other writable data (w). Note that all sections of all your input
* object files must map into one of these memory regions. Other memory types
* that may be specified are "r" for read-only data.
*/
MEMORY
  {
    ILMB (x) : ORIGIN = 0x0, LENGTH = 0x1000
    DLMB (i) : ORIGIN = 0x2000, LENGTH = 0x1000
    DOPB (w) : ORIGIN = 0x8000, LENGTH = 0x30000
  }
```

This script specifies that the ILMB memory will contain all object file sections that have the x flag, the DLMB will contain all object file sections that have the i flag and the DOPB will contain all object file sections that have the w flag. An object file section that has both the x and the i flag (.e.g., the .text section) will be loaded into ILMB memory because this is specified first in the linker script. Refer to the Object-file Sections section of this document for more information on object file sections, and the flags that are set in each.

Your source files can now be compiled by specifying the minimal linker script as though it were a regular file, e.g.,

```
mb-gcc <minimal linker script> file1.c file2.c
```

Remember to specify the minimal linker script as the first source file.

If you want more control over the layout of your memory, e.g., if you want to split up your .text section between ILMB and IOPB, or if you want your stack and heap in DLMB and the rest of the .bss section in DOPB, you will need to write a full-fledged linker script.

Linker Script

You will need to use a linker script if you want to control how your program is targeted to LMB, OPB or External Memory. Remember that LMB memory is faster than both OPB and External Memory, and you may want to keep that portion of your code that is accessed the most frequently in LMB memory, and that which is accessed the least frequently in External Memory.

You will need to provide a linker script to mb-gcc using the following command:

```
mb-gcc -Wl,-T -Wl,<linker script> file1.c file2.c -save-temps
```

This tells mb-gcc to use your linker script only, and to not use the default (built-in) linker script.

The Linker Script defines the layout and the start address of each of the sections for the output executable file. Here is a sample linker script.

```
/*
* Define the memory layout, specifying the start address and size of the
* different memory regions.
*/
MEMORY
  {
    LMB : ORIGIN = 0x0, LENGTH = 0x1000
    OPB : ORIGIN = 0x8000, LENGTH = 0x5000
```

```
  }


/*
* Specify the default entry point to the program
*/
ENTRY(_start)

/*
* Define the sections, and where they are mapped in memory
*/
SECTIONS
{

/*
* Specify that the .text section from all input object files will be placed
* in LMB memory into the output file section .text Note that mb-gdb expects
* the executable to have a section called .text
*/
.text : {
/* Uncomment the following line to add specific files in the opb_text */
/* region */
   /*    *(EXCLUDE_FILE(file1.o).text) */
   /* Comment out the following line to have multiple text sections */

   *(.text)
  } >LMB


  /* Define space for the stack and heap */
  /* Note that variables _heap must be set to the beginning of this area */
  /* and _stack set to the end of this area */

  . = ALIGN(4);
  _heap = .;
  .bss : {
    _STACK_SIZE = 0x400;
    . += _STACK_SIZE;
    . = ALIGN(4);
  } >LMB
  _stack = .;

  /*                      */
  /* Start of OPB memory */
  /*                      */

  .opb_text : {
    /* Uncomment the following line to add an executable section into */
    /* opb memory */
    /*    file1.o(.text) */
  } >OPB

  . = ALIGN(4);
  .rodata : {
    *(.rodata)
  } >OPB

 /* Alignments by 8 to ensure that _SDA2_BASE_ on a word boundary */
 . = ALIGN(8);
 _ssrw = .;
 .sdata2 : {
   *(.sdata2)
```

```
  } >OPB
  . = ALIGN(8);
  _essrw = .;
  _ssrw_size = _essrw - _ssrw;
  _SDA2_BASE_ = _ssrw + (_ssrw_size / 2 );

  . = ALIGN(4);
  .data : {
    *(.data)
  } >OPB

  /* Alignments by 8 to ensure that _SDA_BASE_ on a word boundary */
  /* Note that .sdata and .sbss must be contiguous */

  . = ALIGN(8);
  _ssro = .;
  .sdata : {
    *(.sdata)
  } >OPB
  . = ALIGN(4);
  .sbss : {
    *(.sbss)
  } >OPB
  . = ALIGN(8);
  _essro = .;
  _ssro_size = _essro - _ssro;
  _SDA_BASE_ = _ssro + (_ssro_size / 2 );

  . = ALIGN(4);
  .opb_bss : {
    *(.bss) *(COMMON)
  } > OPB
  . = ALIGN(4);
  _end = .;
}
```

Note that if you choose to write a linker script, you **must** do the following to ensure that your program will work correctly:

- Allocate space in the .bss section for stack and heap. Set the _heap variable to the beginning of this area, and the _stack variable to the end of this area. See the .bss section in the script above for an example.
- Ensure that the _SDA2_BASE_ variable points to the center of the .sdata2 area, and that _SDA2_BASE_ is aligned on a word boundary.
- Ensure that the .sdata and the .sbss sections are contiguous, that the _SDA_BASE_ variable points to the center of this section, and that _SDA_BASE_ is aligned on a word boundary.
- If you are not using the rom monitor, ensure that crt0 is always loaded into memory address zero. mb-gcc ensures that this is the first file specified to the loader, but the loader script needs to ensure that it gets loaded at address zero. See the .text section in the example above to see how this is done.

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

XILINX®

MicroBlaze Application Binary Interface

Jan. 9 , 2002

Summary

This document describes MicroBlaze Application Binary Interface (ABI), which is important for developing software in assembly language for the soft processor. The MicroBlaze Gnu compiler follows the conventions described in this document. Hence any code written by assembly programmers should also follow the same conventions to be compatible with the compiler generated code. Interrupt and Exception handling is also explained briefly in the document.

Data Types

The data types used by MicroBlaze assembly programs are shown in Table 1. Data types such as data8, data16, and data32 are used in place of the usual byte, halfword, and word.

Table 1: **Data types in MicroBlaze assembly programs**

MicroBlaze data types (for assembly programs)	Corresponding ANSI C data types	Size (bytes)
data8	char	1
data16	short	2
data32	int	4
data32	long int	4
data32	enum	4
data16 / data32	pointer[1]	2 / 4

1. Pointers to small data areas, which can be accessed by global pointers are data16.

Register Usage Conventions

The register usage convention for MicroBlaze is given in Table 2

Table 2: **Register usage conventions**

Register	Type	Purpose
R0	Dedicated	Value 0
R1	Dedicated	Stack Pointer
R2	Dedicated	Read-only small data area anchor
R3-R4	Volatile	Return Values
R5-R10	Volatile	Passing parameters / Temporaries
R11-R12	Volatile	Temporaries
R13	Dedicated	Read-write small data area anchor
R14	Dedicated	Return address for Interrupt
R15	Dedicated	Return address for Sub-routine
R16	Dedicated	Return address for Trap (Debugger)
R17	Dedicated	Return Address for Exceptions
R18	Dedicated	Reserved for Assembler

Table 2: **Register usage conventions**

Register	Type	Purpose
R19-R31	Non-Volatile	Must be saved across function calls
RPC	Special	Program counter
RMSR	Special	Machine Status Register

The architecture for MicroBlaze defines 32 general purpose registers (GPRs). These registers are classified as volatile, non-volatile and dedicated.

- The volatile registers are used as temporaries and do not retain values across the function calls. Registers R3 through R12 are volatile, of which R3 and R4 are used for returning values to the caller function, if any. Registers R5 through R10 are used for passing parameters between sub-routines.
- Registers R19 through R31 retain their contents across function calls and are hence termed as non-volatile registers. The callee function is expected to save those non-volatile registers, which are being used. These are typically saved to the stack during the prologue and then reloaded during the epilogue.
- Certain registers are used as dedicated registers and programmers are not expected to use them for any other purpose.
  - Registers R14 through R17 are used for storing return address from interrupts, sub-routines, traps and exceptions in that order. Sub-routines are called using the branch and link instruction, which saves the current Program Counter (PC) onto register R15.
  - Small data area pointers are used for accessing certain memory locations with 16 bit immediate value. These areas are discussed in the memory model section of this document. The read only small data area (SDA) anchor R2 (Read-Only) is used to access the constants such as literals. The other SDA anchor R13 (Read-Write) is used for accessing the values in the small data read-write section.
  - Register R1 is used to store the value of the stack pointer and is updated on entry and exit from functions.
  - Register R18 is used as a temporary register for assembler operations.
- MicroBlaze has certain special registers such as program counter (rpc) and machine status register (rmsr). These registers are not mapped directly to the register file and hence the usage of these registers is different from the general purpose registers. The value from rmsr and rpc can be transferred to general purpose registers by using **`mts`** and **`mfs`** instructions (For more details refer to *Instruction Set Architecture* document).

Stack Convention

The stack conventions used by MicroBlaze are detailed in Figure 1

The shaded area in Figure 1 denotes a part of the caller function's stack frame, while the un-shaded area indicates the callee function's frame. The ABI conventions of the stack frame define the protocol for passing parameters, preserving non-volatile register values and allocating space for the local variables in a function. Functions which contain calls to other sub-routines are called as non-leaf functions, These non-leaf functions have to create a new stack frame area for its own use. When the program starts executing, the stack pointer will have the maximum value. As functions are called, the stack pointer is decremented by the number of words required by every function for its stack frame. The stack pointer of a caller function will always have a higher value as compared to the callee function.

Figure 1: **Stack Convention**

High Address	
	Function Parameters for current Procedure (Arg n ..Arg1) (Optional: Maximum number of arguments required for any called procedure from the current procedure. n = 6)
Old Stack Pointer	Link Register (R15)
	Callee Saved Register (R31....R19) (Optional: Only those registers which are used by the current procedure are saved)
	Local Variables for Current Procedure (Optional: Present only if Locals defined in the procedure)
	Functional Parameters (Arg n .. Arg 1) (Optional: Maximum number of arguments required for any called procedure from the current release)
New Stack Pointer	Link Register
Low Address	

Consider an example where Func1 calls Func2, which in turn calls Func3. The stack representation at different instances is depicted in Figure 2. After the call from Func 1 to Func 2, the value of stack pointer (SP) is decremented. This value of SP is again decremented to accommodate the stack frame for Func3. On return from Func 3 the value of stack pointer is increased to its original value in the function, Func 2.

Details of how stack is maintained are shown in Figure 2.

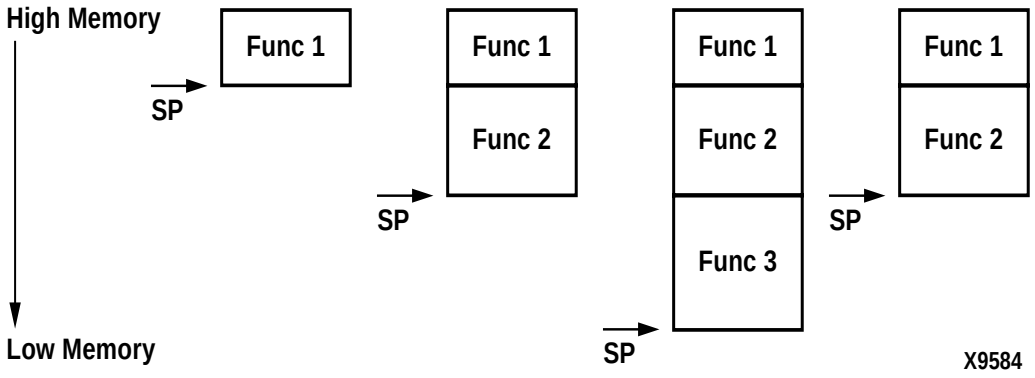


Figure 2: **Stack Frame**

Memory Model

The memory model for MicroBlaze classifies the data into four different parts:

Small data area

Global initialized variables which are small in size are stored in this area. The threshold for deciding the size of the variable to be stored in the small data area is set to 8 bytes in the MicroBlaze C compiler (mb-gcc), but this can be changed by giving a command line option to the compiler. Details about this option are discussed in the MicroBlaze Compiler Tools document. 64K bytes of memory is allocated for the each small data areas. Small data area is accessed using the read-write small data area anchor (R13) and a 16-bit offset. Allocating small variables to this area reduces the requirement of adding *Imm* instructions to the code for accessing global variables. Any variable in the small data area can also be accessed using an absolute address.

Data area

Comparatively large initialized variables are allocated to the data area, which can either be accessed using the read-write SDA anchor R13 or using the absolute address, depending on the command line option given to the compiler.

Common un-initialized area

Un-initialized global variables are allocated to the comm area and can be accessed either using the absolute address or using the read-write small data area anchorR13.

Literals or constants

Constants are placed into the read-only small data area and are accessed using the read-only small data area anchor R2.

The compiler generates appropriate global pointers to act as base pointers. The actual values of the SDA anchors will be decided by the linker, in the final linking stages. For more information on the various sections of the memory please refer to the *MicroBlaze Program Layout* document. The compiler generates appropriate sections, depending on the command line options. Please refer to the *MicroBlaze GNU Compiler Tools* document for more information about these options.

Interrupt and Exception Handling

MicroBlaze assumes certain address locations for handling interrupts and exceptions as indicated in Table 3. When the device is powered ON or on a reset, execution starts at 0x0. If an exception occurs, MicroBlaze jumps to address location 0x8, while in case of an interrupt, the control is passed to address location 0x10. At these locations, code is written to jump to the appropriate handlers.

Table 3: **Interrupt and Exception Handling**

On	Hardware jumps to	Software Labels
Start / Reset	0x0	_start
Exception	0x8	_exception_handler
Interrupt	0x10	_interrupt_handler

The code expected at these locations is as shown in Figure 3. In case of programs compiled without the **-xl-mode-xmdstub** compiler option, the **crt0.o** initialization file is passed by the mb-gcc compiler to the **mb-ld** linker for linking. This file sets the appropriate addresses of the exception handlers.

In case of programs compiled with the **-xl-mode-xmdstub** compiler option, the **crt1.o** initialization file is linked to the output program. This program has to be run with the xmdstub already loaded in the memory at address location 0x0. Hence at run-time, the initialization code

in crt1.o writes the appropriate instructions to location 0x8 through 0x14 depending on the address of the exception and interrupt handlers.

Figure 3: **Code for passing control to exception and interrupt handlers**

```
0x00:   bri     _start1
0x04:   nop
0x08:   imm     <high bits of address(exception handler)>
0x0c:   bri     _exception_handler
0x10:   imm     <high bits of address(interrupt handler)>
0x14:   bri     _interrupt_handler
```

MicroBlaze allows exception and interrupt handler routines to be located at any address location addressable using 32 bits. The exception handler code starts with the label **_exception_handler**, while the interrupt handler code starts with the label **_interrupt_handler**.

In the current MicroBlaze system, there are dummy routines for interrupt or exception handling, which can be changed by the user. In order to override these routines and link user's interrupt and exception handlers, the user has to define the interrupt handler code with an attribute **interrupt_handler**. For more details about the use and syntax of interrupt handler attribute, please refer to the MicroBlaze GNU Compiler Tools document.

MicroBlaze Interrupt Management

Jan. 10, 2002

Summary

This document describes interrupt management for the MicroBlaze soft processor.

Overview

Interrupt Management involves writing interrupt handler routines for peripherals and setting up the MHS and MSS files appropriately. MicroBlaze is capable of handling upto 32 interrupting devices. An interrupt controller peripheral is required for handling more than one interrupt signal. The mechanism of interrupt management is different if an interrupt controller is present than when it is not. This document describes both these management procedures.

Interrupt Handlers

Users are expected to write their own interrupt handlers (or Interrupt Service Routines) for any peripherals that raise interrupts. These routines can be written in C just like any other function. The interrupt handler function can have any name with the signature **void *func* (void)**.

Interrupt handler routines have to be tagged with *int_handler* attribute or *save_volatile* attributes so that mb-gcc can identify these routines as handler routines. Refer to the Interrupt Handlers section in the GNU Compiler Tools documentation for more information on these attributes.

Libgen tags these routines automatically when the recommended interrupt management procedures as described below are followed. Please note that library functions, like **`printf`**, cannot be called from within the interrupt handlers. If any user defined function is required to be called from the interrupt handler functions, this function must be tagged with the *save_volatile* attribute.

The Interrupt Controller Peripheral

An interrupt controller peripheral should be used for handling multiple interrupts. In this case, the user is resposible for writing interrupt handlers for the peripheral interrupt signals only. The interrupt handler for the interrupt controller peripheral is automatically generated by libgen. This handler ensures that interrupts from the peripherals are handled by individual interrupt handlers in the order of their priority. Figure 1 shows peripheral interrupt signals with priorities 1 through 4 connected to the interrupt controller input.

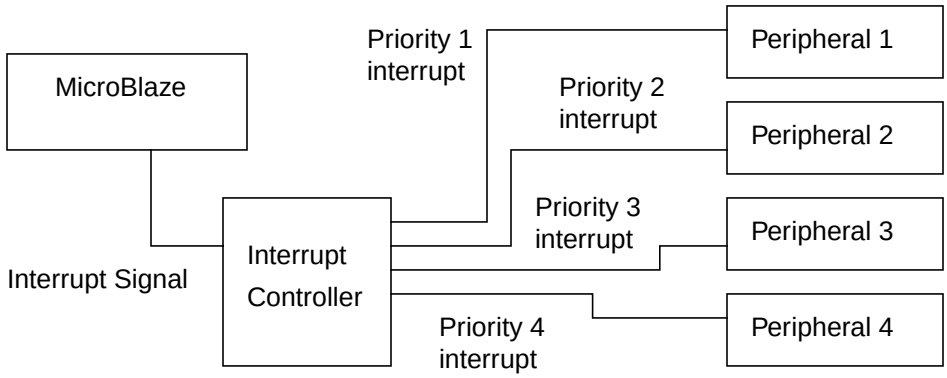


Figure 1: **Interrupt Controller and Peripherals**

The interrupt signal output of the controller is connected to the interrupt input of MicroBlaze. In the MSS file, each peripheral interrupt signal must be associated with interrupt handler routines (also called Interrupt Service Routines). Libgen automatically creates a vector table with the peripheral interrupt handler routines listed in the order of priority. When any peripheral raises an interrupt the default handler for the interrupt controller is called. This handler then queries the interrupt controller to find out which peripheral raised the interrupt and then calls the peripheral specific interrupt handler. For a system where the interrupt controller is not present and only one interrupt signal is connected, the peripheral's interrupt handler (written by the user) gets called when an interrupt occurs.

MicroBlaze Enable Interrupts

The functions *microblaze_enable_interrupts* and *microblaze_disable_interrupts* are used to enable and disable interrupts on MicroBlaze. These functions are described in the Drivers document since they are included by libgen in the same manner that peripheral drivers are included. The include file **mb_interface.h** present in the **MICROBLAZE_PROJECT/include** directory contains the definition of these functions.

System without Interrupt Controller

Single Interrupt Signal

An interrupt controller is not required if there is a single interrupting peripheral and its interrupt signal is level sensitive. Note that a single peripheral may raise multiple interrupts. In this case, an interrupt controller is required.

Procedure

To set up a system without an interrupt controller that handles only one level sensitive interrupt signal, the following steps must be taken:

1. The MHS and MSS file must be set up as follows:
   - The interrupt signal of the peripheral must be connected to the interrupt input of the MicroBlaze in the MHS file.
   - The peripheral must be given an instance name using the INSTANCE keyword in the MHS file. Libgen creates a definition in **mbio.h** (**`MICROBLAZE_PROJECT/include`**) for <INSTANCE_NAME>_BASEADDR mapped to the base address of this peripheral.
2. The interrupt handler routine that handles the signal should be written. The base address of the peripheral instance is accessed as <INSTANCE_NAME>_BASEADDR.
3. The handler function is then designated to be an interrupt handler for the signal using the INT_HANDLER keyword in the MSS file (Refer the Microprocessor Software Specification documentation). The peripheral instance is first selected in the MSS file, and then the INT_HANDLER attribute is given the function name.
4. Libgen and mb-gcc are executed. This has the following implications:
   - the function is marked as an interrupt handler using the mb-gcc *interrupt_handler* attribute. All volatile registers used by this function are saved. Also, this function will return using the *rtid* instruction, rather than the normal *rtsd* instruction. Furthermore, this function will also be given the name *_interrupt_handler* by mb-gcc. By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding rtid instruction is executed.
   - the startup code (crt0, crt1, crt2 or crt3) places the address of _interrupt_handler as the target address that MicroBlaze jumps to when an interrupt occurs. Therefore control will go to the interrupt handler when an interrupt occurs.

Example MHS File

```
SELECT SLAVE opb_uartlite
CSET attribute INSTANCE = myuart
CSET attribute HW_VER = 1.00.a
```

```
CSET attribute C_BASEADDR   = 0xFFFF8000
CSET attribute C_HIGHADDR   = 0xFFFF80ff
CSET attribute C_DATA_BITS  = 8
CSET attribute C_CLK_FREQ   = 40000000
CSET attribute C_BAUDRATE   = 19200
CSET attribute C_USE_PARITY = 0
CSET signal RX = rx
CSET signal TX = tx
CSET signal Interrupt = interrupt, PRIORITY = 1
END

SELECT MASTER microblaze
CSET attribute CONFIGURATION = DOPB_ILMB_DLMB
CSET attribute HW_VER = 1.00.a
CSET signal Clk = sys_clk
CSET signal Reset = sys_reset
CSET signal Interrupt = interrupt
CSET attribute C_LM_BASEADDR = 0x00000000
CSET attribute C_LM_HIGHADDR = 0x00001fff
END
```

Example MSS File snippet

```
SELECT INSTANCE myuart
CSET attribute DRIVER = drv_uartlite
CSET attribute INT_HANDLER = uart_int_handler, Interrupt
END
```

Example C Program

```
#include <uartlite.h>
#include <mb_interface.h>
#include <mbio.h>

/*
 * Interrupt service routine for the uartlite.  It reads characters from
 * the UART until there are no more characters to read.  If it finds a space
 * in the input, or if the internal buffer is filled, it prints the string
 * read so far.
 */

#define MAXCHARS 100
char strread[MAXCHARS+1];
int charposn = 0;

/* UART Lite interrupt handler */
void uart_int_handler() {
        char c;
        int i;
        while (!uartlite_empty(MYUART_BASEADDR)) {
          /* read a character, and print it out */
          c = uartlite_inbyte(MYUART_BASEADDR);
          strread[charposn++] = c;
          if (charposn == MAXCHARS || c == ' ') {
              for (i = 0; i < charposn; i++)
                uartlite_outbyte(MYUART_BASEADDR, strread[i]);
              charposn = 0;
          }
        }
}
```

```
void
main() {

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

  /* Enable interrupts on the UART Lite */
  uartlite_enable_intr(MYUART_BASEADDR);

  /* Wait for interrupts to occur */
  while (1)
      ;

}
```

System with an Interrupt Controller

System with One or More Interrupt Signals

An Interrupt Controller peripheral (**intc**) should be present if more than one interrupt can be raised. When an interrupt is raised, the interrupt handler for the Interrupt Controller (**intc_interrupt_handler**) is called. The intc_interrupt_handler function then accesses the interrupt controller to find the highest priority device that raised an interrupt. This is done via the _interrupt_vector_table created automatically in **MICROBLAZE_PROJECT/libsrc/intc** by libgen. On return from the peripheral interrupt handler, intc_interrupt_handler acknowledges the interrupt. It then handles any lower priority interrupts, if they exist.

Procedure

To set up a system with a one or more interrupting devices and an interrupt controller, the following steps must be taken:

1. The MHS and MSS files must be set up as follows:
   - The interrupt signals of all the peripherals must be assigned to an interrupt bus in the MHS file. A priority must be given to each of these signals using the PRIORITY keyword. The interrupt bus must be connected to the input of **intc**. The interrupt signal output of **intc** is then connected to the interrupt input of MicroBlaze.
   - The peripherals must be given instance names using the INSTANCE keyword in the MHS file. Libgen creates a definition in **mbio.h** for <INSTANCE_NAME>_BASEADDR mapped to the base address of each peripheral for use in the user program. Libgen also creates an interrupt mask for each interrupt signal using the priorities as <INSTANCE_NAME>_<INTERRUPT_SIGNAL_NAME>_MASK. This can be used to enable or disable interrupts.
2. The interrupt handler functions for each interruptible peripheral must be written.
3. Each handler function is then designated to be the handler for an interrupt signal using the INT_HANDLER keyword in the MSS file. Note that **intc** interrupt signal must not be given an INT_HANDLER keyword. If the INT_HANDLER keyword is not present for a particular peripheral, a default dummy interrupt handler is used.
4. Libgen and mb-gcc is run to achieve the following:
   - **intc_interrupt_handler** function is marked as the main interrupt handler by mb-gcc using the *interrupt_handler* attribute. All volatile registers used by this function are saved. Also, this function will return using the *rtid* instruction, rather than the normal *rtsd* instruction. Furthermore, this function will also be given the name *_interrupt_handler.* By default, MicroBlaze turns off interrupts from the time an interrupt is recognized until the corresponding rtid instruction is executed.

- each peripheral handler function is marked using *save_volatiles* attribute by libgen. All volatile registers used by this function are saved. Since this is not the main interrupt handler, it will return using the normal *rtsd* instruction.
- an interrupt vector table is generated and compiled automatically by libgen. This table is accessed by intc_interrupt_handler to call peripheral interrupt handlers in order of priority.
- the startup code (crt0, crt1, crt2 or crt3) places the address of _interrupt_handler as the target address that MicroBlaze jumps to when an interrupt occurs. Therefore control will go to the intc interrupt handler when an interrupt occurs.

Example MHS File Snippet

```
SELECT SLAVE opb_timer
CSET attribute INSTANCE = mytimer
CSET attribute HW_VER = 1.00.a
CSET attribute C_BASEADDR = 0xFFFF0000
CSET attribute C_HIGHADDR = 0xFFFF00ff
CSET attribute C_AWIDTH   = 32
CSET attribute C_DWIDTH   = 32
CSET signal Interrupt = int_bus, PRIORITY=2
CSET signal CaptureTrig0 = net_gnd
END

SELECT SLAVE opb_uartlite
CSET attribute INSTANCE = myuart
cset attribute HW_VER = 1.00.a
CSET attribute C_BASEADDR  = 0xFFFF8000
CSET attribute C_HIGHADDR = 0xFFFF80FF
CSET attribute C_DATA_BITS  = 8
CSET attribute C_CLK_FREQ   = 30000000
CSET attribute C_BAUDRATE   = 19200
CSET attribute C_USE_PARITY = 0
CSET signal RX = rx
CSET signal TX = tx
CSET signal Interrupt = int_bus, PRIORITY=1
END

SELECT SLAVE opb_intc
CSET attribute INSTANCE = myintc
CSET attribute HW_VER = 1.00.a
CSET attribute C_BASEADDR = 0xFFFF1000
CSET attribute C_HIGHADDR = 0xFFFF10ff
CSET attribute C_NUM_INTR_INPUTS = 2
CSET attribute C_KIND_OF_INTR = 1
CSET attribute C_KIND_OF_EDGE = 1
CSET signal Irq = interrupt
CSET signal Int = int_bus
END

SELECT MASTER microblaze
CSET attribute INSTANCE = microblaze
CSET attribute HW_VER = 1.00.a
CSET attribute CONFIGURATION = DOPB_ILMB_DLMB
CSET signal Interrupt = interrupt
CSET attribute C_LM_BASEADDR = 0x00000000
CSET attribute C_LM_HIGHADDR = 0x00000fff
END
```

Example MSS File Snippet

```
SELECT INSTANCE mytimer
CSET attribute DRIVER = drv_timer
CSET attribute DRIVER_VER = 1.00.a
CSET attribute INT_HANDLER = timer_int_handler, Interrupt
END

SELECT INSTANCE myuart
CSET attribute DRIVER = drv_uartlite
CSET attribute DRIVER_VER = 1.00.a
CSET attribute INT_HANDLER = uart_int_handler, Interrupt
END

SELECT INSTANCE myintc
CSET attribute DRIVER_VER = 1.00.a
CSET attribute DRIVER = drv_intc
END

SELECT INSTANCE microblaze
CSET attribute DRIVER_VER = 1.00.a
CSET attribute DRIVER = drv_microblaze
END
```

Example C Program

```
/*
 * This program uses the Timebase to calculate the number of cycles
 * required by a piece of code. It reads the TimeBase Register at the
 * start and end of the code segment.  It also uses an interrupt handler
 * to count the number of times the TimeBase rolls over.
 */
#include <timebase_wdt.h>
#include <mbio.h>
#include <mb_interface.h>
#include <intc.h>


unsigned int num_rollovers = 0;

/* Interrupt handler for the Timebase */
void timebase_int_handler() {
        num_rollovers++;
}

void
main() {
  int i;
  int j = 1;
  int timebase;
  int rollovers;

  /* Enable microblaze interrupts */
  microblaze_enable_interrupts();

  /* Start the interrupt controller */
  intc_start(MY_INTC_BASEADDR);

  /* print the current value of the timebase */
  print("Timebase value at start ");
  putnum(timebase_wdt_get_timebase(MY_TIMEBASE_BASEADDR));
```

```
    /* Enable interrupts for the timebase */
    intc_enable_interrupt(MY_INTC_BASEADDR,
                          MY_TIMEBASE_TIMEBASE_INTERRUPT_MASK);

    /* Piece of code to be timed */
    for (i=0xffffff; i>0; i--) {
          j = j*i;
    }

    /* Save the number of rollovers and the elapsed time */
    timebase = timebase_wdt_get_timebase(MY_TIMEBASE_BASEADDR);
    rollovers = num_rollovers;

    /* Print the elapsed time */
    print("Timebase value at end ");
    putnum(timebase);

    /* Print the number of rollovers */
    print("Number of timebase rollovers ");
    putnum(rollovers);
}
```

Breakpoints in Interrupt Handlers

Certain precautions must be taken when debugging programs that have interrupt handler routines. These are enumerated below.

1. Breakpoints can be present in the interrupt handlers when interrupts are not enabled by the user in the interrupt handler routines. In this case, there must not be any breakpoints in other parts of the code.
2. Breakpoints can be present outside the interrupt handler routines. It is acceptable for interrupts to occur when **xmd** is servicing a breakpoint. Please note that a sufficient delay must be given to **xmd** (using the -d option) so that **xmd** waits until interrupts are serviced.
3. There should never be breakpoints in both interrupt handler routines and other user code.

Microblaze Instruction Set Architecture

XILINX®

Mar. 18, 2002

MicroBlaze Instruction Set Architecture

Summary

This document provides a detailed guide to the Instruction Set Architecture of MicroBlaze™.

Notation

The symbols used throughout this document are defined in Table 1.

Table 1: **Symbol notation**

Symbol	Meaning
+	Add
-	Subtract
×	Multiply
∧	Bitwise logical AND
∨	Bitwise logical OR
⊕	Bitwise logical XOR
$\overline{x}$	Bitwise logical complement of *x*
←	Assignment
r*x*	Register *x*
x[*i*]	Bit *i* in register *x*
x[*i*:*j*]	Bits *i* through *j* in register *x*
=	Equal comparison
≠	Not equal comparison
>	Greater than comparison
>=	Greater than or equal comparison
<	Less than comparison
<=	Less than or equal comparison
sext(*x*)	Sign-extend *x*
Mem(*x*)	Memory location at address *x*

Formats

MicroBlaze uses two instruction formats: Type A and Type B.

Type A

Type A is used for register-register instructions. It contains the opcode, one destination and two source registers.

Opcode	Destination Reg	Source Reg A	Source Reg B	0	0	0	0	0	0	0	0	0	0	0
0	6	11	16	21										31

Type B

Type B is used for register-immediate instructions. It contains the opcode, one destination and one source registers, and a source 16-bit immediate value.

Opcode	Destination Reg	Source Reg A	Immediate Value
0	6	11	16 31

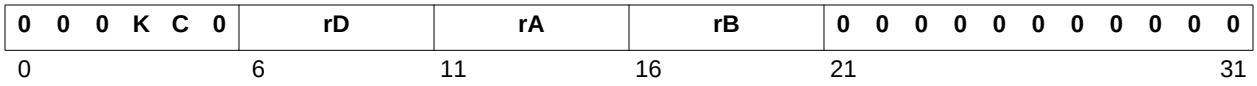
Instructions

MicroBlaze instructions are described next. Instructions are listed in alphabetical order. For each instruction we provide the mnemonic, encoding, a description of it, pseudocode of the instruction semantics, and the registers that the instruction alters.

add

Arithmetic Add

add	rD, rA, rB	Add
addc	rD, rA, rB	Add with Carry
addk	rD, rA, rB	Add and Keep Carry
addkc	rD, rA, rB	Add with Carry and Keep Carry



Description

The sum of the contents of registers rA and rB, is placed into register rD.

Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic addk. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic addc. Both bits are set to a one for the mnemonic addkc.

When an add instruction has bit 3 set (addk, addkc), the carry flag will Keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (add, addc), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (addc, addkc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (add, addk), the content of the carry flag does not affect the execution of the instruction (providing a normal addition).

Pseudocode

```
if C = 0 then
  (rD) ← (rA) + (rB)
else
  (rD) ← (rA) + (rB) + MSR[C]
if K = 0 then
  MSR[C] ← CarryOut
```

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

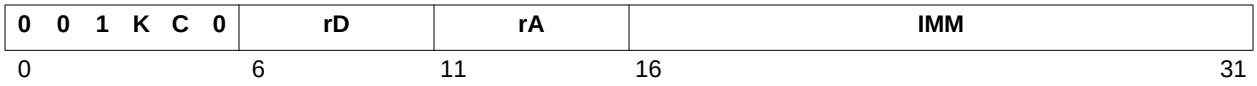
Note

The C bit in the instruction opcode is not the same as the carry bit in the MSR register.

addi

Arithmetic Add Immediate

addi	rD, rA, IMM	Add Immediate
addic	rD, rA, IMM	Add Immediate with Carry
addik	rD, rA, IMM	Add Immediate and Keep Carry
addikc	rD, rA, IMM	Add Immediate with Carry and Keep Carry



Description

The sum of the contents of registers rA and the value in the IMM field, sign-extended to 32 bits, is placed into register rD.

Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic addik. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic addic. Both bits are set to a one for the mnemonic addikc.

When an addi instruction has bit 3 set (addik, addikc), the carry flag will Keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (addi, addic), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (addic, addikc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (addi, addik), the content of the carry flag does not affect the execution of the instruction (providing a normal addition).

Pseudocode

```
if C = 0 then
  (rD) ← (rA) + sext(IMM)
else
  (rD) ← (rA) + sext(IMM) + MSR[C]
if K = 0 then
  MSR[C] ← CarryOut
```

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

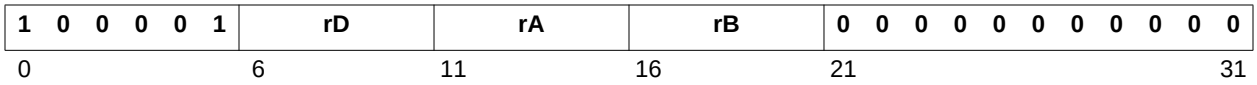
The C bit in the instruction opcode is not the same as the carry bit in the MSR register.

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

and

Logical AND

and rD, rA, rB



Description

The contents of register rA are ANDed with the contents of register rB; the result is placed into register rD.

Pseudocode

(rD) $\leftarrow$ (rA) $\wedge$ (rB)

Registers Altered

- rD

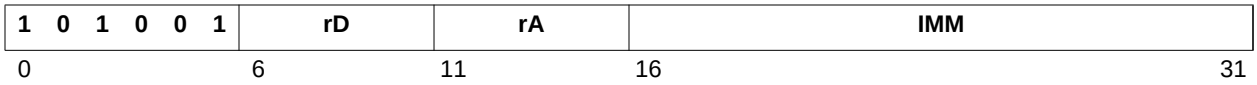
Latency

1 cycle

andi

Logial AND with Immediate

andi rD, rA, IMM



Description

The contents of register rA are ANDed with the value of the IMM field, sign-extended to 32 bits; the result is placed into register rD.

Pseudocode

```
(rD) ← (rA) ∧ sext(IMM)
```

Registers Altered

- rD

Latency

1 cycle

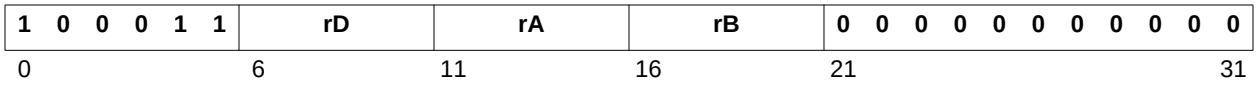
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an IMM instruction. See IMM instruction for details on using 32-bit immediate values.

andn

Logical AND NOT

andn rD, rA, rB



Description

The contents of register rA are ANDed with the logical complement of the contents of register rB; the result is placed into register rD.

Pseudocode

$(rD) \leftarrow (rA) \wedge (\overline{rB})$

Registers Altered

- rD

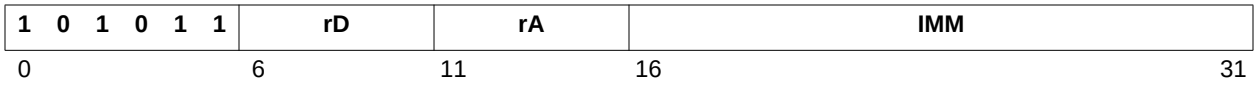
Latency

1 cycle

andni

Logical AND NOT with Immediate

andni rD, rA, IMM



Description

The IMM field is sign-extended to 32 bits. The contents of register rA are ANDed with the logical complement of the extended IMM field; the result is placed into register rD.

Pseudocode

$(rD) \leftarrow (rA) \wedge (\overline{sext(IMM)})$

Registers Altered

- rD

Latency

1 cycle

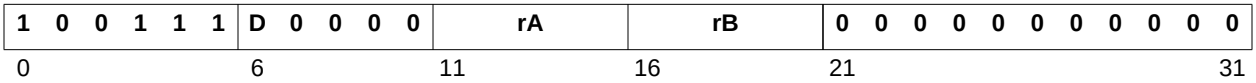
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

beq

Branch if Equal

beq	rA, rB	Branch if Equal
beqd	rA, rB	Branch if Equal with Delay



Description

Branch if rA is equal to 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic beqd will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA = 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

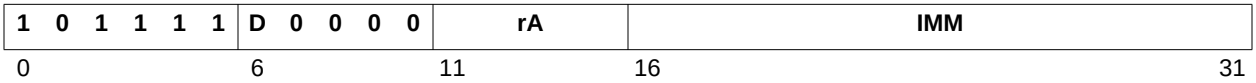
2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

beqi

Branch Immediate if Equal

beqi	rA, IMM	Branch Immediate if Equal
beqid	rA, IMM	Branch Immediate if Equal with Delay



Description

Branch if rA is equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic beqid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA = 0 then
  PC ← PC + sext(IMM)
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

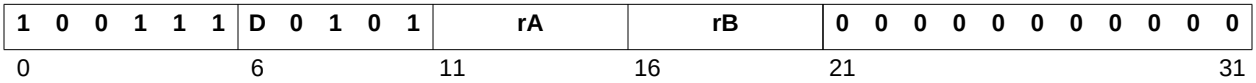
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

bge

Branch if Greater or Equal

bge	rA, rB	Branch if Greater or Equal
bged	rA, rB	Branch if Greater or Equal with Delay



Description

Branch if rA is greater or equal to 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic bged will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA >= 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

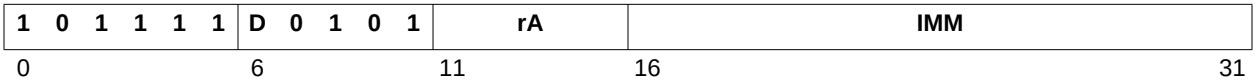
2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

bgei

Branch Immediate if Greater or Equal

bgei	rA, IMM	Branch Immediate if Greater or Equal
bgeid	rA, IMM	Branch Immediate if Greater or Equal with Delay



Description

Branch if rA is greater or equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic bgeid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA >= 0 then
  PC ← PC + sext(IMM)
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

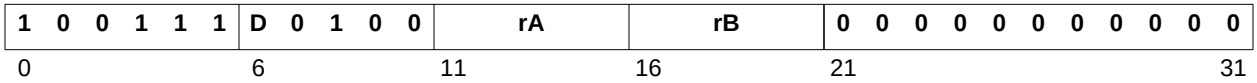
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

bgt

Branch if Greater Than

bgt	rA, rB	Branch if Greater Than
bgtd	rA, rB	Branch if Greater Than with Delay



Description

Branch if rA is greater than 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic bgtd will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA > 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

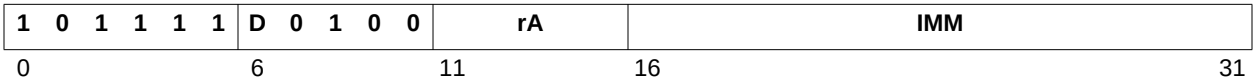
2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

bgti

Branch Immediate if Greater Than

bgti	rA, IMM	Branch Immediate if Greater Than
bgtid	rA, IMM	Branch Immediate if Greater Than with Delay



Description

Branch if rA is greater than 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic bgtid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA > 0 then
  PC ← PC + sext(IMM)
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

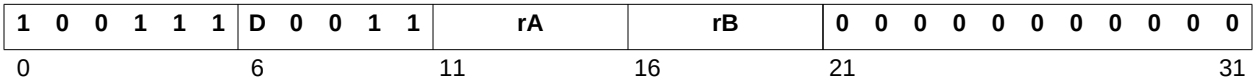
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

ble

Branch if Less or Equal

ble	rA, rB	Branch if Less or Equal
bled	rA, rB	Branch if Less or Equal with Delay



Description

Branch if rA is less or equal to 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic bled will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA <= 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

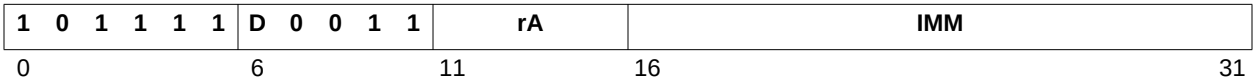
2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

blei

Branch Immediate if Less or Equal

blei	rA, IMM	Branch Immediate if Less or Equal
bleid	rA, IMM	Branch Immediate if Less or Equal with Delay



Description

Branch if rA is less or equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic bleid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA <= 0 then
  PC ← PC + sext(IMM)
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

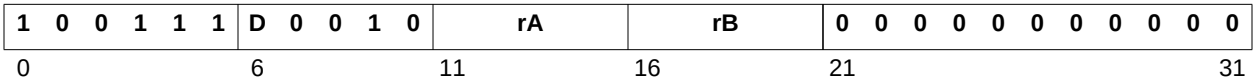
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

blt

Branch if Less Than

blt	rA, rB	Branch if Less Than
bltd	rA, rB	Branch if Less Than with Delay



Description

Branch if rA is less than 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic bltd will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA < 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

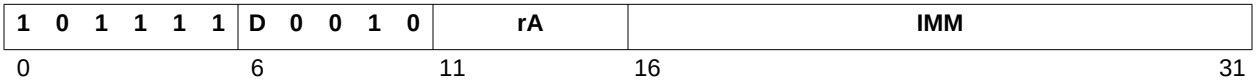
2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

blti

Branch Immediate if Less Than

blti	rA, IMM	Branch Immediate if Less Than
bltid	rA, IMM	Branch Immediate if Less Than with Delay



Description

Branch if rA is less than 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic bltid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA < 0 then
  PC ← PC + sext(IMM)
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

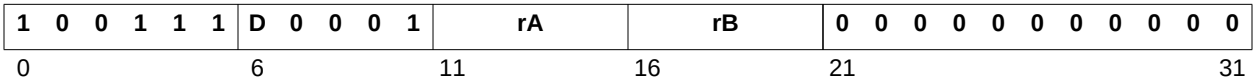
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

bne

Branch if Not Equal

bne	rA, rB	Branch if Not Equal
bned	rA, rB	Branch if Not Equal with Delay



Description

Branch if rA not equal to 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic bned will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA ≠ 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

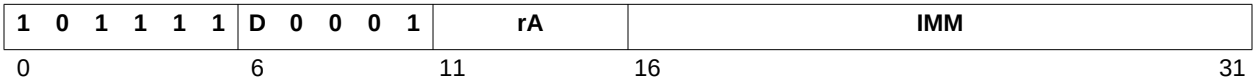
2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

bnei

Branch Immediate if Not Equal

bnei	rA, IMM	Branch Immediate if Not Equal
bneid	rA, IMM	Branch Immediate if Not Equal with Delay



Description

Branch if rA not equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic bneid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
If rA ≠ 0 then
  PC ← PC + sext(IMM)
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

3 cycles (if branch is taken and the D bit is not set)

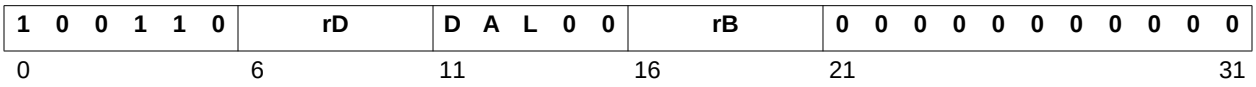
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

br

Unconditional Branch

br	rB	Branch
bra	rB	Branch Absolute
brd	rB	Branch with Delay
brad	rB	Branch Absolute with Delay
brld	rD, rB	Branch and Link with Delay
brald	rD, rB	Branch Absolute and Link with Delay



Description

Branch to the instruction located at address determined by rB.

The mnemonics brld and brald will set the L bit. If the L bit is set, linking will be performed. The current value of PC will be stored in rD.

The mnemonics bra, brad and brald will set the A bit. If the A bit is set, it means that the branch is to an absolute value and the target is the value in rB, otherwise, it is a relative branch and the target will be PC + rB.

The mnemonics brd, brad, brld and brald will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
if L = 1 then
  (rD) ← PC
if A = 1 then
  PC ← (rB)
else
  PC ← PC + (rB)
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- rD
- PC

Latency

2 cycles (if the D bit is set)

3 cycles (if the D bit is not set)

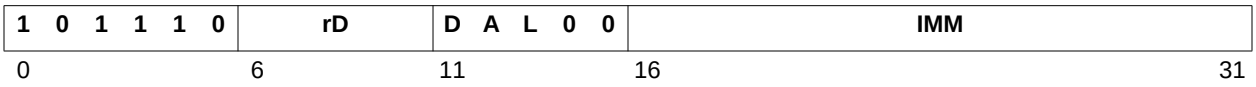
Note

The instructions brl and bral are not available.

bri

Unconditional Branch Immediate

bri	IMM	Branch Immediate
brai	IMM	Branch Absolute Immediate
brid	IMM	Branch Immediate with Delay
braid	IMM	Branch Absolute Immediate with Delay
brlid	rD, IMM	Branch and Link Immediate with Delay
bralid	rD, IMM	Branch Absolute and Link Immediate with Delay



Description

Branch to the instruction located at address determined by IMM, sign-extended to 32 bits.

The mnemonics brlid and bralid will set the L bit. If the L bit is set, linking will be performed. The current value of PC will be stored in rD.

The mnemonics brai, braid and bralid will set the A bit. If the A bit is set, it means that the branch is to an absolute value and the target is the value in IMM, otherwise, it is a relative branch and the target will be PC + IMM.

The mnemonics brid, braid, brlid and bralid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

```
if L = 1 then
  (rD) ← PC
if A = 1 then
  PC ← (rB)
else
  PC ← PC + (rB)
if D = 1 then
  allow following instruction to complete execution
```

Registers Altered

- rD
- PC

Latency

2 cycles (if the D bit is set)

3 cycles (if the D bit is not set)

Notes

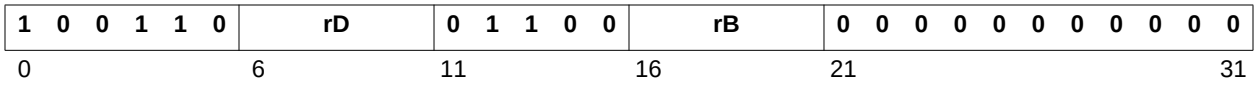
The instructions brli and brali are not available.

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

brk

Break

brk rD, rB



Description

Branch and link to the instruction located at address value in rB. The current value of PC will be stored in rD. The BIP flag in the MSR will be set.

Pseudocode

```
(rD) ← PC
PC ← (rB)
MSR[BIP] ← 1
```

Registers Altered

- rD
- PC
- MSR[BIP]

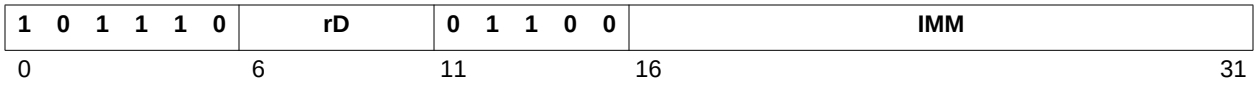
Latency

3 cycles

brki

Break Immediate

brki rD, IMM



Description

Branch and link to the instruction located at address value in IMM, sign-extended to 32 bits. The current value of PC will be stored in rD. The BIP flag in the MSR will be set.

Pseudocode

```
(rD) ← PC
PC ← sext(IMM)
MSR[BIP] ← 1
```

Registers Altered

- rD
- PC
- MSR[BIP]

Latency

3 cycles

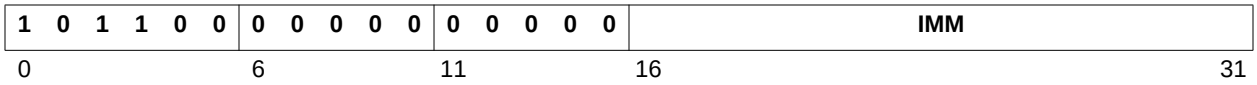
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

imm

Immediate

imm IMM



Description

The instruction imm loads the IMM value into a temporary register. It also locks this value so it can be used by the following instruction and form a 32-bit immediate value.

The instruction imm is used in conjunction with Type B instructions. Since Type B instructions have only a 16-bit immediate value field, a 32-bit immediate value cannot be used directly. However, 32-bit immediate values can be used in MicroBlaze. By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. The imm instruction locks the 16-bit IMM value temporarily for the next instruction. A Type B instruction that immediately follows the imm instruction will then form a 32-bit immediate value from the 16-bit IMM value of the imm instruction (upper 16 bits) and its own 16-bit immediate value field (lower 16 bits). If no Type B instruction follows the IMM instruction, the locked value gets unlocked and becomes useless.

Latency

1 cycle

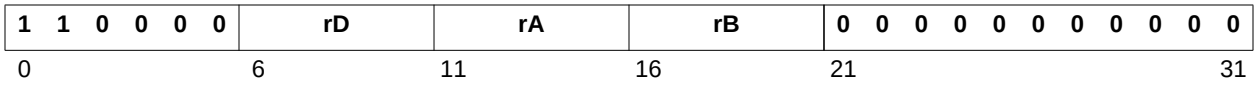
Note

The imm instruction and the Type B instruction following it are atomic, hence no interrupts are allowed between them.

lbu

Load Byte Unsigned

lbu rD, rA, rB



Description

Loads a byte (8 bits) from the memory location that results from adding the contents of registers rA and rB. The data is placed in the least significant byte of register rD and the other three bytes in rD are cleared.

Pseudocode

```
Addr ← (rA) + (rB)
(rD)[24:31] ← Mem(Addr)
(rD)[0:23]  ← 0
```

Registers Altered

- rD

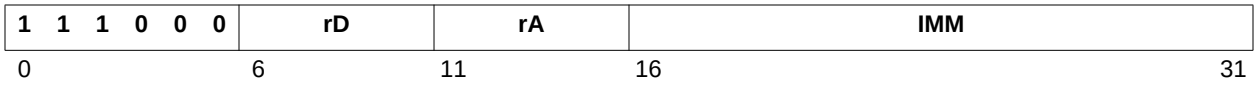
Latency

2 cycles

lbui

Load Byte Unsigned Immediate

lbui rD, rA, IMM



Description

Loads a byte (8 bits) from the memory location that results from adding the contents of register rA with the value in IMM, sign-extended to 32 bits. The data is placed in the least significant byte of register rD and the other three bytes in rD are cleared.

Pseudocode

```
Addr ← (rA) + sext(IMM)
(rD)[24:31]  ← Mem(Addr)
(rD)[0:23]  ← 0
```

Registers Altered

- rD

Latency

2 cycles

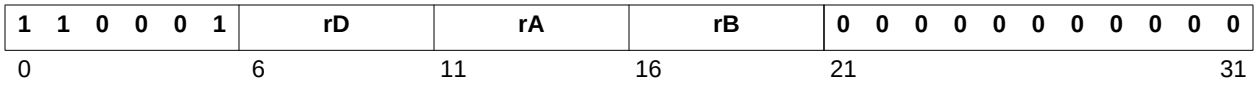
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

lhu

Load Halfword Unsigned

lhu rD, rA, rB



Description

Loads a halfword (16 bits) from the halfword aligned memory location that results from adding the contents of registers rA and rB. The data is placed in the least significant halfword of register rD and the most significant halfword in rD is cleared.

Pseudocode

```
Addr ← (rA) + (rB)
Addr[31] ← 0
(rD)[16:31] ← Mem(Addr)
(rD)[0:15] ← 0
```

Registers Altered

- rD

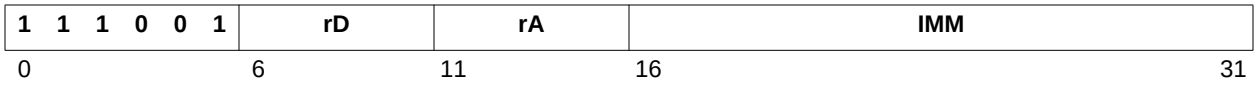
Latency

2 cycles

lhui

Load Halfword Unsigned Immediate

lhui rD, rA, IMM



Description

Loads a halfword (16 bits) from the halfword aligned memory location that results from adding the contents of register rA and the value in IMM, sign-extended to 32 bits. The data is placed in the least significant halfword of register rD and the most significant halfword in rD is cleared.

Pseudocode

```
Addr ← (rA) + sext(IMM)
Addr[31] ← 0
(rD)[16:31] ← Mem(Addr)
(rD)[0:15] ← 0
```

Registers Altered

- rD

Latency

2 cycles

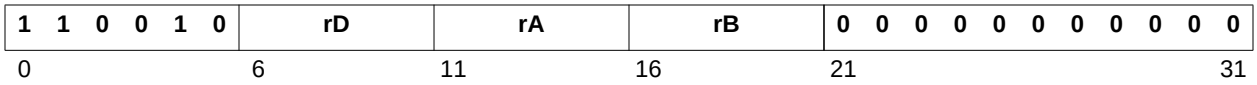
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

lw

Load Word

lw rD, rA, rB



Description

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of registers rA and rB. The data is placed in register rD.

Pseudocode

```
Addr ← (rA) + (rB)
Addr[30:31] ← 00
(rD) ← Mem(Addr)
```

Registers Altered

- rD

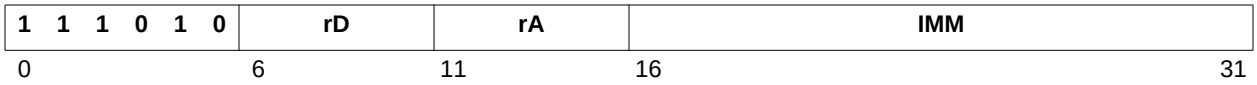
Latency

2 cycles

lwi

Load Word Immediate

lwi rD, rA, IMM



Description

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits. The data is placed in register rD.

Pseudocode

```
Addr ← (rA) + sext(IMM)
Addr[30:31] ← 00
(rD) ← Mem(Addr)
```

Registers Altered

- rD

Latency

2 cycles

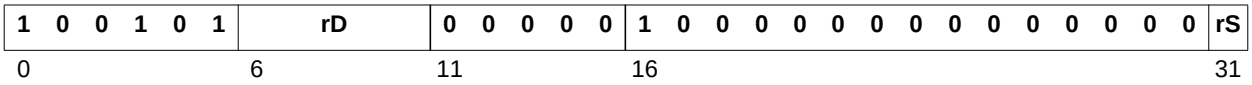
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

mfs

Move From Special Purpose Register

mfs rD, rS



Description

Copies the contents of the special purpose register rS into register rD.

Pseudocode

```
(rD) ← (rS)
```

Registers Altered

- rD

Latency

1 cycle

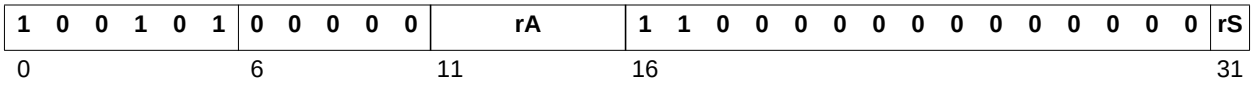
Note

To refer to special purpose registers in assembly language, use rpc for PC and rmsr for MSR.

mts

Move To Special Purpose Register

mts rS, rA



Description

Copies the contents of register rD into the MSR register.

Pseudocode

```
(rS) ← (rA)
```

Registers Altered

- rS

Latency

1 cycle

Notes

You can not write to the PC using the MTS instruction.

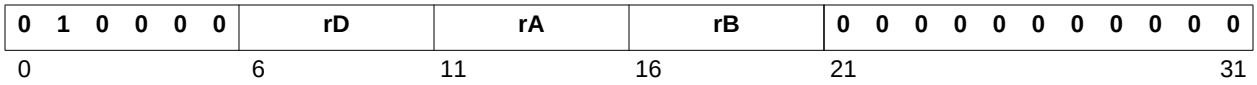
When writing to MSR using MTS, the value written will take effect one clock cycle after executing the MTS instruction.

To refer to special purpose registers in assembly language, use rpc for PC and rmsr for MSR.

mul

Multiply

mul rD, rA, rB



Description

Multiplies the contents of registers rA and rB and puts the result in register rD. This is a 32-bit by 32-bit multiplication that will produce a 64-bit result. The least significant word of this value is placed in rD.

Pseudocode

(rD) ← (rA) × (rB)

Registers Altered

- rD

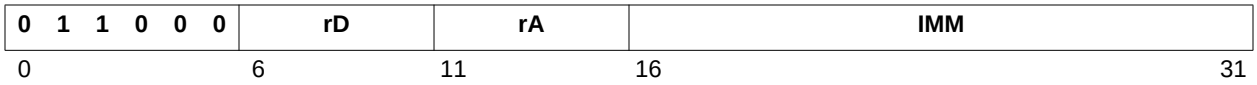
Latency

3 cycles

muli

Multiply Immediate

muli rD, rA, IMM



Description

Multiplies the contents of registers rA and the value IMM, sign-extended to 32 bits; and puts the result in register rD. This is a 32-bit by 32-bit multiplication that will produce a 64-bit result. The least significant word of this value is placed in rD.

Pseudocode

```
(rD) ← (rA) × sext(IMM)
```

Registers Altered

- rD

Latency

3 cycles

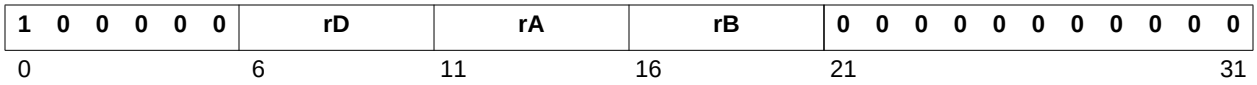
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

or

Logical OR

or rD, rA, rB



Description

The contents of register rA are ORed with the contents of register rB; the result is placed into register rD.

Pseudocode

```
(rD) ← (rA) ∨ (rB)
```

Registers Altered

- rD

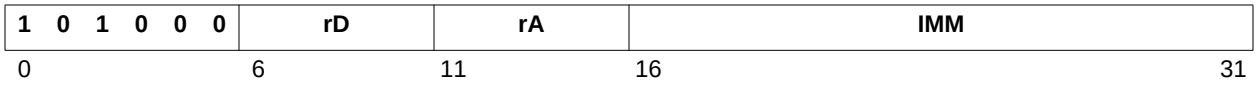
Latency

1 cycle

ori

Logical OR with Immediate

ori rD, rA, IMM



Description

The contents of register rA are ORed with the extended IMM field, sign-extended to 32 bits; the result is placed into register rD.

Pseudocode

```
(rD) ← (rA) ∨ (IMM)
```

Registers Altered

- rD

Latency

1 cycle

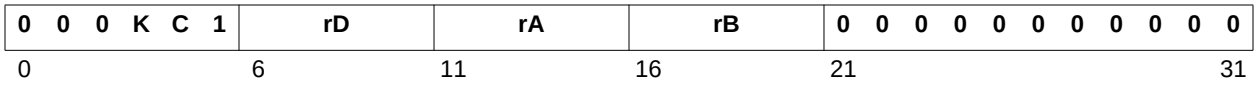
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

rsub

Arithmetic Reverse Subtract

rsub	rD, rA, rB	Subtract
rsubc	rD, rA, rB	Subtract with Carry
rsubk	rD, rA, rB	Subtract and Keep Carry
rsubkc	rD, rA, rB	Subtract with Carry and Keep Carry



Description

The contents of register rA is subtracted from the contents of register rB and the result is placed into register rD.

Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic rsubk. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic rsubc. Both bits are set to a one for the mnemonic rsubkc.

When an rsub instruction has bit 3 set (rsubk, rsubkc), the carry flag will Keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (rsub, rsubc), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (rsubc, rsubkc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (rsub, rsubk), the content of the carry flag does not affect the execution of the instruction (providing a normal subtraction).

Pseudocode

```
if C = 0 then
  (rD) ← (rB) + (rA) + 1
else
  (rD) ← (rB) + (rA) + MSR[C]
if K = 0 then
  MSR[C] ← CarryOut
```

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

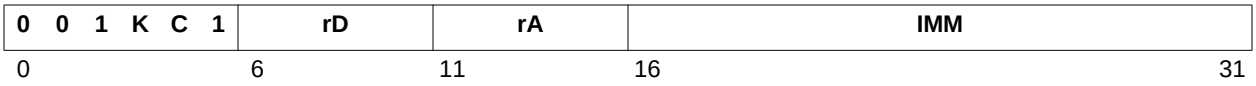
Notes

In subtractions, Carry = (Borrow). When the Carry is set by a subtraction, it means that there is no Borrow, and when the Carry is cleared, it means that there is a Borrow.

rsubi

Arithmetic Reverse Subtract Immediate

rsubi	rD, rA, IMM	Subtract Immediate
rsubic	rD, rA, IMM	Subtract Immediate with Carry
rsubik	rD, rA, IMM	Subtract Immediate and Keep Carry
rsubikc	rD, rA, IMM	Subtract Immediate with Carry and Keep Carry



Description

The contents of register rA is subtracted from the value of IMM, sign-extended to 32 bits, and the result is placed into register rD.

Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic rsubik. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic rsubic. Both bits are set to a one for the mnemonic rsubikc.

When an rsubi instruction has bit 3 set (rsubik, rsubikc), the carry flag will Keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (rsubi, rsubic), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (rsubic, rsubikc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (rsubi, rsubik), the content of the carry flag does not affect the execution of the instruction (providing a normal subtraction).

Pseudocode

```
if C = 0 then
  (rD) ← sext(IMM) + (rA) + 1
else
  (rD) ← sext(IMM) + (rA) + MSR[C]
if K = 0 then
  MSR[C] ← CarryOut
```

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

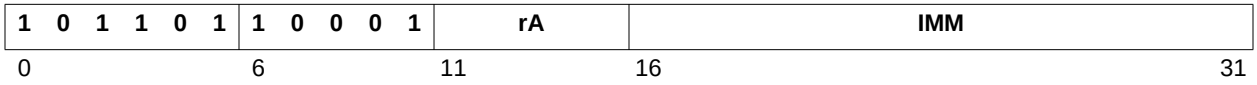
In subtractions, Carry = ($\overline{\text{Borrow}}$). When the Carry is set by a subtraction, it means that there is no Borrow, and when the Carry is cleared, it means that there is a Borrow.

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

rtbd

Return from Break

rtbd rA, IMM



Description

Return from break will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits. It will also enable breaks after execution by clearing the BIP flag in the MSR.

This instruction always has a delay slot. The instruction following the RTBD is always executed before the branch target. That delay slot instruction has breaks disabled.

Pseudocode

```
PC ← (rA) + sext(IMM)
allow following instruction to complete execution
MSR[BIP] ← 0
```

Registers Altered

- PC
- MSR[BIP]

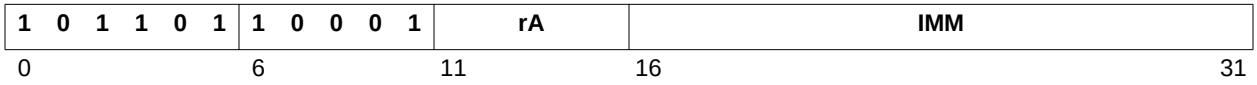
Latency

2 cycles

rtid

Return from Interrupt

rtid rA, IMM



Description

Return from interrupt will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits. It will also enable interrupts after execution.

This instruction always has a delay slot. The instruction following the RTID is always executed before the branch target. That delay slot instruction has interrupts disabled.

Pseudocode

```
PC ← (rA) + sext(IMM)
allow following instruction to complete execution
MSR[IE] ← 1
```

Registers Altered

- PC
- MSR[IE]

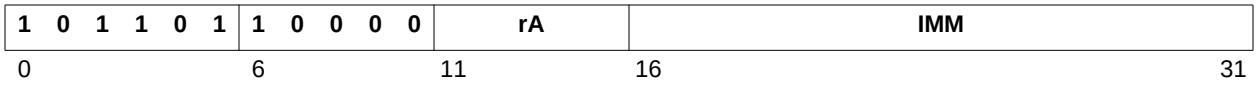
Latency

2 cycles

rtsd

Return from Subroutine

rtsd rA, IMM



Description

Return from subroutine will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits.

This instruction always has a delay slot. The instruction following the RTSD is always executed before the branch target.

Pseudocode

```
PC ← (rA) + sext(IMM)
allow following instruction to complete execution
```

Registers Altered

- PC

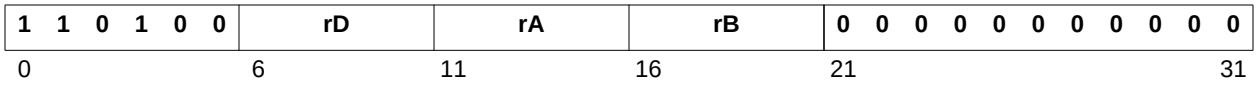
Latency

2 cycles

sb

Store Byte

sb rD, rA, rB



Description

Stores the contents of the least significant byte of register rD, into the memory location that results from adding the contents of registers rA and rB.

Pseudocode

```
Addr ← (rA) + (rB)
Mem(Addr) ←(rD)[24:31]
```

Registers Altered

- None

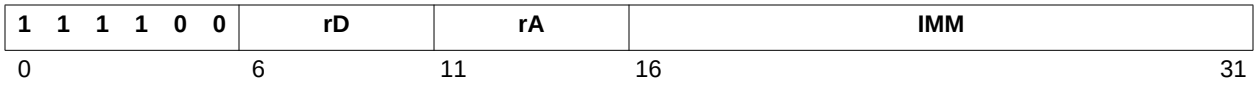
Latency

2 cycles

sbi

Store Byte Immediate

sbi rD, rA, IMM



Description

Stores the contents of the least significant byte of register rD, into the memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits.

Pseudocode

```
Addr ← (rA) + sext(IMM)
Mem(Addr) ← (rD)[24:31]
```

Registers Altered

- None

Latency

2 cycles

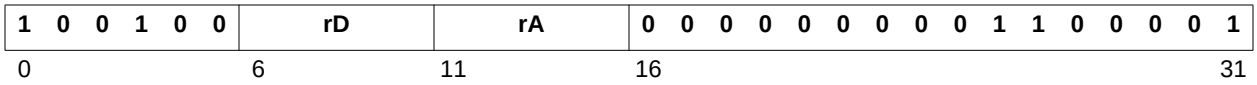
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

sext16

Sign Extend Halfword

sext16 rD, rA



Description

This instruction sign-extends a halfword (16 bits) into a word (32 bits). Bit 16 in rA will be copied into bits 0-15 of rD. Bits 16-31 in rA will be copied into bits 16-31 of rD.

Pseudocode

```
(rD)[0:15] ← (rA)[16]
(rD)[16:31] ← (rA)[16:31]
```

Registers Altered

- rD

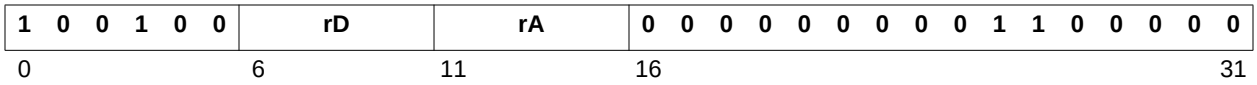
Latency

1 cycle

sext8

Sign Extend Byte

sext8 rD, rA



Description

This instruction sign-extends a byte (8 bits) into a word (32 bits). Bit 24 in rA will be copied into bits 0-23 of rD. Bits 24-31 in rA will be copied into bits 24-31 of rD.

Pseudocode

```
(rD)[0:23] ← (rA)[24]
(rD)[24:31] ← (rA)[24:31]
```

Registers Altered

- rD

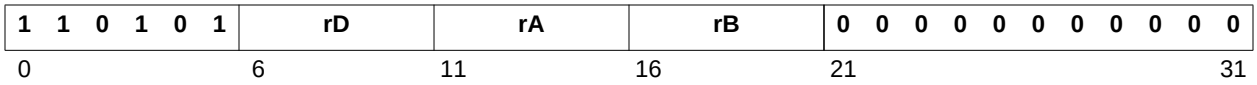
Latency

1 cycle

sh

Store Halfword

sh rD, rA, rB



Description

Stores the contents of the least significant halfword of register rD, into the halfword aligned memory location that results from adding the contents of registers rA and rB.

Pseudocode

```
Addr ← (rA) + (rB)
Addr[31] ← 0
Mem(Addr) ← (rD)[16:31]
```

Registers Altered

- None

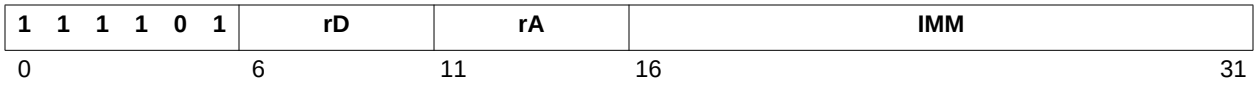
Latency

2 cycles

shi

Store Halfword Immediate

shi rD, rA, IMM



Description

Stores the contents of the least significant halfword of register rD, into the halfword aligned memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits.

Pseudocode

```
Addr ← (rA) + sext(IMM)
Addr[31] ← 0
Mem(Addr) ←(rD)[16:31]
```

Registers Altered

- None

Latency

2 cycles

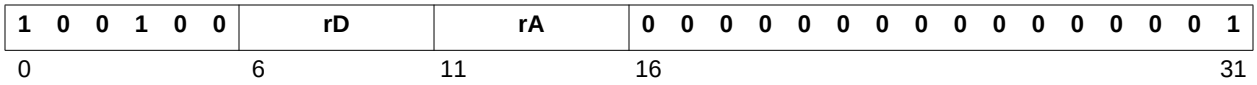
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

sra

Shift Right Arithmetic

sra rD, rA



Description

Shifts arithmetically the contents of register rA, one bit to the right, and places the result in rD. The most significant bit of rA (i.e. the sign bit) placed in the most significant bit of rD. The least significant bit coming out of the shift chain is placed in the Carry flag.

Pseudocode

```
(rD)[0] ← (rA)[0]
(rD)[1:31] ← (rA)[0:30]
MSR[C] ← (rA)[31]
```

Registers Altered

- rD
- MSR[C]

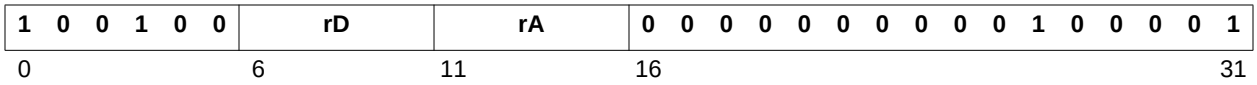
Latency

1 cycle

src

Shift Right with Carry

src rD, rA



Description

Shifts the contents of register rA, one bit to the right, and places the result in rD. The Carry flag is shifted in the shift chain and placed in the most significant bit of rD. The least significant bit coming out of the shift chain is placed in the Carry flag.

Pseudocode

```
(rD)[0] ← MSR[C]
(rD)[1:31] ←(rA)[0:30]
MSR[C] ←(rA)[31]
```

Registers Altered

- rD
- MSR[C]

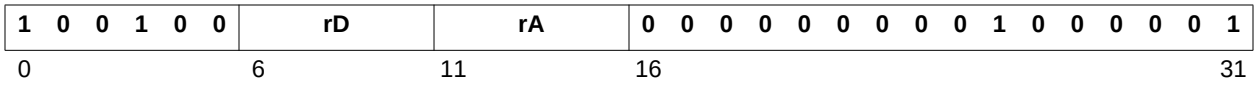
Latency

1 cycle

srl

Shift Right Logical

srl rD, rA



Description

Shifts logically the contents of register rA, one bit to the right, and places the result in rD. A zero is shifted in the shift chain and placed in the most significant bit of rD. The least significant bit coming out of the shift chain is placed in the Carry flag.

Pseudocode

```
(rD)[0] ← 0
(rD)[1:31] ←(rA)[0:30]
MSR[C] ←(rA)[31]
```

Registers Altered

- rD
- MSR[C]

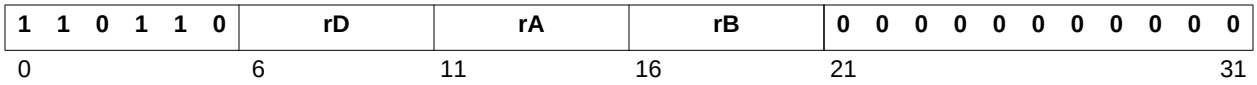
Latency

1 cycle

SW

Store Word

sw rD, rA, rB



Description

Stores the contents of register rD, into the word aligned memory location that results from adding the contents of registers rA and rB.

Pseudocode

```
Addr ← (rA) + (rB)
Addr[30:31] ← 00
Mem(Addr) ←(rD)[0:31]
```

Registers Altered

- None

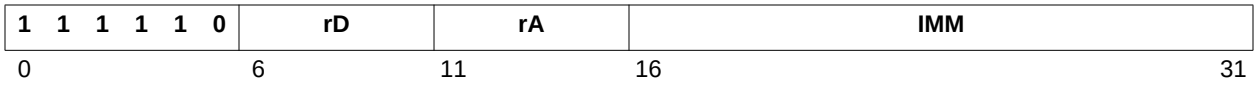
Latency

2 cycles

swi

Store Word Immediate

swi rD, rA, IMM



Description

Stores the contents of register rD, into the word aligned memory location that results from adding the contents of registers rA and the value IMM, sign-extended to 32 bits.

Pseudocode

```
Addr ← (rA) + sext(IMM)
Addr[30:31] ← 00
Mem(Addr) ← (rD)[0:31]
```

Register Altered

- None

Latency

2 cycles

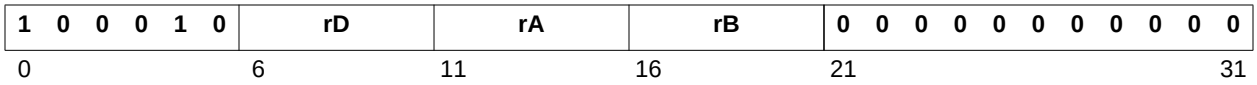
Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

xor

Logical Exclusive OR

xor rD, rA, rB



Description

The contents of register rA are XORed with the contents of register rB; the result is placed into register rD.

Pseudocode

```
(rD) ← (rA) ⊕ (rB)
```

Registers Altered

- rD

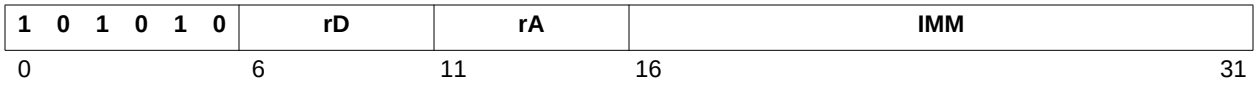
Latency

1 cycle

xori

Logical Exclusive OR with Immediate

xori rA, rD, IMM



Description

The IMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register rA are XORed with the extended IMM field; the result is placed into register rD.

Pseudocode

```
(rD) ← (rA) ⊕ sext(IMM)
```

Registers Altered

- rD

Latency

1 cycle

Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See imm instruction for details on using 32-bit immediate values.

Index