The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/07/00</td>
<td>1.0</td>
</tr>
</tbody>
</table>
# Table of Contents

## Preface: About This Guide
- Manual Contents .................................................. 7
- Additional Resources ............................................. 7
- Conventions ......................................................... 8
  - Typographical .................................................. 8
  - Online Document ............................................... 9

## Chapter 1: MicroBlaze Architecture
- Summary ............................................................ 11
- Overview ........................................................... 11
  - Features ......................................................... 11
- Instructions ........................................................ 12
- Registers ............................................................ 16
  - General Purpose Registers .................................... 16
  - Special Purpose Registers ................................... 17
- Pipeline ............................................................. 19
  - Pipeline Architecture ........................................ 19
  - Branches ....................................................... 20
- Load/Store Architecture .......................................... 20
- Interrupts, Exceptions and Breaks .............................. 21
  - Interrupts ...................................................... 21
  - Exceptions .................................................... 22
  - Breaks ......................................................... 22
- Instruction Cache .................................................. 23
  - Overview ....................................................... 23
  - Cache Organization ........................................... 23
  - Cache Operation ............................................... 24
  - Software ....................................................... 25
  - LMB Memory ................................................... 26
- Data Cache ........................................................... 26
  - Overview ....................................................... 26
  - Cache Organization ........................................... 26
  - Cache Operation ............................................... 27
  - Software ....................................................... 28
  - LMB Memory ................................................... 28
- Fast Simplex Link Interface ..................................... 28
  - FSL Read Instructions ......................................... 29
  - FSL Write Instructions ....................................... 30
- Debug Interface .................................................... 30
  - Debugging Features .......................................... 31

## Chapter 2: MicroBlaze Bus Interfaces
- Summary ............................................................ 33
Overview .................................................. 33
Features .................................................. 33
Bus Configurations ....................................... 33
Bus Configurations
- Typical Peripheral Placement ..................... 35
Bit and Byte Labeling ................................... 43
Core I/O .................................................. 43
Bus Organization ......................................... 45
- OPB Bus Configuration ............................... 45
LMB Bus Configuration ................................. 49
LMB Bus Operations .................................... 50
Read and Write Data Steering ....................... 53
FSL Bus Operation ....................................... 54
Debug Interface .......................................... 56
Implementation .......................................... 57
- Parameterization ..................................... 57

Chapter 3: MicroBlaze Endianness
Definitions .............................................. 59
Bit Naming Conventions ................................ 59
Data Types and Endianness ............................. 59
VHDL Example .......................................... 61
- BRAM – LMB Example ............................... 61
- BRAM – OPB Example ............................... 63

Chapter 4: MicroBlaze Application Binary Interface
Scope ...................................................... 67
Data Types ............................................... 67
Register Usage Conventions ......................... 67
Stack Convention ....................................... 69
- Calling Convention .................................. 70
Memory Model .......................................... 70
- Small data area ..................................... 70
- Data area ............................................. 71
- Common un-initialized area ......................... 71
- Literals or constants ............................... 71
Interrupt and Exception Handling .................. 71

Chapter 5: MicroBlaze Instruction Set Architecture
Summary .................................................. 73
Notation .................................................. 73
Formats .................................................. 74
Instructions .............................................. 74
About This Guide

Welcome to the MicroBlaze Processor Reference Guide. This document provides information about the 32-bit soft processor, MicroBlaze, included in the Embedded Processor Development Kit (EDK). The document is meant as a guide to the MicroBlaze hardware and software architecture.

Manual Contents

This manual discusses the following topics specific to MicroBlaze soft processor:

- Core Architecture
- Bus Interfaces and Endianness
- Application Binary Interface
- Instruction Set Architecture

Additional Resources

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

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description/URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorials</td>
<td>Tutorials covering Xilinx design flows, from design entry to verification and debugging</td>
</tr>
<tr>
<td></td>
<td><a href="http://support.xilinx.com/support/techsup/tutorials/index.htm">http://support.xilinx.com/support/techsup/tutorials/index.htm</a></td>
</tr>
<tr>
<td>Answer Browser</td>
<td>Database of Xilinx solution records</td>
</tr>
<tr>
<td></td>
<td><a href="http://support.xilinx.com/xlnx/xil_ans_browser.jsp">http://support.xilinx.com/xlnx/xil_ans_browser.jsp</a></td>
</tr>
<tr>
<td>Application Notes</td>
<td>Descriptions of device-specific design techniques and approaches</td>
</tr>
<tr>
<td></td>
<td><a href="http://support.xilinx.com/apps/appsweb.htm">http://support.xilinx.com/apps/appsweb.htm</a></td>
</tr>
<tr>
<td>Data Book</td>
<td>Pages from The Programmable Logic Data Book, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging</td>
</tr>
<tr>
<td></td>
<td><a href="http://support.xilinx.com/partinfo/databook.htm">http://support.xilinx.com/partinfo/databook.htm</a></td>
</tr>
<tr>
<td>Problem Solvers</td>
<td>Interactive tools that allow you to troubleshoot your design issues</td>
</tr>
<tr>
<td></td>
<td><a href="http://support.xilinx.com/support/troubleshoot/psolvers.htm">http://support.xilinx.com/support/troubleshoot/psolvers.htm</a></td>
</tr>
</tbody>
</table>
Conventions

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

**Typographical**

The following typographical conventions are used in this document:

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

The following conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning or Use</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical ellipsis . . .</td>
<td>Repetitive material that has been omitted</td>
<td>IOB #1: Name = QOUT’ IOB #2: Name = CLKin’ . . .</td>
</tr>
<tr>
<td>Horizontal ellipsis ...</td>
<td>Repetitive material that has been omitted</td>
<td>allow block block_name loc1 loc2 ... locn;</td>
</tr>
</tbody>
</table>

Online Document

The following conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning or Use</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue text</td>
<td>Cross-reference link to a location in the current file or in another file in the current document</td>
<td>See the section “Additional Resources” for details. Refer to “Title Formats” in Chapter 1 for details.</td>
</tr>
<tr>
<td>Red text</td>
<td>Cross-reference link to a location in another document</td>
<td>See Figure 2-5 in the Virtex-II Handbook.</td>
</tr>
<tr>
<td>Blue, underlined text</td>
<td>Hyperlink to a website (URL)</td>
<td>Go to <a href="http://www.xilinx.com">http://www.xilinx.com</a> for the latest speed files.</td>
</tr>
</tbody>
</table>
Chapter 1

MicroBlaze Architecture

Summary

This document describes the architecture for the MicroBlaze™ 32-bit soft processor core.

Overview

The MicroBlaze embedded soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx field programmable gate arrays (FPGAs). See Figure 1-1 for a block diagram depicting the MicroBlaze core.

Features

The MicroBlaze embedded soft core includes the following features:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- Separate 32-bit instruction and data buses that conform to IBM's OPB (On-chip Peripheral Bus) specification
- Separate 32-bit instruction and data buses with direct connection to on-chip block RAM through a LMB (Local Memory Bus)
- 32-bit address bus
- Single issue pipeline
- Instruction and data cache
- Hardware debug logic
- FSL (Fast Simplex Link) support
- Hardware multiplier (in Virtex-II and subsequent devices)
Chapter 1: MicroBlaze Architecture

Instructions

All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit immediate operand (which can be extended to 32 bits by preceding the Type B instruction with an IMM instruction). Type B instructions have a single destination register operand. Instructions are provided in the following functional categories: arithmetic, logical, branch, load/store, and special.

Table 1-2 lists the MicroBlaze instruction set. Refer to the MicroBlaze Instruction Set Architecture document for more information on these instructions. Table 1-1 describes the instruction set nomenclature used in the semantics of each instruction.

Table 1-1: Instruction Set Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>R0 - R31, General Purpose Register, source operand a</td>
</tr>
<tr>
<td>Rb</td>
<td>R0 - R31, General Purpose Register, source operand b</td>
</tr>
<tr>
<td>Rd</td>
<td>R0 - R31, General Purpose Register, destination operand,</td>
</tr>
<tr>
<td>C</td>
<td>Carry flag, MSR[29]</td>
</tr>
<tr>
<td>Sa</td>
<td>Special Purpose Register, source operand</td>
</tr>
<tr>
<td>Sd</td>
<td>Special Purpose Register, destination operand</td>
</tr>
<tr>
<td>s(x)</td>
<td>Sign extend argument x to 32-bit value</td>
</tr>
<tr>
<td>*Addr</td>
<td>Memory contents at location Addr (data-size aligned)</td>
</tr>
</tbody>
</table>
### Table 1-2: MicroBlaze Instruction Set Summary

<table>
<thead>
<tr>
<th>Type A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rd,Ra,Rb</td>
<td>000000</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Ra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUB Rd,Ra,Rb</td>
<td>000001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Rb + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDC Rd,Ra,Rb</td>
<td>000010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Rb + C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBC Rd,Ra,Rb</td>
<td>000011</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Rb + C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDK Rd,Ra,Rb</td>
<td>000100</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Ra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBK Rd,Ra,Rb</td>
<td>000101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Rb + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDKC Rd,Ra,Rb</td>
<td>000110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Rb + C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBKC Rd,Ra,Rb</td>
<td>000111</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb + Rb + C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP Rd,Ra,Rb</td>
<td>000101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb cmp Ra (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPU Rd,Ra,Rb</td>
<td>000101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb cmp Ra (unsigned)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI Rd,Ra,Imm</td>
<td>001000</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBI Rd,Ra,Imm</td>
<td>001001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDIC Rd,Ra,Imm</td>
<td>001010</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra + C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBIC Rd,Ra,Imm</td>
<td>001011</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra + C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDIK Rd,Ra,Imm</td>
<td>001100</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBIK Rd,Ra,Imm</td>
<td>001101</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDIKC Rd,Ra,Imm</td>
<td>001110</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra + C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSUBIKC Rd,Ra,Imm</td>
<td>001111</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := s(Imm) + Ra + C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL Rd,Ra,Rb</td>
<td>010000</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Ra * Rb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSRL Rd,Ra,Rb</td>
<td>010001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Ra &gt;&gt; Rb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSRA Rd,Ra,Rb</td>
<td>010001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>010000000000</td>
<td>Rd := Ra[0], (Ra &gt;&gt; Rb)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSLL Rd,Ra,Rb</td>
<td>010001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>100000000000</td>
<td>Rd := Ra &lt;&lt; Rb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULI Rd,Ra,Imm</td>
<td>011000</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra * s(Imm)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSRLI Rd,Ra,Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>0000</td>
<td>0000.. Imm</td>
<td>Rd := Ra &gt;&gt; Imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSRAI Rd,Ra,Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>0000</td>
<td>0100.. Imm</td>
<td>Rd := Ra[0], (Ra &gt;&gt; Imm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSLLI Rd,Ra,Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>0000</td>
<td>1000.. Imm</td>
<td>Rd := Ra &lt;&lt; Imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDIV Rd,Ra,Rb</td>
<td>010010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb/Ra, signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDIVU Rd,Ra,Rb</td>
<td>010010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Rd := Rb/Ra, unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>0000</td>
<td>0000</td>
<td>FSLx</td>
<td>Rd := FSLx (blocking data read)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUT Ra,FSLx</td>
<td>011011</td>
<td>0000</td>
<td>Ra</td>
<td>1000</td>
<td>FSLx</td>
<td>FSLx := Ra (blocking data write)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nGET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>0000</td>
<td>0100</td>
<td>FSLx</td>
<td>FSLx := Ra (non-blocking data read)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nPUT Ra,FSLx</td>
<td>011011</td>
<td>0000</td>
<td>Ra</td>
<td>1100</td>
<td>FSLx</td>
<td>FSLx := Ra (non-blocking data write)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 1-2: MicroBlaze Instruction Set Summary (Continued)

<table>
<thead>
<tr>
<th>Type A</th>
<th>0-5</th>
<th>6-10</th>
<th>11-15</th>
<th>16-20</th>
<th>21-31</th>
<th>Type B</th>
<th>0-5</th>
<th>6-10</th>
<th>11-15</th>
<th>16-31</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>cGET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000</td>
<td>0010</td>
<td>FSLx</td>
<td>Rd := FSLx (blocking control read)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cPUT Ra,FSLx</td>
<td>011011</td>
<td>00000</td>
<td>Ra</td>
<td>1010</td>
<td>FSLx</td>
<td>FSLx := Ra (blocking control write)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ncGET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000</td>
<td>0110</td>
<td>FSLx</td>
<td>Rd := FSLx (non-blocking control read)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ncPUT Ra,FSLx</td>
<td>011011</td>
<td>00000</td>
<td>Ra</td>
<td>1110</td>
<td>FSLx</td>
<td>FSLx := Ra (non-blocking control write)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR Rd,Ra,Rb</td>
<td>100000</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>Rd := Ra or Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND Rd,Ra,Rb</td>
<td>100001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>Rd := Ra and Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR Rd,Ra,Rb</td>
<td>100010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>Rd := Ra xor Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANDN Rd,Ra,Rb</td>
<td>100011</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>Rd := Ra and Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRA Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
<td>0000000000000000</td>
<td>Rd := Ra[0], (Ra &gt;&gt; 1); C := Ra[31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
<td>0000000000000000</td>
<td>Rd := C, (Ra &gt;&gt; 1); C := Ra[31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
<td>0000000000000000</td>
<td>Rd := 0, (Ra &gt;&gt; 1); C := Ra[31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEXT16 Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
<td>0000000000000000</td>
<td>Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIC Rd,Ra</td>
<td>100100</td>
<td>Ra</td>
<td>Ra</td>
<td>Rb</td>
<td>01101000</td>
<td>ICache_Tag := Ra, ICache_Data := Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC Rd,Ra</td>
<td>100100</td>
<td>Ra</td>
<td>Ra</td>
<td>Rb</td>
<td>01101000</td>
<td>DCache_Tag := Ra, DCache_Data := Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTS Sd,Ra</td>
<td>100101</td>
<td>00000</td>
<td>Ra</td>
<td>1100000000000000</td>
<td>Sd := Ra , where S1 is MSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MFS Rd,Sa</td>
<td>100101</td>
<td>Rd</td>
<td>00000</td>
<td>1000000000000000</td>
<td>Rd := Sa , where S0 is PC and S1 is MSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR Rd</td>
<td>100110</td>
<td>00000</td>
<td>00000</td>
<td>0000000000000000</td>
<td>PC := PC + Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRD Rb</td>
<td>100110</td>
<td>00000</td>
<td>10000</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRLD Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>10100</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>PC := PC + Rb; Rd := PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRA Rb</td>
<td>100110</td>
<td>00000</td>
<td>01000</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>PC := Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRAD Rb</td>
<td>100110</td>
<td>00000</td>
<td>11000</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>PC := Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRALD Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>11100</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>PC := Rb; Rd := PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRK Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>01100</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>PC := Rb; Rd := PC; MSR[BIP] := 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ Ra,Rb</td>
<td>100111</td>
<td>00000</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE Ra,Rb</td>
<td>100111</td>
<td>00001</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra /= 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLT Ra,Rb</td>
<td>100111</td>
<td>00010</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra &lt; 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLE Ra,Rb</td>
<td>100111</td>
<td>00011</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra &lt;= 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGT Ra,Rb</td>
<td>100111</td>
<td>00100</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra &gt; 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGE Ra,Rb</td>
<td>100111</td>
<td>00101</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra &gt;= 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQD Ra,Rb</td>
<td>100111</td>
<td>10000</td>
<td>Ra</td>
<td>Rb</td>
<td>0000000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 1-2: MicroBlaze Instruction Set Summary (Continued)

<table>
<thead>
<tr>
<th>Type A</th>
<th>0-5</th>
<th>6-10</th>
<th>11-15</th>
<th>16-20</th>
<th>21-31</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNED Ra,Rb</td>
<td>100111</td>
<td>10001</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra /= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLTD Ra,Rb</td>
<td>100111</td>
<td>10010</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &lt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLED Ra,Rb</td>
<td>100111</td>
<td>10011</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &lt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGTD Ra,Rb</td>
<td>100111</td>
<td>10100</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &gt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGED Ra,Rb</td>
<td>100111</td>
<td>10101</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &gt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>ORI Rd,Ra,Imm</td>
<td>101000</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra or s(Imm)</td>
<td></td>
</tr>
<tr>
<td>ANDI Rd,Ra,Imm</td>
<td>101001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra and s(Imm)</td>
<td></td>
</tr>
<tr>
<td>XORI Rd,Ra,Imm</td>
<td>101010</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra xor s(Imm)</td>
<td></td>
</tr>
<tr>
<td>ANDNI Rd,Ra,Imm</td>
<td>101011</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra and s(Imm)</td>
<td></td>
</tr>
<tr>
<td>IMM Imm</td>
<td>101100</td>
<td>00000</td>
<td>00000</td>
<td>Imm</td>
<td>Imm[0:15] := Imm</td>
<td></td>
</tr>
<tr>
<td>RTS Ra,Imm</td>
<td>101101</td>
<td>10000</td>
<td>Ra</td>
<td>Imm</td>
<td>PC := Ra + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>RTID Ra,Imm</td>
<td>101101</td>
<td>10001</td>
<td>Ra</td>
<td>Imm</td>
<td>PC := Ra + s(Imm); MSR[IE] := 1</td>
<td></td>
</tr>
<tr>
<td>RTBD Ra,Imm</td>
<td>101101</td>
<td>10010</td>
<td>Ra</td>
<td>Imm</td>
<td>PC := Ra + s(Imm); MSR[BIP] := 0</td>
<td></td>
</tr>
<tr>
<td>BRID Imm</td>
<td>101110</td>
<td>00000</td>
<td>10000</td>
<td>Imm</td>
<td>PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BRLD Rd,Imm</td>
<td>101110</td>
<td>Rd</td>
<td>10100</td>
<td>Imm</td>
<td>PC := PC + s(Imm); Rd := PC</td>
<td></td>
</tr>
<tr>
<td>BRAI Imm</td>
<td>101110</td>
<td>00000</td>
<td>01000</td>
<td>Imm</td>
<td>PC := s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BRAID Imm</td>
<td>101110</td>
<td>00000</td>
<td>11000</td>
<td>Imm</td>
<td>PC := s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BRALID Rd,Imm</td>
<td>101110</td>
<td>Rd</td>
<td>11100</td>
<td>Imm</td>
<td>PC := s(Imm); Rd := PC</td>
<td></td>
</tr>
<tr>
<td>BRKI Rd,Imm</td>
<td>101110</td>
<td>Rd</td>
<td>01100</td>
<td>Imm</td>
<td>PC := s(Imm); Rd := PC; MSR[BIP] := 1</td>
<td></td>
</tr>
<tr>
<td>BEQI Ra,Imm</td>
<td>101111</td>
<td>00000</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra = 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BNEI Ra,Imm</td>
<td>101111</td>
<td>00001</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra /= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BLTI Ra,Imm</td>
<td>101111</td>
<td>00010</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &lt; 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BLEI Ra,Imm</td>
<td>101111</td>
<td>00011</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &lt;= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BGTI Ra,Imm</td>
<td>101111</td>
<td>00100</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &gt; 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BGEI Ra,Imm</td>
<td>101111</td>
<td>00101</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &gt;= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BEQID Ra,Imm</td>
<td>101111</td>
<td>10000</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra = 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BNEID Ra,Imm</td>
<td>101111</td>
<td>10001</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra /= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BLTID Ra,Imm</td>
<td>101111</td>
<td>10010</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &lt; 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BLEID Ra,Imm</td>
<td>101111</td>
<td>10011</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &lt;= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BGTD Ra,Imm</td>
<td>101111</td>
<td>10100</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &gt; 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BGEID Ra,Imm</td>
<td>101111</td>
<td>10101</td>
<td>Ra</td>
<td>Imm</td>
<td>if Ra &gt;= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>LBU Rd,Ra,Rb</td>
<td>110000</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>Addr := Ra + Rb; Rd[0:23] := 0, Rd[24:31] := *Addr</td>
</tr>
</tbody>
</table>
Registers

MicroBlaze is a fully orthogonal architecture. It has thirty-two 32-bit general purpose registers and two 32-bit special purpose registers.

### General Purpose Registers

The thirty-two 32-bit General Purpose Registers are numbered R0 through R31. R0 is defined to always have the value of zero. Anything written to R0 is discarded, and zero is always read.
Special Purpose Registers

Program Counter (PC)

The Program Counter is the 32-bit address of the next instruction word to be fetched. It can be read by accessing RPC with an MFS instruction. It cannot be written to using an MTS instruction.

Table 1-3: General Purpose Registers (R0-R31)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
</table>
| 0:31   | R0 through R31 | General Purpose Register  
|        |          | R0 through R31 are 32-bit general purpose registers. R0 is always zero. | 0x00000000   |

Machine Status Register (MSR)

The Machine Status Register contains the carry flag and enables for interrupts, buslock, cache and FSL error. It can be read by accessing RMSR with an MFS instruction. When reading the MSR, bit 29 is replicated in bit 0 as the carry copy. MSR can be written to with an MTS instruction. Writes to MSR are delayed one clock cycle. When writing to MSR using MTS, the value written takes effect one clock cycle after executing the MTS instruction. Any value written to bit 0 is discarded.

Table 1-4: Program Counter (PC)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
</table>
| 0:31   | PC   | Program Counter  
|        |      | Address of next instruction to fetch     | 0x00000000   |
## Machine Status Register (MSR)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
</table>
| 0    | CC   | Arithmetic Carry Copy  
Copy of the Arithmetic Carry (bit 29). Read only. | 0 |
| 1:23 | Reserved | | |
| 24   | DCE  | Data Cache Enable  
0 Data Cache is Disabled  
1 Data Cache is Enabled | 0 |
| 25   | DZ   | Division by Zero  
0 No division by zero has occurred  
1 Division by zero has occurred | 0 |
| 26   | ICE  | Instruction Cache Enable  
0 Instruction Cache is Disabled  
1 Instruction Cache is Enabled | 0 |
| 27   | FSL  | FSL Error  
0 FSL get/put had no error  
1 FSL get/put had mismatch in instruction type and value type | 0 |
| 28   | BIP  | Break in Progress  
0 No Break in Progress  
1 Break in Progress  
Source of break can be software break instruction or hardware break from Ext_Brk or Ext_NM_Brk pin. | 0 |
Pipeline

This section describes the MicroBlaze pipeline architecture.

Pipeline Architecture

The MicroBlaze pipeline is a parallel pipeline, divided into three stages:

- Fetch
- Decode
- Execute

In general, each stage takes one clock cycle to complete. Consequently, it takes three clock cycles (ignoring any delays or stalls) for the instruction to complete.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>C</td>
<td>Arithmetic Carry</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 No Carry (Borrow)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Carry (No Borrow)</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>IE</td>
<td>Interrupt Enable</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Interrupts disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Interrupts enabled</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>BE</td>
<td>Buslock Enable</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Buslock disabled on data-side OPB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Buslock enabled on data-side OPB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Buslock Enable does not affect operation of ILMB, DLMB, or IOPB.</td>
<td></td>
</tr>
</tbody>
</table>
Branches

Similar to other processor pipelines, the MicroBlaze pipeline can originate control hazards that affect the pipeline execution rate. When an instruction that changes the control flow of a program (branches) is executed and completed, and eventually changes the program flow (taken branches), the previous pipeline work becomes useless. When the processor executes a taken branch, the instructions in the fetch and decode stages are not the correct ones, and must be discarded or flushed from the pipeline. The processor must refill the pipeline with the correct instructions, taking three clock cycles for a taken branch, adding a latency of two cycles for refilling the pipeline.

MicroBlaze uses two techniques to reduce the penalty of taken branches. One technique is to use delay slots and another is use of a history buffer.

Delay Slots

When the processor executes a taken branch and flushes the pipeline, it takes three clock cycles to refill the pipeline. By allowing the instruction following a branch to complete, this penalty is reduced. Instead of flushing the instructions in both the fetch and decode stages, only the fetch stage is discarded and the instruction in the decode stage is allowed to complete. This effectively produces a delayed branch or delay slot. Since the work done on the delay slot instruction is not discarded, this technique effectively reduces the branch penalty from two clock cycles to one. Branch instructions that allow execution of the subsequent instruction in the delay slot are denoted by a D in the instruction mnemonic. For example, the BNE instruction does not execute the subsequent instruction in the delay slot, whereas BNED does execute the next instruction in the delay slot before control is transferred to the branch location.

Load/Store Architecture

MicroBlaze can access memory in the following three data sizes:

- Byte (8 bits)
- Halfword (16 bits)
- Word (32 bits)

Memory accesses are always data-size aligned. For halfword accesses, the least significant address bit is forced to 0. Similarly, for word accesses, the two least significant address bits are forced to 0.

MicroBlaze is a Big-Endian processor and uses the Big-Endian address and labeling conventions shown in Figure 1-5 when accessing memory. The following abbreviations are used:

- MSByte: Most Significant Byte
- LSByte: Least Significant Byte
- MSBit: Most Significant Bit
- LSBit: Least Significant Bit
Interrupts, Exceptions and Breaks

When a Reset or a Debug_Rst occurs, MicroBlaze starts executing from address 0. PC and MSR are reset to the default values. When an Ext_Brk occurs, MicroBlaze starts executing from address 0x18 and stores the return address in register 16. An Ext_Brk is not executed if the BIP bit in MSR is active (equal to 1). When an Ext_NM_Brk occurs, MicroBlaze starts executing from address 0x18 and stores the return address in register 16. This occurs independent of the BIP bit value in MSR.

Interrupts

When an interrupt occurs, MicroBlaze stops the current execution to handle the interrupt request. MicroBlaze branches to address 0x00000010 and uses the General Purpose Register 14 to store the address of the instruction that was to be executed when the interrupt occurred. It also disables future interrupts by clearing the Interrupt Enable flag in the Machine Status Register (setting bit 30 to 0 in MSR). The instruction located at the address where the current PC points to is not executed. Interrupts do not occur if the BIP bit in the MSR register is active (equal to 1).

Interrupts, Exceptions and Breaks

When a Reset or a Debug_Rst occurs, MicroBlaze starts executing from address 0. PC and MSR are reset to the default values. When an Ext_Brk occurs, MicroBlaze starts executing from address 0x18 and stores the return address in register 16. An Ext_Brk is not executed if the BIP bit in MSR is active (equal to 1). When an Ext_NM_Brk occurs, MicroBlaze starts executing from address 0x18 and stores the return address in register 16. This occurs independent of the BIP bit value in MSR.

Interrupts

When an interrupt occurs, MicroBlaze stops the current execution to handle the interrupt request. MicroBlaze branches to address 0x00000010 and uses the General Purpose Register 14 to store the address of the instruction that was to be executed when the interrupt occurred. It also disables future interrupts by clearing the Interrupt Enable flag in the Machine Status Register (setting bit 30 to 0 in MSR). The instruction located at the address where the current PC points to is not executed. Interrupts do not occur if the BIP bit in the MSR register is active (equal to 1).
Latency

The time it will take MicroBlaze to enter an Interrupt Service Routine (ISR) from the time an interrupt occurs, depends on the configuration of the processor. If MicroBlaze is configured to have a hardware divider, the largest latency will happen when an interrupt occurs during the execution of a division instruction.

Table 1-6 shows the different scenarios for interrupts. The cycle count includes the cycles for completing the current instruction, the cycles for to branch and the cycles to access the first instruction in the ISR.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>ISR in LMB</th>
<th>ISR in OPB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normally</td>
<td>4 cycles</td>
<td>6 cycles</td>
</tr>
<tr>
<td>Worst case without hardware divider</td>
<td>6 cycles</td>
<td>8 cycles</td>
</tr>
<tr>
<td>Worst case with hardware divider</td>
<td>38 cycles</td>
<td>40 cycles</td>
</tr>
</tbody>
</table>

Equivalent Pseudocode

\[
\begin{align*}
\text{r14} & \leftarrow \text{PC} \\
\text{PC} & \leftarrow 0x00000010 \\
\text{MSR[IE]} & \leftarrow 0
\end{align*}
\]

Exceptions

When an exception occurs, MicroBlaze stops the current execution to handle the exception. MicroBlaze branches to address 0x00000008 and uses the General Purpose Register 17 to store the address of the instruction that was to be executed when the exception occurred. The instruction located at the address where the current PC points to is not executed.

Equivalent Pseudocode

\[
\begin{align*}
\text{r17} & \leftarrow \text{PC} \\
\text{PC} & \leftarrow 0x00000008
\end{align*}
\]

Breaks

There are two kinds of breaks:

- Software (internal) breaks
- Hardware (external) breaks

Software Breaks

To perform a software break, use the brk and brki instructions. Refer to the Instruction Set Architecture documentation for more information on software breaks.

Hardware Breaks

Hardware breaks are performed by asserting the external break signal. When a hardware break occurs, MicroBlaze stops the current execution to handle the break. MicroBlaze branches to address 0x00000018 and uses the General Purpose Register 16 to store the
address of the instruction that was to be executed when the break occurred. MicroBlaze also disables future breaks by setting the Break In Progress (BIP) flag in the Machine Status Register (setting bit 28 to 1 in MSR). The instruction located at the address where the current PC points to is not executed.

Hardware breaks are only handled when there is no break in progress (the Break In Progress flag is set to 0). The Break In Progress flag has higher precedence than the Interrupt Enabled flag. While no interrupts are handled when the Break In Progress flag is set, breaks that occur when interrupts are disabled are handled immediately. However, it is important to note that non-maskable hardware breaks are always handled immediately.

Equivalent Pseudocode

\[
\begin{align*}
& r16 \leftarrow \text{PC} \\
& \text{PC} \leftarrow 0x00000018 \\
& \text{MSR}[\text{IE}] \leftarrow 1
\end{align*}
\]

Instruction Cache

Overview

MicroBlaze may be used with an optional instruction cache for improved performance when executing code that resides outside the LMB address range.

The instruction cache has the following features:

- User selectable cacheable memory area
- Configurable cache size and tag size
- Individual cache line lock capability
- Cache on and off controlled using a new bit in the MSR register
- Instructions to write to the instruction cache
- Does not require special memory controllers. Will work with existing OPB peripherals
- Memory is organized into a cacheable and a non-cacheable segment
- Very little area or frequency impact (< 20 LUTs)
- Can be used in conjunction with Instruction side LMB

Cache Organization

When the instruction cache is used, the memory address space is split into two segments - a cacheable segment and a non-cacheable segment. The cacheable segment is determined by two parameters, C_ICACHE_BASEADDR and C_ICACHE_HIGHADDR. All addresses within this range correspond to the cacheable address space segment. All other addresses are non-cacheable.
All cacheable instruction addresses are further split into two segments - a cache line segment and a tag address segment. The size of the two segments can be configured by the user. The address bits between bit 0 and the first tag address bit are ignored in the cache. The size of the cache line can be between 9 to 14 bits. This results in a cache size ranging from 4 Kbytes to 64 Kbytes. There is no limit on the tag address size.

**Cache Operation**

In the instruction fetch stage, MicroBlaze writes the instruction address to the instruction address bus and waits for a ready signal. To reduce wait states, a request is done simultaneously on the instruction OPB and the instruction LMB. If an acknowledge signal is received from the LMB in the next cycle, the instruction access from OPB is aborted. For every instruction fetched, the instruction cache detects if the instruction address belongs to the cacheable segment. If the address is non-cacheable, the cache ignores the instruction and allows the LMB or the OPB to fulfill the request. If the address is cacheable, a lookup is performed on the tag memory to check if the requested instruction is in the cache. The lookup is successful when both the valid bit is set and the tag address is the same as the tag address segment of the instruction address.
Instruction Cache

If the instruction is in the cache, the cache will drive the ready signal (Cache_Hit) for MicroBlaze and the instruction data for the address. If the instruction is not in the cache, the cache will not drive the ready signal but will wait until the OPB fulfills the request and updates the cache with the new information.

Software

MSR Bit

Bit 26 in the MSR indicates whether or not the cache is enabled. The MFS and MTS instructions are used to read and write to the MSR respectively.

The contents of the cache are preserved by default when the cache is disabled. The user may overwrite the contents of the cache using the WIC instruction or using the hardware debug logic of MicroBlaze.

WIC Instruction

The WIC instruction may be used to update the instruction cache from a software program. The assembly instruction is

\[
\text{WIC } Ra, Rb
\]

Where Ra contains cache line, tag address, valid and lock bit, Rb contains the instruction data.

Ra(31) is the lock bit, Ra(30) is the valid bit (valid when bit is set to ‘1’), the rest of the Ra contains the instruction address.

This instruction can only be used when the cache is disabled. The lock bit is described in the Lock Bit section below. The

HW Debug Logic

The HW debug logic may be used to perform a similar operation as the WIC instruction.

Figure 1-7: Cache Operation

![Cache Operation Diagram]

IOPB_Address

Tag Address

Cache Line

IOPB_Select

IOPB_XferAck

IOPB_Data

0,1 (Locked,Valid)

Tag BRAM

Data

WE

Address

Instruction BRAM

Address

WE

Data

IOPB_XferAck

0 1

Figure 1-7: Cache Operation

If the instruction is in the cache, the cache will drive the ready signal (Cache_Hit) for MicroBlaze and the instruction data for the address. If the instruction is not in the cache, the cache will not drive the ready signal but will wait until the OPB fulfills the request and updates the cache with the new information.

Software

MSR Bit

Bit 26 in the MSR indicates whether or not the cache is enabled. The MFS and MTS instructions are used to read and write to the MSR respectively.

The contents of the cache are preserved by default when the cache is disabled. The user may overwrite the contents of the cache using the WIC instruction or using the hardware debug logic of MicroBlaze.

WIC Instruction

The WIC instruction may be used to update the instruction cache from a software program. The assembly instruction is

\[
\text{WIC } Ra, Rb
\]

Where Ra contains cache line, tag address, valid and lock bit, Rb contains the instruction data.

Ra(31) is the lock bit, Ra(30) is the valid bit (valid when bit is set to ‘1’), the rest of the Ra contains the instruction address.

This instruction can only be used when the cache is disabled. The lock bit is described in the Lock Bit section below. The

HW Debug Logic

The HW debug logic may be used to perform a similar operation as the WIC instruction.
Lock Bit

The lock bit can be used to permanently lock a code segment into the cache and therefore guarantee the instruction execution time. Locking of the cacheline however may result in a decrease in the number of cache hits. This is because there could be addresses that were not cached as the cacheline is locked.

The use of instruction LMB in most cases would be a better choice for locking code segments since the wait states for accessing the LMB is the same as for cache hits.

LMB Memory

Instruction LMB memory can be used even when instruction cache is used. The LMB address in the case has to be in the non-cacheable memory segment.

Data Cache

Overview

MicroBlaze may be used with an optional data cache for improved performance when reading data that resides outside the LMB address range.

The data cache has the following features

- Write-through data cache
- User selectable cacheable memory area
- Configurable cache size and tag size
- Individual cache line lock capability
- Cache on and off controlled using a new bit in the MSR register
- Instructions to write to the data cache
- Does not require special memory controllers. Will work with existing OPB peripherals
- Memory is organized into a cacheable and a non-cacheable segments
- Very little area or frequency impact ( < 20 LUTs)
- Can be used in conjunction with Data side LMB

Cache Organization

When the data cache is used, the memory address space in split into two segments - a cacheable segment and a non-cacheable segment. The cacheable area is determined by two parameters, C_DCACHE_BASEADDR and C_DCACHE_HIGHADDR. All addresses within this range correspond to the cacheable address space segment. All other addresses are non-cacheable.
Data Cache

All cacheable data addresses are further split into two segments - a cache line segment and a tag address segment. The size of the two segments can be configured by the user. The address bits between bit 0 and the first tag address bit are ignored in the cache. The size of the cache line can be between 9 to 14 bits. This results in a cache sizes ranging from 4 Kbytes to 64 Kbytes. There is no limit on the tag address size.

### Cache Operation

When MicroBlaze executes a store instruction, the operation is performed as normal but if the address is within the cacheable address segment, the data cache is updated with the new data.

When MicroBlaze executes a load instruction, the address is first checked to see if the address is within the cacheable area and secondly if the address is in the data cache. If that case, the data is fetch from the data cache.

![Figure 1-8: Cache Organization](image-url)

![Figure 1-9: Cache Operation](image-url)
If the read data is in the cache, the cache will drive the ready signal (Cache_Hit) for MicroBlaze and the data for the address. If the read data is not in the cache, the cache will not drive the ready signal but will wait until the OPB fulfills the request.

**Software**

**MSR Bit**

Bit 24 in the MSR indicates whether or not the cache is enabled. The MFS and MTS instructions are used to read and write to the MSR respectively.

The contents of the cache are preserved by default when the cache is disabled. The user may overwrite the contents of the cache using the WDC instruction or using the hardware debug logic of MicroBlaze.

Note: The cache cannot be turned on/off from an interrupt handler routine as the changes to the MSR is lost once the interrupt is handled (the MSR state is restored after interrupt handling).

**WDC Instruction**

The WDC instruction may be used to update the data cache from a software program. The assembly instruction is

```
WDC Ra, Rb
```

Where Ra contains cache line, tag address, valid and lock bit, Rb contains the data.

Ra(31) is the lock bit, Ra(30) is the valid bit (valid when bit is set to '1'), the rest of the Ra contains the instruction address.

This instruction can only be used when the cache is disabled. The lock bit is described in the Lock Bit section below. The

**HW Debug Logic**

The HW debug logic may be used to perform a similar operation as the WDC instruction.

**Lock Bit**

The lock bit can be used to permanently lock a code segment into the cache and therefore guarantee that this data is always in the cache. Locking of the cacheline however may result in a decrease in the number of cache hits. This is because there could be addresses that were not cached as the cacheline is locked.

The use of data LMB in most cases would be a better choice for locking data since the wait states for accessing the LMB is the same as for cache hits.

**LMB Memory**

Data LMB memory can be used even when data cache is used. The LMB address in the case has to be in the non-cacheable memory segment.

**Fast Simplex Link Interface**

MicroBlaze contains eight input and eight output Fast Simplex Link (FSL) interfaces. For detailed information on the FSL interface, please refer to the FSL Bus documentation. The
FSL channels are dedicated uni-directional point-to-point data streaming interfaces. The FSL interfaces on MicroBlaze are 32 bits wide. Further, the same FSL channels can be used to transmit or receive either control or data words. A separate bit indicates whether the transmitted (received) word is control or data information.

FSL Read Instructions

The Get instructions are used for reading data or control from an input FSL channel into a MicroBlaze register. There are 4 types of get instructions.

Blocking Data Get Instruction

The assembly instruction to perform a blocking get is

\[
get \ regM, \ fslN
\]

The blocking get instruction stalls the MicroBlaze pipeline until data becomes available in the input FSL, fslN. Once the data is available, the instruction is completed in two clock cycles. The get instruction is used for getting Data values. If a get instruction is used to read a Control value (the control_in bit of the fslN is set), a FSL get error bit is set in the MSR (Bit 27).

Non-blocking Data Get Instruction

The assembly instruction to perform a non-blocking get is

\[
nget \ regM, \ fslN
\]

The non-blocking get instruction does not stall the MicroBlaze pipeline whether or not data is present on the input FSL, fslN. The instruction is completed in two clock cycles. If the data is available, the carry bit (Bit 29)in the MSR is set. If the instruction fails the carry bit in the MSR is reset. Bit 0 of the MSR has the copy of the carry bit. Hence, a direct branch on carry may be performed following the nget instruction. The nget instruction is also used to read Data values. If a Control value is read, the FSL error bit (Bit 27 of MSR) is set.

Blocking Control Get Instruction

The assembly instruction to perform a blocking control get is

\[
cget \ regM, \ fslN
\]

The blocking control get instruction stalls the MicroBlaze pipeline until data becomes available in the input FSL, fslN. Once the data is available, the instruction is completed in two clock cycles. The cget instruction is used for reading Control values (the control_in bit of the fslN is set). If the value read is a data value, the FSL error bit (Bit 27 of MSR) is set.

Non-blocking Control Get Instruction

The assembly instruction to perform a non-blocking get is

\[
ncget \ regM, \ fslN
\]

The non-blocking control get instruction does not stall the MicroBlaze pipeline whether or not data is present on the input FSL, fslN. The instruction is completed in two clock cycles. If the data is available, the carry bit (Bit 29) in the MSR is set. If the instruction fails the carry bit in the MSR is reset. Bit 0 of the MSR has the copy of the carry bit. Hence, a direct branch on carry may be performed following the ncget instruction. The ncget instruction is also used to read Control values (the control_in bit of the fslN is set). If the value read is a data value, the FSL error bit (Bit 27 of the MSR) is set.
FSL Write Instructions

The Put instructions are used for writing data or control to an output FSL channel into a MicroBlaze register. There are 4 types of put instructions.

Blocking Data Put Instruction

The assembly instruction to perform a blocking put is

```
put regM, fslN
```

The blocking put instruction stalls the MicroBlaze pipeline until a data can be written to the output FSL, fslN (data can be written when the full bit is not set). Once the data can be written, the instruction is completed in two clock cycles. The put instruction is used for writing Data values (the control_out bit of the fslN is reset).

Non-blocking Data Put Instruction

The assembly instruction to perform a non-blocking put is

```
nput regM, fslN
```

The non-blocking put instruction does not stall the MicroBlaze pipeline whether or not data can be written to the output FSL, fslN (data can be written when the full bit is not set). The instruction is completed in two clock cycles. If the data write succeeds, the carry bit (Bit 29) in the MSR is set. If the data write fails, the carry bit in the MSR is reset. Bit 0 of the MSR has the copy of the carry bit. Hence, a direct branch on carry may be performed following the nput instruction. The nput instruction is also used to write Data values (the control_out bit of fslN is reset).

Blocking Control Put Instruction

The assembly instruction to perform a blocking control put is

```
cput regM, fslN
```

The blocking put instruction stalls the MicroBlaze pipeline until a data can be written to the output FSL, fslN (data can be written when the full bit is not set). Once the data can be written, the instruction is completed in two clock cycles. The put instruction is used for writing Control values (the control_out bit of the fslN is set).

Non-blocking Data Put Instruction

The assembly instruction to perform a non-blocking control put is

```
nccput regM, fslN
```

The non-blocking put instruction does not stall the MicroBlaze pipeline whether or not data can be written to the output FSL, fslN (data can be written when the full bit is not set). The instruction is completed in two clock cycles. If the data write succeeds, the carry bit (Bit 29) in the MSR is set. If the data write fails, the carry bit in the MSR is reset. Bit 0 of the MSR has the copy of the carry bit. Hence, a direct branch on carry may be performed following the nput instruction. The nput instruction is also used to write Control values (the control_out bit of fslN is set).

Debug Interface

MicroBlaze features a debug interface to support JTAG based software debugging tools (commonly known as BDM or Background Debug Mode debuggers) like the Xilinx.
Microprocessor Debug (XMD) tool. The debug interface is designed to be connected to the Xilinx Microprocessor Debug Module (MDM) IP core, which interfaces with the JTAG port of Xilinx FPGAs. Multiple MicroBlazes can be interfaced with a single MDM to enable multiprocessor debugging.

Debugging Features

- Configurable number of hardware breakpoints and watchpoints and unlimited software breakpoints
- External processor control enables debug tools to stop, reset and single step MicroBlaze
- Read and write memory and all registers including PC and MSR
- Support for multiple processors
- Write to Instruction and data cache
Chapter 2

MicroBlaze Bus Interfaces

Summary

This document describes the MicroBlaze™ Local Memory Bus (LMB) and On-chip Peripheral Bus (OPB) interfaces.

Overview

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. Each bus interface unit is further split into a Local Memory Bus (LMB) and IBM’s On-chip Peripheral Bus (OPB). The LMB provides single-cycle access to on-chip dual-port block RAM. The OPB interface provides a connection to both on-and off-chip peripherals and memory. Further, the MicroBlaze core provides 8 input and 8 output interfaces to Fast Simplex Link (FSL) buses. The FSL buses are uni-directional non-arbitrated dedicated communication channels.

Features

The MicroBlaze bus interfaces include the following features:

- OPB V2.0 bus interface with byte-enable support (see IBM’s 64-Bit On-Chip Peripheral Bus, Architectural Specifications, Version 2.0)
- LMB provides simple synchronous protocol for efficient block RAM transfers
- LMB provides guaranteed performance of 125 MHz for local memory subsystem
- FSL provides a fast non-arbitrated streaming communication mechanism.

Bus Configurations

The block diagram in Figure 2-1 depicts the MicroBlaze core with the bus interfaces defined as follows:

- DOPB: Data interface, On-chip Peripheral Bus
- DLMB: Data interface, Local Memory Bus (BRAM only)
- IOPB: Instruction interface, On-chip Peripheral Bus
- ILMB: Instruction interface, Local Memory Bus (BRAM only)
- MFSL0..MFSL7: Master data interface, Fast Simplex Link
- SFSL0..SFSL7: Slave data interface, Fast Simplex Link
- Core: Miscellaneous signals (Clock, Reset, Interrupt)
Chapter 2: MicroBlaze Bus Interfaces

MicroBlaze bus interfaces are available in six configurations, as shown in the following figure.

NOTE: All the 6 configurations can be used along with the FSL bus configuration.

Figure 2-1: MicroBlaze Core Block Diagram

Figure 2-2: MicroBlaze Bus Configurations
The optimal configuration for your application depends on code size and data spaces, and if you require fast access to internal block RAM. The performance implications and supported memory models for each configuration is shown in the following table:

### Table 2-1: MicroBlaze Bus Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Core Fmax</th>
<th>Debug available</th>
<th>Memory Models Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 IOPB+ILMB+DOPB+DLMB</td>
<td>110</td>
<td>SW/JTAG</td>
<td>Large external instruction memory, Fast internal instruction memory (BRAM), Large external data memory, Fast internal data memory (BRAM)</td>
</tr>
<tr>
<td>2 IOPB+DOPB+DLMB</td>
<td>125</td>
<td>SW/JTAG</td>
<td>Large external instruction memory, Large external data memory, Fast internal data memory (BRAM)</td>
</tr>
<tr>
<td>3 ILMB+DOPB+DLMB</td>
<td>125</td>
<td>SW/JTAG</td>
<td>Fast internal instruction memory (BRAM), Large external data memory, Fast internal data memory (BRAM)</td>
</tr>
<tr>
<td>4 IOPB+ILMB+DOPB</td>
<td>110</td>
<td>JTAG for ILMB memory¹ SW/for IOPB memory</td>
<td>Large external instruction memory, Fast internal instruction memory (BRAM), Large external data memory,</td>
</tr>
<tr>
<td>5 IOPB+DOPB</td>
<td>125</td>
<td>SW/JTAG</td>
<td>Large external instruction memory, Large external data memory,</td>
</tr>
<tr>
<td>6 ILMB+DOPB</td>
<td>125</td>
<td>JTAG¹</td>
<td>Fast internal instruction memory (BRAM), Large external data memory,</td>
</tr>
</tbody>
</table>

**Note:** ILMB memory can be debugged via a software resident monitor if the second port of the dual-ported ILMB BRAM is connected to an OPB BRAM memory controller. See Figure 2-6 and Figure 2-8. Also, all the above 6 configurations can be used with a special FSL configuration. See Figure 2-9.

### Typical Peripheral Placement

This section provides typical peripheral placement and usage for each of the six configurations and the FSL configuration. Because there are many options for interconnecting a MicroBlaze system, you should use the following examples as guidelines for selecting a configuration closest to your application.
Configuration 1

Purpose

Use this configuration when your application requires more instruction and data memory than is available in the on-chip block RAM (BRAM). Critical sections of instruction and data memory can be allocated to the faster ILMB BRAM to improve your application’s performance. Depending on how much data memory is required, the data-side memory controller may not be present. The data-side OPB is also used for other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes.

Characteristics

Because of the extra logic required to implement two buses per side, the maximum clock rate of the CPU may be slightly less than configurations with one bus per side. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.
Configuration 2

Purpose
Use this configuration when your application requires more instruction and data memory than is available in the on-chip BRAM. In this configuration, all of the instruction memory is resident in off-chip memory or on-chip memory on the instruction-side OPB. Depending on how much data memory is required, the data-side memory controller may not be present. The data-side OPB is also used for other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications
- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes.

Characteristics
This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. Instruction fetches on the OPB, however, are slower than fetches from BRAM on the LMB. Overall processor performance is lower than implementations using LMB unless a large percentage of code is run from the internal instruction history buffer. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.
Configuration 3

**Figure 2-5: Configuration 3: ILMB+DOPB+DLMB**

**Purpose**

Use this configuration when your application code fits into the on-chip BRAM, but more memory may be required for data memory. Critical sections of data memory can be allocated to the faster DLMB BRAM to improve your application’s performance. Depending on how much data memory is required, the data-side memory controller may not be present. The data-side OPB is also used for other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals.

**Typical Applications**

- Data-intensive controllers
- Small to medium state machines

**Characteristics**

This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. The instruction-side LMB provides two-cycle pipelined read access from the BRAM for an effective access rate of one instruction per clock. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.
Configuration 4

Purpose

Use this configuration when your application requires more instruction and data memory than is available in the on-chip BRAM. Critical sections of instruction memory can be allocated to the faster ILMB BRAM to improve your application’s performance. The data-side OPB is used for one or more external memory controllers and other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes

Characteristics

Because of the extra logic required to implement two buses per side, the maximum clock rate of the CPU may be slightly less than configurations with one bus per side. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging. However, software-based debugging of code in the ILMB BRAM can only be performed if a BRAM memory controller is included on the D-side OPB bus to provide write access to the LMB BRAM.
Configuration 5

Purpose

Use this configuration when your application requires external instruction and data memory. In this configuration, all of the instruction and data memory is resident in off-chip memory or on-chip memory on the OPB buses. The data-side OPB is used for one or more external memory controllers and other peripherals such as UARTs, timers, general purpose I/O, BRAM, and custom peripherals. The OPB-to-OPB bridge is only required if the data-side OPB needs access to the instruction-side OPB peripherals, such as for software-based debugging.

Typical Applications

- MPEG Decoder
- Communications Controller
- Complex state machine for process control and other embedded applications
- Set top boxes

Characteristics

This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. However, instruction fetches on the OPB are slower than fetches from BRAM on the LMB. Overall processor performance is lower than implementations using LMB unless a large percentage of code is run from the internal instruction history buffer. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging.
Configuration 6

**Purpose**
Use this configuration when your application code fits into the on-chip ILMB BRAM, but more memory may be required for data memory. The data-side OPB is used for one or more external memory controllers and other peripherals such as UARTs, timers, general purpose I/O, additional BRAM, and custom peripherals.

**Typical Applications**
- Minimal controllers
- Small to medium state machines

**Characteristics**
This configuration allows the CPU core to operate at the maximum clock rate because of the simpler instruction-side bus structure. The instruction-side LMB provides two-cycle pipelined read access from the BRAM for an effective access rate of one instruction per clock. This configuration allows debugging of application code through either software-based debugging (resident monitor debugging) or hardware-based JTAG debugging. However, software-based debugging of code in the ILMB BRAM can only be performed if a BRAM memory controller is included on the D-side OPB bus to provide write access to the LMB BRAM.

**FSL Configuration**
Along with any of the above specified configurations, MicroBlaze can optionally include up to 8 FSL input interfaces and 8 FSL output interfaces.

*Figure 2-8: Configuration 6: ILMB+DOPB*
Purpose

Use this configuration for transmitting data directly from the MicroBlaze core to other peripherals or processors without using a shared bus. MicroBlaze contains several instructions to read from the input FSLs and write to the output FSLs. The read and write each consume two clock cycles. The number of FSL's in MicroBlaze can be configured by using the C_NUM_FSL parameter.

Typical Applications

The FSLs are particularly useful for streaming data style applications. These include signal processing, image processing, DSP and Network processing applications. The FSL communication channels can also be used to interface with hardware accelerators that are implemented on the reconfigurable fabric.

Characteristics

The CPU clock frequency is unaffected by the addition of FSLs to the MicroBlaze core. The area of the MicroBlaze core increases slightly based on the number of FSL interfaces.
Bit and Byte Labeling

The MicroBlaze buses are labeled using a big-endian naming convention. The bit and byte labeling for the MicroBlaze data types is shown in the following figure:

![MicroBlaze Big-Endian Data Types](image)

**Core I/O**

The MicroBlaze core implements separate buses for instruction fetch and data access, denoted the I side and D side buses, respectively. These buses are split into the following two bus types:

- OPB V2.0 compliant bus for OPB peripherals and memory controllers
- Local Memory Bus used exclusively for high-speed access to internal block RAM (BRAM).

All core I/O signals are listed in Table 2-2. Page numbers prefaced by OPB reference IBM’s 64-Bit On-Chip Peripheral Bus, Architectural Specifications, Version 2.0.

The core interfaces shown in the following table are defined as follows:
### Chapter 2: MicroBlaze Bus Interfaces

DOPB: Data interface, On-chip Peripheral Bus  
DLMB: Data interface, Local Memory Bus (BRAM only)  
IOPB: Instruction interface, On-chip Peripheral Bus  
ILMB: Instruction interface, Local Memory Bus (BRAM only)  
MFSL0..MFSL7: FSL master interface  
SFSL0..SFSL7: FSL slave interface  
Core: Miscellaneous signals

#### Table 2-2: Summary of MicroBlaze Core I/O

<table>
<thead>
<tr>
<th>Signal</th>
<th>Interface</th>
<th>I/O</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM_ABus[0:31]</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB address bus</td>
<td>OPB-11</td>
</tr>
<tr>
<td>DM_BE[0:3]</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB byte enables</td>
<td>OPB-16</td>
</tr>
<tr>
<td>DM_busLock</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB buslock</td>
<td>OPB-9</td>
</tr>
<tr>
<td>DM_DBus[0:31]</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB write data bus</td>
<td>OPB-13</td>
</tr>
<tr>
<td>DM_request</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB bus request</td>
<td>OPB-8</td>
</tr>
<tr>
<td>DM_RNW</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB read, not write</td>
<td>OPB-12</td>
</tr>
<tr>
<td>DM_select</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB select</td>
<td>OPB-12</td>
</tr>
<tr>
<td>DM_seqAddr</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB sequential address</td>
<td>OPB-13</td>
</tr>
<tr>
<td>DOPB_DBus[0:31]</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB read data bus</td>
<td>OPB-13</td>
</tr>
<tr>
<td>DOPB_errAck</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB error acknowledge</td>
<td>OPB-15</td>
</tr>
<tr>
<td>DOPB_MGrant</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB bus grant</td>
<td>OPB-9</td>
</tr>
<tr>
<td>DOPB_retry</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB bus cycle retry</td>
<td>OPB-10</td>
</tr>
<tr>
<td>DOPB_timeout</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB timeout error</td>
<td>OPB-10</td>
</tr>
<tr>
<td>DOPB_xferAck</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB transfer acknowledge</td>
<td>OPB-14</td>
</tr>
<tr>
<td>IM_ABus[0:31]</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB address bus</td>
<td>OPB-11</td>
</tr>
<tr>
<td>IM_BE[0:3]</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB byte enables</td>
<td>OPB-16</td>
</tr>
<tr>
<td>IM_busLock</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB buslock</td>
<td>OPB-9</td>
</tr>
<tr>
<td>IM_DBus[0:31]</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB write data bus (always 0x00000000)</td>
<td>OPB-13</td>
</tr>
<tr>
<td>IM_request</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB bus request</td>
<td>OPB-8</td>
</tr>
<tr>
<td>IM_RNW</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB read, not write (tied to '0')</td>
<td>OPB-12</td>
</tr>
<tr>
<td>IM_select</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB select</td>
<td>OPB-12</td>
</tr>
<tr>
<td>IM_seqAddr</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB sequential address</td>
<td>OPB-13</td>
</tr>
<tr>
<td>IOPB_DBus[0:31]</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB read data bus</td>
<td>OPB-13</td>
</tr>
<tr>
<td>IOPB_errAck</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB error acknowledge</td>
<td>OPB-15</td>
</tr>
<tr>
<td>IOPB_MGrant</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB bus grant</td>
<td>OPB-9</td>
</tr>
<tr>
<td>IOPB_retry</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB bus cycle retry</td>
<td>OPB-10</td>
</tr>
<tr>
<td>IOPB_timeout</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB timeout error</td>
<td>OPB-10</td>
</tr>
</tbody>
</table>
Table 2-2: Summary of MicroBlaze Core I/O (Continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Interface</th>
<th>I/O</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOPB_xferAck</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB transfer acknowledge</td>
<td>OPB-12</td>
</tr>
<tr>
<td>Data.Addr[0:31]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB address bus</td>
<td>49</td>
</tr>
<tr>
<td>Byte_Enable[0:3]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB byte enables</td>
<td>49</td>
</tr>
<tr>
<td>Data_Write[0:31]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB write data bus</td>
<td>50</td>
</tr>
<tr>
<td>D_AS</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB address strobe</td>
<td>50</td>
</tr>
<tr>
<td>Read_Strobe</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB read strobe</td>
<td>50</td>
</tr>
<tr>
<td>Write_Strobe</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB write strobe</td>
<td>50</td>
</tr>
<tr>
<td>Data_Read[0:31]</td>
<td>DLMB</td>
<td>I</td>
<td>Data interface LB read data bus</td>
<td>50</td>
</tr>
<tr>
<td>DReady</td>
<td>DLMB</td>
<td>I</td>
<td>Data interface LB data ready</td>
<td>50</td>
</tr>
<tr>
<td>Instr.Addr[0:31]</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LB address bus</td>
<td>49</td>
</tr>
<tr>
<td>I_AS</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LB address strobe</td>
<td>50</td>
</tr>
<tr>
<td>IFetch</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LB instruction fetch</td>
<td>50</td>
</tr>
<tr>
<td>Instr[0:31]</td>
<td>ILMB</td>
<td>I</td>
<td>Instruction interface LB read data bus</td>
<td>50</td>
</tr>
<tr>
<td>IReady</td>
<td>ILMB</td>
<td>I</td>
<td>Instruction interface LB data ready</td>
<td>50</td>
</tr>
<tr>
<td>FSL0_M .. FSL7_M</td>
<td>MFSL</td>
<td>O</td>
<td>Master interface to Output FSL channels</td>
<td></td>
</tr>
<tr>
<td>FSL0_S .. FSL7_S</td>
<td>SFSL</td>
<td>I</td>
<td>Slave interface to Input FSL channels</td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td>Core</td>
<td>I</td>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td>Core</td>
<td>I</td>
<td>Core reset</td>
<td></td>
</tr>
<tr>
<td>Clk</td>
<td>Core</td>
<td>I</td>
<td>Clock</td>
<td></td>
</tr>
<tr>
<td>Debug_Rst</td>
<td>Core</td>
<td>I</td>
<td>Reset signal from OPB JTAG UART</td>
<td></td>
</tr>
<tr>
<td>Ext_BRK</td>
<td>Core</td>
<td>I</td>
<td>Break signal from OPB JTAG UART</td>
<td></td>
</tr>
<tr>
<td>Ext_NM_BRK</td>
<td>Core</td>
<td>I</td>
<td>Non-maskable break signal from OPB JTAG UART</td>
<td></td>
</tr>
<tr>
<td>Dbg_...</td>
<td>Core</td>
<td>IO</td>
<td>Debug signals from OPB MDM</td>
<td></td>
</tr>
</tbody>
</table>

Bus Organization

**OPB Bus Configuration**

The MicroBlaze OPB interfaces are organized as byte-enable capable only masters. The byte-enable architecture is an optional subset of the OPB V2.0 specification and is ideal for low-overhead FPGA implementations such as MicroBlaze.

The OPB data bus interconnects are illustrated in Figure 2-11. The write data bus (from masters and bridges) is separated from the read data bus (from slaves and bridges) to break up the bus OR logic. In minimal cases this can completely eliminate the OR logic for the read or write data buses. Optionally, you can "OR" together the read and write buses to create the correct functionality for the OPB bus monitor. Note that the instruction-side OPB contains a write data bus (tied to 0x00000000) and a RNW signal (tied to logic 1) so that its
interface remains consistent with the data-side OPB. These signals are constant and generally are minimized in implementation.

A multi-ported slave is used instead of a bridge in the example shown in Figure 2-12. This could represent a memory controller with a connection to both the IOPB and the DOPB. In this case, the bus multiplexing and prioritization must be done in the slave. The advantage of this approach is that a separate I-to-D bridge and an OPB arbiter on the instruction side are not required. The arbiter function must still exist in the slave device.
Figure 2-11: OPB Interconnection (breaking up read and write buses)
Figure 2-12: OPB Interconnection (with multi-ported slave and no bridge)
LMB Bus Definition

The Local Memory Bus (LMB) is a synchronous bus used primarily to access on-chip block RAM. It uses a minimum number of control signals and a simple protocol to ensure that local block RAM is accessed in a single clock cycle. LMB signals and definitions are shown in the following table. All LMB signals are high true.

Table 2-3: LMB Bus Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Data Interface</th>
<th>Instr. Interface</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr[0:31]</td>
<td>Data_Addr[0:31]</td>
<td>Instr_Addr[0:31]</td>
<td>O</td>
<td>Address bus</td>
</tr>
<tr>
<td>Byte_Enable[0:3]</td>
<td>Byte_Enable[0:3]</td>
<td>not used</td>
<td>O</td>
<td>Byte enables</td>
</tr>
<tr>
<td>Data_Write[0:31]</td>
<td>Data_Write[0:31]</td>
<td>not used</td>
<td>O</td>
<td>Write data bus</td>
</tr>
<tr>
<td>AS</td>
<td>D_AS</td>
<td>I_AS</td>
<td>O</td>
<td>Address strobe</td>
</tr>
<tr>
<td>Read_Strobe</td>
<td>Read_Strobe</td>
<td>IFetch</td>
<td>O</td>
<td>Read in progress</td>
</tr>
<tr>
<td>Write_Strobe</td>
<td>Write_Strobe</td>
<td>not used</td>
<td>O</td>
<td>Write in progress</td>
</tr>
<tr>
<td>Data_Read[0:31]</td>
<td>Data_Read[0:31]</td>
<td>Instr[0:31]</td>
<td>I</td>
<td>Read data bus</td>
</tr>
<tr>
<td>Ready</td>
<td>DReady</td>
<td>IReady</td>
<td>I</td>
<td>Ready for next transfer</td>
</tr>
<tr>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>I</td>
<td>Bus clock</td>
</tr>
</tbody>
</table>

Addr[0:31]

The address bus is an output from the core and indicates the memory address that is being accessed by the current transfer. It is valid only when AS is high. In multicycle accesses (accesses requiring more than one clock cycle to complete), Addr[0:31] is valid only in the first clock cycle of the transfer.

Byte_Enable[0:3]

The byte enable signals are outputs from the core and indicate which byte lanes of the data bus contain valid data. Byte_Enable[0:3] is valid only when AS is high. In multicycle accesses (accesses requiring more than one clock cycle to complete), Byte_Enable[0:3] is valid only in the first clock cycle of the transfer. Valid values for Byte_Enable[0:3] are shown in the following table:

Table 2-4: Valid Values for Byte_Enable[0:3]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2: MicroBlaze Bus Interfaces

### Data_Write[0:31]

The write data bus is an output from the core and contains the data that is written to memory. It becomes valid when AS is high and goes invalid in the clock cycle after Ready is sampled high. Only the byte lanes specified by Byte_Enable[0:3] contain valid data.

### AS

The address strobe is an output from the core and indicates the start of a transfer and qualifies the address bus and the byte enables. It is high only in the first clock cycle of the transfer, after which it goes low and remains low until the start of the next transfer.

### Read_Strobe

The read strobe is an output from the core and indicates that a read transfer is in progress. This signal goes high in the first clock cycle of the transfer, and remains high until the clock cycle after Ready is sampled high. If a new read transfer is started in the clock cycle after Ready is high, then Read_Strobe remains high.

### Write_Strobe

The write strobe is an output from the core and indicates that a write transfer is in progress. This signal goes high in the first clock cycle of the transfer, and remains high until the clock cycle after Ready is sampled high. If a new write transfer is started in the clock cycle after Ready is high, then Write_Strobe remains high.

### Data_Read[0:31]

The read data bus is an input to the core and contains data read from memory. Data_Read[0:31] is valid on the rising edge of the clock when Ready is high.

### Ready

The Ready signal is an input to the core and indicates completion of the current transfer and that the next transfer can begin in the following clock cycle. It is sampled on the rising edge of the clock. For reads, this signal indicates the Data_Read[0:31] bus is valid, and for writes it indicates that the Data_Write[0:31] bus has been written to local memory.

### Clk

All operations on the LMB are synchronous to the MicroBlaze core clock.

### LMB Bus Operations

The following diagrams provide examples of LMB bus operations.
Bus Organization

Generic Write Operation

![Generic Write Operation Diagram]

Figure 2-13: LMB Generic Write Operation

Generic Read Operation

![Generic Read Operation Diagram]

Figure 2-14: LMB Generic Read Operation
Back-to-Back Write Operation (Typical LMB access - 2 clocks per write)

Single Cycle Back-to-Back Read Operation (Typical I-side access - 1 clock per read)
Back-to-Back Mixed Read/Write Operation (Typical D-side timing)

Figure 2-17: Back-to-Back Mixed Read/Write Operation

Read and Write Data Steering

The MicroBlaze data-side bus interface performs the read steering and write steering required to support the following transfers:

- byte, halfword, and word transfers to word devices
- byte and halfword transfers to halfword devices
- byte transfers to byte devices

MicroBlaze does not support transfers that are larger than the addressed device. These types of transfers require dynamic bus sizing and conversion cycles that are not supported by the MicroBlaze bus interface. Data steering for read cycles is shown in Table 2-5, and data steering for write cycles is shown in Table 2-6.

Table 2-5: Read Data Steering (load to Register rD)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0001</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>Byte3</td>
</tr>
<tr>
<td>10</td>
<td>0010</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>Byte2</td>
</tr>
<tr>
<td>01</td>
<td>0100</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>Byte1</td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>Byte0</td>
</tr>
<tr>
<td>10</td>
<td>0011</td>
<td>halfword</td>
<td></td>
<td></td>
<td>Byte2</td>
<td>Byte3</td>
</tr>
<tr>
<td>00</td>
<td>1100</td>
<td>halfword</td>
<td></td>
<td></td>
<td>Byte0</td>
<td>Byte1</td>
</tr>
<tr>
<td>00</td>
<td>1111</td>
<td>word</td>
<td>Byte0</td>
<td>Byte1</td>
<td>Byte2</td>
<td>Byte3</td>
</tr>
</tbody>
</table>
Note that other OPB masters may have more restrictive requirements for byte lane placement than those allowed by MicroBlaze. OPB slave devices are typically attached "left-justified" with byte devices attached to the most-significant byte lane, and halfword devices attached to the most significant halfword lane. The MicroBlaze steering logic fully supports this attachment method.

**FSL Bus Operation**

The FSLs are implemented on the FPGA as a FIFO using the SRL16 primitives. The FSL bus provides a point-to-point communication channel between an output FIFO and an input FIFO. For more information on the FSL bus, please refer to the FSL bus documentation.

**Master FSL signals on MicroBlaze**

MicroBlaze may contain up to 8 master FSL interfaces. The master signals are depicted in Table 2-7.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSLn_M_Clk</td>
<td>Clock</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_Write</td>
<td>Write signal that enables writing to FIFO when set</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_Data</td>
<td>Input Data</td>
<td>std_logic_vector</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_CONTROL</td>
<td>Control Bit indicating the FSLn_IN_DATA is a Control Word</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_FULL</td>
<td>Full Bit indicating input FIFO full when set</td>
<td>std_logic</td>
<td>input</td>
</tr>
</tbody>
</table>
Slave FSL signals on MicroBlaze

MicroBlaze may contain up to 8 slave FSL interfaces. The slave FSL interface signals are depicted in Table 2-8.

### Table 2-8: Slave FSL signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSLn_S_Clk</td>
<td>Clock</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_S_Read</td>
<td>Read signal requesting next available input to be read</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_S_Data</td>
<td>Output Data</td>
<td>std_logic_vector</td>
<td>input</td>
</tr>
<tr>
<td>FSLn_S_Control</td>
<td>Control Bit indicating the FSLn_OUT_DATA is a Control Word</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>FSLn_S_Exists</td>
<td>Data Exists Bit indicating data exists in input FIFO</td>
<td>std_logic</td>
<td>input</td>
</tr>
</tbody>
</table>

**FSL BUS Write Operation**

The write to the FSL bus is controlled by the FSL_M_Write signal. When FSL_M_Write is set to '1' at the current clock edge, the data and control signals, FSL_M_Data and FSL_M_Control, are pushed onto the FSL FIFO on the next rising clock edge. The timing
diagram in Figure 2-18 depicts a back-to-back write operation on the FSL bus. The FIFO_Data and FIFO_Control signals denote the data and control signals of the FSL FIFO. When the FSL FIFO is full, the FSL_M_Full signal is set to ‘1’.

**FSL BUS Read Operation**

The read end of the FIFO always contains the last unpopped data and control signals, FSL_S_Data and FSL_S_Control. When the FIFO is empty the FSL_S_Exists signal is set to ‘0’ else it is set to ‘1’. When the slave side peripheral of the FSL has finished reading the data, the FSL_S_Read acknowledge signal must be set to ‘1’ for one clock cycle. The data and control values are then popped out of the top of the FIFO. See Figure 2-19 for timing diagram.

![Figure 2-19: FSL Back to Back Read Operation](image)

**Debug Interface**

The debug interface on MicroBlaze is designed to work with the Xilinx Microprocessor Debug Module (MDM) IP core, which interfaces with the JTAG port of Xilinx FPGAs. An external software debug tool can control MicroBlaze using the MDM core and the debug port on MicroBlaze. The MDM can support connections to multiple MicroBlaze debug ports. The debug signals on MicroBlaze are listed in Table 2-9.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dbg_Clk</td>
<td>JTAG Clock from MDM</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>Dbg_TDI</td>
<td>JTAG TDI from MDM</td>
<td>std_logic</td>
<td>input</td>
</tr>
</tbody>
</table>
Implementation

Parameterization

The following characteristics of MicroBlaze can be parameterized:

- Data Interface options: OPB only, LMB+OPB
- Instruction Interface options: LMB only, LMB+OPB, OPB only
- Barrel shifter
- Number of FSL interfaces (same number for both input and output)
- Interrupt port
- Debug port
- Instruction cache
- Data cache

Table 2-9: MicroBlaze Debug signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dbg_TDO</td>
<td>JTAG TDO to MDM</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Dbg_Reg_En</td>
<td>Debug Register Enable from MDM</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>Dbg_Capture</td>
<td>JTAG BSCAN Capture signal from MDM</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>Dbg_Update</td>
<td>JTAG BSCAN Update signal from MDM</td>
<td>std_logic</td>
<td>input</td>
</tr>
</tbody>
</table>

Table 2-10: MPD Parameters

<table>
<thead>
<tr>
<th>Feature/Description</th>
<th>Parameter Name</th>
<th>Allowable Values</th>
<th>Default Value</th>
<th>VHDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Family</td>
<td>C_FAMILY</td>
<td>Xilinx FPGA families</td>
<td>virtex2</td>
<td>string</td>
</tr>
<tr>
<td>Data Size</td>
<td>C_DATA_SIZE</td>
<td>32</td>
<td>32</td>
<td>integer</td>
</tr>
<tr>
<td>Instance Name</td>
<td>C_INSTANCE</td>
<td>Any instance name</td>
<td>microblaze</td>
<td>string</td>
</tr>
<tr>
<td>Data side OPB interface</td>
<td>C_D_OPB</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>Data side LMB interface</td>
<td>C_D_LMB</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>Instruction side OPB interface</td>
<td>C_I_OPB</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>Instruction side LMB interface</td>
<td>C_I_LMB</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>Barrel Shifter</td>
<td>C_USE_BARREL</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Divide Unit</td>
<td>C_USE_DIV</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Number of FSL interfaces</td>
<td>C_FSL_LINKS</td>
<td>0..8</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Feature/Description</td>
<td>Parameter Name</td>
<td>Allowable Values</td>
<td>Default Value</td>
<td>VHDL Type</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------</td>
<td>------------------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>FSL data bus size</td>
<td>C_FSL_DATA_SIZE</td>
<td>32</td>
<td>32</td>
<td>integer</td>
</tr>
<tr>
<td>Level/Edge Interrupt</td>
<td>C_INTERRUPT_IS_EDGE</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Negative/Positive Edge Interrupt</td>
<td>C_EDGE_IS_POSITIVE</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>MDM Debug interface</td>
<td>C_DEBUG_ENABLED</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Number of hardware breakpoints</td>
<td>C_NUMBER_OF_PC_BRK</td>
<td>0-8</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>Number of read address watchpoints</td>
<td>C_NUMBER_OF_RD_ADDR_BRK</td>
<td>0-4</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Number of write address watchpoints</td>
<td>C_NUMBER_OF_WR_ADDR_BRK</td>
<td>0-4</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>C_USE_ICACHE</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Instruction cache address tags</td>
<td>C_ADDR_TAG_BITS</td>
<td>0-24</td>
<td>7</td>
<td>integer</td>
</tr>
<tr>
<td>Instruction cache size</td>
<td>C_CACHE_BYTE_SIZE</td>
<td>512,1024,2048,4096,8192,16384,32768,65536</td>
<td>8192</td>
<td>integer</td>
</tr>
<tr>
<td>Instruction cache base address</td>
<td>C_ICACHE_BASEADDR</td>
<td>X“00000000” - X“FFFFFFFF”</td>
<td>X“00000000”</td>
<td>std_logic_vector</td>
</tr>
<tr>
<td>Instruction cache high address</td>
<td>C_ICACHE_HIGHADDR</td>
<td>X“00000000” - X“FFFFFFFF”</td>
<td>X“3FFFFFFFFF”</td>
<td>std_logic_vector</td>
</tr>
<tr>
<td>Instruction cache write enable</td>
<td>C_ALLOW_ICACHE_WR</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>Data cache</td>
<td>C_USE_DCACHE</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>Data cache address tags</td>
<td>C_DCACHE_ADDR_TAGS</td>
<td>0-24</td>
<td>7</td>
<td>integer</td>
</tr>
<tr>
<td>Data cache size</td>
<td>C_DCACHE_BYTE_SIZE</td>
<td>512,1024,2048,4096,8192,16384,32768,65536</td>
<td>8192</td>
<td>integer</td>
</tr>
<tr>
<td>Data cache base address</td>
<td>C_DCACHE_BASEADDR</td>
<td>X“00000000” - X“FFFFFFFF”</td>
<td>X“00000000”</td>
<td>std_logic_vector</td>
</tr>
<tr>
<td>Data cache high address</td>
<td>C_DCACHE_HIGHADDR</td>
<td>X“00000000” - X“FFFFFFFF”</td>
<td>X“3FFFFFFFFF”</td>
<td>std_logic_vector</td>
</tr>
<tr>
<td>Data cache write enable</td>
<td>C_ALLOW_DCACHE_WR</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
</tr>
</tbody>
</table>
Chapter 3

MicroBlaze Endianness

This chapter describes big-endian and little-endian data objects and how to use little-endian data with the big-endian MicroBlaze soft processor. This chapter includes the following sections

• “Definitions”
• “Bit Naming Conventions”
• “Data Types and Endianness”
• “VHDL Example”

Definitions

Data are stored or retrieved in memory, in byte, half word, word, or double word units. Endianness refers to the order in which data are stored and retrieved. Little-endian specifies that the least significant byte is assigned the lowest byte address. Big-endian specifies that the most significant byte is assigned the lowest byte address.

Note Endianness does not affect single byte data.

Bit Naming Conventions

The MicroBlaze architecture uses a bus and register bit naming convention in which the most significant bit (MSB) name incorporates zero (‘0’). As the significance of the bits decreases across the bus, the number in the name increases linearly so that a 32-bit vector has a least significant bit (LSB) name equal to 31. Other Xilinx interfaces such as the PCI Core use the opposite convention in which a name with a ‘0’ represents the LSB vector position.

Data Types and Endianness

Hardware supported data types for MicroBlaze are word, half word, and byte. The data organization for each type is shown in the following tables.

<table>
<thead>
<tr>
<th>Table 3-1: Word Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte address</td>
</tr>
<tr>
<td>Byte label</td>
</tr>
<tr>
<td>Byte significance</td>
</tr>
</tbody>
</table>
Chapter 3: MicroBlaze Endianness

The following C language structure includes various scalars and character strings. The comments indicate the value assumed to be in each structure element. These values show how the bytes comprising each structure element are mapped into storage.

```c
struct {
    int a; /* 0x1112_1314 word */
    long long b; /* 0x2122_2324_2526_2728 double word */
    char *c; /* 0x3132_3334 word */
    char d[7]; /* 'A','B','C','D','E','F','G' array of bytes */
    short e; /* 0x5152 halfword */
    int f; /* 0x6162_6364 word */
} s;
```

C structure mapping rules permit the use of padding (skipped bytes) to align scalars on desirable boundaries. The structure mapping examples show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between a and b, one byte between d and e, and two bytes between e and f. The same amount of padding is present in both big-endian and little-endian mappings.

Note For the MicroBlaze core, all operands in the ALU and GPRs, and all pipeline instructions are big-endian.
The big-endian mapping of “struct” is shown in the following table. (The data is highlighted in the structure mappings). Hexadecimal addresses are below the data stored at the address. The contents of each byte, as defined in the structure, are shown as a number (hexadecimal) or character (for the string elements).

**Table 3-4: Big-endian Mapping**

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>0x04</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>0x03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>0x08</td>
<td>0x09</td>
<td>0x0A</td>
<td>0x0B</td>
<td>0x0C</td>
<td>0x0D</td>
<td>0x0E</td>
<td>0x0F</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>‘A’</td>
<td>‘B’</td>
<td>‘C’</td>
<td>‘D’</td>
</tr>
<tr>
<td>0x10</td>
<td>0x11</td>
<td>0x12</td>
<td>0x13</td>
<td>0x14</td>
<td>0x15</td>
<td>0x16</td>
<td>0x17</td>
</tr>
<tr>
<td>‘E’</td>
<td>‘F’</td>
<td>‘G’</td>
<td>0x1A</td>
<td>0x1B</td>
<td>0x1C</td>
<td>0x1D</td>
<td>0x1E</td>
</tr>
<tr>
<td>0x18</td>
<td>0x19</td>
<td></td>
<td>0x1B</td>
<td></td>
<td>0x1C</td>
<td></td>
<td>0x1D</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>0x23</td>
<td>0x24</td>
<td>0x25</td>
<td>0x26</td>
</tr>
<tr>
<td>0x20</td>
<td>0x21</td>
<td>0x22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3-5: Little-endian Mapping**

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>0x04</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>0x03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>0x08</td>
<td>0x09</td>
<td>0x0A</td>
<td>0x0B</td>
<td>0x0C</td>
<td>0x0D</td>
<td>0x0E</td>
<td>0x0F</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>‘A’</td>
<td>‘B’</td>
<td>‘C’</td>
<td>‘D’</td>
</tr>
<tr>
<td>0x10</td>
<td>0x11</td>
<td>0x12</td>
<td>0x13</td>
<td>0x14</td>
<td>0x15</td>
<td>0x16</td>
<td>0x17</td>
</tr>
<tr>
<td>‘E’</td>
<td>‘F’</td>
<td>‘G’</td>
<td>0x1A</td>
<td>0x1B</td>
<td>0x1C</td>
<td>0x1D</td>
<td>0x1E</td>
</tr>
<tr>
<td>0x18</td>
<td>0x19</td>
<td></td>
<td>0x1B</td>
<td></td>
<td>0x1C</td>
<td></td>
<td>0x1D</td>
</tr>
<tr>
<td>64</td>
<td>63</td>
<td>62</td>
<td>61</td>
<td>0x23</td>
<td>0x24</td>
<td>0x25</td>
<td>0x26</td>
</tr>
<tr>
<td>0x20</td>
<td>0x21</td>
<td>0x22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VHDL Example**

**BRAM – LMB Example**

LMB uses big-endian byte addressing, while the BRAM uses little-endian byte addressing. To translate data between the two busses, swap the data and address bytes.
Interface Between BRAM and MicroBlaze

```vhdl
entity Local_Memory is
  port (
    Clk : in std_logic;
    Reset : in boolean;

    -- Instruction Bus
    Instr_Addr : in std_logic_vector(0 to 31);
    Instr : out std_logic_vector(0 to 31);
    IFetch : in std_logic;
    I_AS : in std_logic;
    IReady : out std_logic;

    -- ports to "Decode_I"
    Data_Addr : in std_logic_vector(0 to 31);
    Data_Read : out std_logic_vector(0 to 31);
    Data_Write : in std_logic_vector(0 to 31);
    D_AS : in std_logic;
    Read_Strobe : in std_logic;
    Write_Strobe : in std_logic;
    DReady : out std_logic;
    Byte_Enable : in std_logic_vector(0 to 3);
  );
end Local_Memory;

architecture IMP of Local_Memory is

BRAM Component Declaration (little-endian)

```vhdl
component mem_dp_0 is
  port (
    addra : in std_logic_vector(9 downto 0);
    addrb : in std_logic_vector(9 downto 0);
    clka : in std_logic;
    clkb : in std_logic;
    dinb : in std_logic_vector(7 downto 0);
    douta : out std_logic_vector(7 downto 0);
    doutb : out std_logic_vector(7 downto 0);
    web : in std_logic);
end component mem_dp_0;

Swap BRAM Little-endian Data to Big-endian

```vhdl
Swap_BE_and_LE_order : process (...)
begin
  for I in addra'range loop
    addra(I) <= Instr_Addr(29-I);
  end loop;
  for I in addrb'range loop
    addrb(I) <= Data_Addr(29-I);
  end loop;
  for I in 0 to 3 loop
    for J in 0 to 7 loop
      dinb(I*8+J) <= Data_Write((3-I)*8+(7-J));
      Instr((3-I)*8+(7-J)) <= douta(I*8+J);
      Data_Read((3-I)*8+(7-J)) <= doutb(I*8+J);
    end loop;
  end loop;
end process;
```
VHDL Example

end loop;
end process Swap_BE_and_LE_order;

BRAM Instantiation

mem_dp_0_I : mem_dp_0
port map (
  addra=>addra, --[IN std_logic_VECTOR(9 downto 0)]
  addrb=>addrb, --[IN std_logic_VECTOR(9 downto 0)]
  clka=>Clk, --[IN std_logic]
  clkb=>Clk, --[IN std_logic]
  dinb=>dinb(31 downto 24)--[IN std_logic_VECTOR(7 downto 0)]
  douta=>douta(31 downto 24), --[OUT std_logic_VECTOR(7 downto 0)]
  doutb => doutb(31 downto 24), --[OUT std_logic_VECTOR(7 downto 0)]
  web=>we(0)); --[IN std_logic]

BRAM – OPB Example

OPB uses big-endian byte addressing, while the BRAM uses little-endian byte addressing. To translate data between the two buses, swap the data and address bytes.

Interface Between BRAM and MicroBlaze

library IEEE;
use IEEE.std_logic_1164.all;

entity OPB_BRAM is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"B000_0000";
    C_NO_BRAMS : natural := 4;  -- Can be 4,8,16,32 only
    C_VIRTEXII : boolean := true
  );
  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;
    -- OPB signals
    OPB_ABus : in std_logic_vector(0 to 31);
    OPB_BE : in std_logic_vector(0 to 3);
    OPB_RNW : in std_logic;
    OPB_select : in std_logic;
    OPB_seqAddr : in std_logic;
    OPB_DBus : in std_logic_vector(0 to 31);
    -- OPB_BRAM signals (other port)
    BRAM_Clk : in std_logic;
    BRAM_Addr : in std_logic_vector(0 to 31);
    BRAM_WE : in std_logic_vector(0 to 3);
    BRAM_Write_Data : in std_logic_vector(0 to 31);
    BRAM_Read_Data : out std_logic_vector(0 to 31);
    OPB_BRAM_DBus : out std_logic_vector(0 to 31);
    OPB_BRAM_errAck : out std_logic;
    OPB_BRAM_retry : out std_logic;
    OPB_BRAM_toutSup : out std_logic;
    OPB_BRAM_xferAck : out std_logic;
  );
end entity OPB_BRAM;

architecture IMP of OPB_BRAM is

BRAM Component Declaration (little-endian)

component 'RAMB16_S9_S9
    port ( 
        DIA : in  std_logic_vector (7 downto 0);
        DIB : in  std_logic_vector (7 downto 0);
        DIPA : in  std_logic_vector (0 downto 0);
        DIPB : in  std_logic_vector (0 downto 0);
        ENA : in  std_ulogic;
        ENB : in  std_ulogic;
        WEA : in  std_ulogic;
        WEB : in  std_ulogic;
        WEB : in  std_ulogic;
        SSRA : in  std_ulogic;
        SSRB : in  std_ulogic;
        CLKA : in  std_ulogic;
        CLKB : in  std_ulogic;
        ADDRA : in  std_logic_vector (10 downto 0);
        ADDRB : in  std_logic_vector (10 downto 0);
        DOA : out std_logic_vector (7 downto 0);
        DOB : out std_logic_vector (7 downto 0);
        DOPA : out std_logic_vector (0 downto 0);
        DOPB : out std_logic_vector (0 downto 0) );
end component;

Swap BRAM Little-endian Data to Big-endian
BE_to_LE : for I in 0 to 31 generate
    opb_dbus_le(I) <= OPB_DBus(31-I);
    bram_write_data_le(I) <= BRAM_Write_Data(31-I);
    BRAM_Read_Data(I)    <= bram_Read_Data_LE(31-I);
    opb_ABus_LE(I) <= OPB_ABus(31-I);
    bram_Addr_LE(I) <= BRAM_Addr(31-I);
end generate BE_to_LE;

BRAM Instantiation

All_Brams : for I in 0 to C_NO_BRAMS-1 generate

By_8 : if (C_NO_BRAMS = 4) generate

RAMB16_S9_S9_I : RAMB16_S9_S9
port map ( 
    DIA => opb_DBUS_LE(((I+1)*8-1) downto I*8), --[in std_logic_vector(7
downto 0)]
    DIB => bram_Write_Data_LE(((I+1)*8)-1 downto I*8), --[in
std_logic_vector (downto 0)]
    DIPA => null_1, -- [in  std_logic_vector (7 downto 0)]
    DIPB  => null_1, -- [in  std_logic_vector (7 downto 0)]
    ENA   => '1', -- [in  std_ulogic]
    ENB   => '1', -- [in  std_ulogic]
    WEA   => opb_WE(I), -- [in std_ulogic]
    WEB   => BRAM_WE(I), -- [in std_ulogic]
    SSRA  => '0', -- [in std_ulogic]
    SSRB  => '0', -- [in std_ulogic]
    CLKA  => OPB_Clkl, -- [in std_ulogic]
    CLKB  => BRAM_ClkI, -- [in std_ulogic]
ADDRA => opb_ABus_LE(12 downto 2), -- [in std_logic_vector (10 downto 0)]
ADDRB => bram_Addr_LE(12 downto 2), -- [in std_logic_vector (10 downto 0)]
DOA=>opb_BRAM_DBus_LE_I(((I+1)*8-1)downto I*8),--[out std_logic_vector(7 downto 0)]
DOB =>bram_Read_Data_LE(((I+1)*8-1) downto I*8),--[out std_logic_vector(7 downto 0)]
DOPA  => open, -- [out std_logic_vector (0 downto 0)]
DOPB  => open); -- [out std_logic_vector (0 downto 0)]
end generate By_8;
Chapter 4

MicroBlaze Application Binary Interface

Scope

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 4-1. Data types such as data8, data16, and data32 are used in place of the usual byte, halfword, and word.

Table 4-1: Data types in MicroBlaze assembly programs

<table>
<thead>
<tr>
<th>MicroBlaze data types (for assembly programs)</th>
<th>Corresponding ANSI C data types</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data8</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>data16</td>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>data32</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>data32</td>
<td>long int</td>
<td>4</td>
</tr>
<tr>
<td>data32</td>
<td>enum</td>
<td>4</td>
</tr>
<tr>
<td>data16/data32</td>
<td>pointer(^a)</td>
<td>2/4</td>
</tr>
</tbody>
</table>

\(^a\) 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 4-2.
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 the 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 stores the value of the stack pointer and is updated on entry and exit from functions.

### Table 4-2: Register usage conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>Dedicated</td>
<td>Value 0</td>
</tr>
<tr>
<td>R1</td>
<td>Dedicated</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>R2</td>
<td>Dedicated</td>
<td>Read-only small data area anchor</td>
</tr>
<tr>
<td>R3-R4</td>
<td>Volatile</td>
<td>Return Values</td>
</tr>
<tr>
<td>R5-R10</td>
<td>Volatile</td>
<td>Passing parameters/Temporaries</td>
</tr>
<tr>
<td>R11-R12</td>
<td>Volatile</td>
<td>Temporaries</td>
</tr>
<tr>
<td>R13</td>
<td>Dedicated</td>
<td>Read-write small data area anchor</td>
</tr>
<tr>
<td>R14</td>
<td>Dedicated</td>
<td>Return address for Interrupt</td>
</tr>
<tr>
<td>R15</td>
<td>Dedicated</td>
<td>Return address for Sub-routine</td>
</tr>
<tr>
<td>R16</td>
<td>Dedicated</td>
<td>Return address for Trap (Debugger)</td>
</tr>
<tr>
<td>R17</td>
<td>Dedicated</td>
<td>Return Address for Exceptions</td>
</tr>
<tr>
<td>R18</td>
<td>Dedicated</td>
<td>Reserved for Assembler</td>
</tr>
<tr>
<td>R19-R31</td>
<td>Non-Volatile</td>
<td>Must be saved across function calls</td>
</tr>
<tr>
<td>RPC</td>
<td>Special</td>
<td>Program counter</td>
</tr>
<tr>
<td>RMSR</td>
<td>Special</td>
<td>Machine Status Register</td>
</tr>
</tbody>
</table>
Register R18 is used as a temporary register for assembler operations.

MicroBlaze has certain special registers such as a 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 the “MicroBlaze Application Binary Interface” chapter).

Stack Convention

The stack conventions used by MicroBlaze are detailed in Figure 4-1

The shaded area in Figure 4-1 denotes a part of the caller function’s stack frame, while the unshaded 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 4-1: Stack Convention

<table>
<thead>
<tr>
<th>High Address</th>
<th>Function Parameters for called sub-routine (Arg n ..Arg1) (Optional: Maximum number of arguments required for any called procedure from the current procedure.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Stack Pointer</td>
<td>Link Register (R15) (Callee Saved Register (R31....R19) (Optional: Only those registers which are used by the current procedure are saved))</td>
</tr>
<tr>
<td>Local Variables for Current Procedure (Optional: Present only if Locals defined in the procedure)</td>
<td></td>
</tr>
<tr>
<td>Functional Parameters (Arg n .. Arg 1) (Optional: Maximum number of arguments required for any called procedure from the current procedure)</td>
<td></td>
</tr>
<tr>
<td>New Stack Pointer</td>
<td>Link Register</td>
</tr>
<tr>
<td>Low Address</td>
<td></td>
</tr>
</tbody>
</table>
Consider an example where Func1 calls Func2, which in turn calls Func3. The stack representation at different instances is depicted in Figure 4-2. After the call from Func 1 to Func 2, the value of the 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 the stack pointer is increased to its original value in the function, Func 2.

Details of how the stack is maintained are shown in Figure 4-2.

![Figure 4-2: Stack Frame](image)

**Calling Convention**

The caller function passes parameters to the callee function using either the registers (R5 through R10) or on its own stack frame. The callee uses the caller’s stack area to store the parameters passed to the callee.

Refer to Figure 4-2. The parameters for Func 2 are stored either in the registers R5 through R10 or on the stack frame allocated for Func 1.

**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 GNU Compiler Tools chapter. 64K bytes of memory is allocated for the small data areas. The 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 anchor R13.

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 are decided by the linker, in the final linking stages. For more information on the various sections of the memory please refer to the Address Management chapter. The compiler generates appropriate sections, depending on the command line options. Please refer to the GNU Compiler Tools chapter for more information about these options.

Interrupt and Exception Handling

MicroBlaze assumes certain address locations for handling interrupts and exceptions as indicated in Table 4-3. 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 4-3: Interrupt and Exception Handling

<table>
<thead>
<tr>
<th>On</th>
<th>Hardware jumps to</th>
<th>Software Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start / Reset</td>
<td>0x0</td>
<td>_start</td>
</tr>
<tr>
<td>Exception</td>
<td>0x8</td>
<td>_exception_handler</td>
</tr>
<tr>
<td>Interrupt</td>
<td>0x10</td>
<td>_interrupt_handler</td>
</tr>
</tbody>
</table>

The code expected at these locations is as shown in Figure 4-3. 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 4-3: Code for passing control to exception and interrupt handlers

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>bri _start1</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td>imm high bits of address (exception handler)</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td>bri _exception_handler</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>imm high bits of address (interrupt handler)</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td>bri _interrupt_handler</td>
<td></td>
</tr>
</tbody>
</table>

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 you can change. In order to override these routines and link your interrupt and exception handlers, you must define the interrupt handler code with an attribute interrupt_handler. For more details about the use and syntax of the interrupt handler attribute, please refer to the GNU Compiler Tools chapter.
Chapter 5

MicroBlaze Instruction Set Architecture

Summary

This chapter provides a detailed guide to the Instruction Set Architecture of MicroBlaze™.

Notation

The symbols used throughout this document are defined in Table 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>×</td>
<td>Multiply</td>
</tr>
<tr>
<td>∧</td>
<td>Bitwise logical AND</td>
</tr>
<tr>
<td>∨</td>
<td>Bitwise logical OR</td>
</tr>
<tr>
<td>⊕</td>
<td>Bitwise logical XOR</td>
</tr>
<tr>
<td>¬</td>
<td>Bitwise logical complement of x</td>
</tr>
<tr>
<td>←</td>
<td>Assignment</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
</tr>
<tr>
<td>rx</td>
<td>Register x</td>
</tr>
<tr>
<td>x[i]</td>
<td>Bit i in register x</td>
</tr>
<tr>
<td>x[i:j]</td>
<td>Bits i through j in register x</td>
</tr>
<tr>
<td>=</td>
<td>Equal comparison</td>
</tr>
<tr>
<td>≠</td>
<td>Not equal comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than comparison</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal comparison</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than comparison</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal comparison</td>
</tr>
<tr>
<td>sext(x)</td>
<td>Sign-extend x</td>
</tr>
</tbody>
</table>
Chapter 5: MicroBlaze Instruction Set Architecture

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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination Reg</th>
<th>Source Reg A</th>
<th>Source Reg B</th>
<th>Immediate Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination Reg</th>
<th>Source Reg A</th>
<th>Immediate Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

Instructions

MicroBlaze instructions are described next. Instructions are listed in alphabetical order. For each instruction Xilinx provides the mnemonic, encoding, a description of it, pseudocode of its semantics, and a list of registers that it modifies.
add

Arithmetic Add

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>addc</td>
<td>Add with Carry</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>addk</td>
<td>Add and Keep Carry</td>
<td>rD, rA, rB</td>
</tr>
<tr>
<td>addkc</td>
<td>Add with Carry and Keep Carry</td>
<td>rD, rA, rB</td>
</tr>
</tbody>
</table>

### 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

```plaintext
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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>rD, rA, IMM</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>addic</td>
<td>rD, rA, IMM</td>
<td>Add Immediate with Carry</td>
</tr>
<tr>
<td>addik</td>
<td>rD, rA, IMM</td>
<td>Add Immediate and Keep Carry</td>
</tr>
<tr>
<td>addikc</td>
<td>rD, rA, IMM</td>
<td>Add Immediate with Carry and Keep Carry</td>
</tr>
</tbody>
</table>

![Instruction Format](image)

**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**

\[
\begin{align*}
\text{if } C = 0 \text{ then} \\
& (rD) \leftarrow (rA) + \text{sext(IMM)} \\
\text{else} \\
& (rD) \leftarrow (rA) + \text{sext(IMM)} + \text{MSR}[C] \\
\text{if } K = 0 \text{ then} \\
& \text{MSR}[C] \leftarrow \text{CarryOut}
\end{align*}
\]

**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 the imm instruction for details on using 32-bit immediate values.
and

Logical AND

\[
\text{and} \quad rD, rA, rB
\]

<table>
<thead>
<tr>
<th>1 0 0 0 0 1</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

The contents of register rA are ANDed with the contents of register rB; the result is placed into register rD.

Pseudocode

\[
(rD) \leftarrow (rA) \land (rB)
\]

Registers Altered

- rD

Latency

1 cycle
**andi**

Logical AND with Immediate

```
andi rD, rA, IMM
```

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The contents of register rA are ANDeed 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 the imm instruction for details on using 32-bit immediate values.
andn  Logical AND NOT

\[
\text{andn} \quad r_D, r_A, r_B
\]

| 1 | 0 | 0 | 0 | 1 | 1 | r_D | r_A | r_B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 6 | 11 | 16 | 21 | 31 |

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

\[
(r_D) \leftarrow (r_A) \land (\overline{r_B})
\]

Registers Altered

- rD

Latency

1 cycle
andni

Logical AND NOT with Immediate

andni  rD, rA, IMM

<table>
<thead>
<tr>
<th>1 0 1 0 1</th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 6 11 16 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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) ← (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 the imm instruction for details on using 32-bit immediate values.
### beq

**Branch if Equal**

\[
\text{beq} \quad \text{rA, rB} \quad \text{Branch if Equal}
\]

\[
\text{beqd} \quad \text{rA, rB} \quad \text{Branch if Equal with Delay}
\]

| 1 | 0 | 0 | 1 | 1 | D | 0 | 0 | 0 | 0 | rA | rB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 6 | 11 | 16 | 21 | 31 |

**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\[\text{rA, IMM}\] Branch Immediate if Equal
beqid\[\text{rA, IMM}\] Branch Immediate if Equal with Delay

<table>
<thead>
<tr>
<th>1 0 1 1 1 1</th>
<th>D 0 0 0 0</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6 0 11 16</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

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 the imm instruction for details on using 32-bit immediate values.
Instructions

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**

<table>
<thead>
<tr>
<th>bgei</th>
<th>rA, IMM</th>
<th>Branch Immediate if Greater or Equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgeid</td>
<td>rA, IMM</td>
<td>Branch Immediate if Greater or Equal with Delay</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 0 1 1 1</th>
<th>D 0 1 0 1</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 6 11 16</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 bgei 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**

```plaintext
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 the 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

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**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 the imm instruction for details on using 32-bit immediate values.
ble \hspace{1em} \textbf{Branch if Less or Equal} \\

\textbf{ble} \hspace{1em} rA, rB \hspace{1em} \text{Branch if Less or Equal} \\
\textbf{bled} \hspace{1em} rA, rB \hspace{1em} \text{Branch if Less or Equal with Delay} \\

<table>
<thead>
<tr>
<th>\text{Address}</th>
<th>rA</th>
<th>rB</th>
<th>D</th>
<th>\text{rA &lt;= 0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\textit{Description}

Branch if \( \text{rA} \) is less or equal to 0, to the instruction located in the offset value of \( \text{rB} \). The target of the branch will be the instruction at address \( \text{PC} + \text{rB} \).

The mnemonic \textit{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.

\textit{Pseudocode}

\begin{verbatim}
If \( \text{rA} \leq 0 \) then \\
    \text{PC} \leftarrow \text{PC} + \text{rB} \\
else \\
    \text{PC} \leftarrow \text{PC} + 4 \\
    \text{if D = 1 then} \\
    \text{allow following instruction to complete execution}
\end{verbatim}

\textit{Registers Altered}

- \text{PC}

\textit{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

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>D</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 the 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 \( rA, \text{IMM} \) Branch Immediate if Less Than  
bltid \( rA, \text{IMM} \) Branch Immediate if Less Than with Delay

<table>
<thead>
<tr>
<th>1 0 1 1 1 1</th>
<th>D 0 0 1 0</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**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  
\[ PC \leftarrow PC + \text{sext(IMM)} \]
else  
\[ PC \leftarrow PC + 4 \]
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 the 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

<table>
<thead>
<tr>
<th>1 0 0 1 1 1</th>
<th>D 0 0 0 1</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

**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**

```plaintext
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

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>D</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>ra</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 the imm instruction for details on using 32-bit immediate values.
**br**

### Unconditional Branch

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>br</strong></td>
<td>rB</td>
<td>Branch</td>
</tr>
<tr>
<td><strong>bra</strong></td>
<td>rB</td>
<td>Branch Absolute</td>
</tr>
<tr>
<td><strong>brd</strong></td>
<td>rB</td>
<td>Branch with Delay</td>
</tr>
<tr>
<td><strong>brad</strong></td>
<td>rB</td>
<td>Branch Absolute with Delay</td>
</tr>
<tr>
<td><strong>brld</strong></td>
<td>rD, rB</td>
<td>Branch and Link with Delay</td>
</tr>
<tr>
<td><strong>brald</strong></td>
<td>rD, rB</td>
<td>Branch Absolute and Link with Delay</td>
</tr>
</tbody>
</table>

#### Pseudocode

```plaintext
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) or 3 cycles (if the D bit is not set)

#### Note
The instructions brl and bral are not available.

#### 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.
### bri

**Unconditional Branch Immediate**

<table>
<thead>
<tr>
<th>bri</th>
<th>IMM</th>
<th>Branch Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>brai</td>
<td>IMM</td>
<td>Branch Absolute Immediate</td>
</tr>
<tr>
<td>brid</td>
<td>IMM</td>
<td>Branch Immediate with Delay</td>
</tr>
<tr>
<td>braid</td>
<td>IMM</td>
<td>Branch Absolute Immediate with Delay</td>
</tr>
<tr>
<td>brlid</td>
<td>rD, IMM</td>
<td>Branch and Link Immediate with Delay</td>
</tr>
<tr>
<td>bralid</td>
<td>rD, IMM</td>
<td>Branch Absolute and Link Immediate with Delay</td>
</tr>
</tbody>
</table>

#### 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

```c
if L = 1 then
   (rD) ← PC
if A = 1 then
   PC ← (IMM)
else
   PC ← PC + (IMM)
if D = 1 then
   allow following instruction to complete execution
```

#### Registers Altered

- rD
- PC

#### Latency

2 cycles (if the D bit is set) or 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 the imm instruction for details on using 32-bit immediate values.
brk  

**Break**

\[
\text{brk} \quad rD, rB
\]

<table>
<thead>
<tr>
<th>1 0 0 1 1 0</th>
<th>rD</th>
<th>0 1 1 0 0</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 6 11 16 21 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
\text{(rD)} & \leftarrow \text{PC} \\
\text{PC} & \leftarrow \text{(rB)} \\
\text{MSR[BIP]} & \leftarrow 1
\end{align*}
\]

**Registers Altered**

- rD
- PC
- MSR[BIP]

**Latency**

3 cycles
**brki**

**Break Immediate**

\[
\text{brki} \quad rD, \text{IMM}
\]

<table>
<thead>
<tr>
<th>1 0 1 1 0</th>
<th>rD</th>
<th>0 1 1 0 0</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
(rD) & \leftarrow \text{PC} \\
\text{PC} & \leftarrow \text{sext(IMM)} \\
\text{MSR}[\text{BIP}] & \leftarrow 1
\end{align*}
\]

**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 the imm instruction for details on using 32-bit immediate values.
Chapter 5: MicroBlaze Instruction Set Architecture

bs
Barrel Shift

bsrl rD, rA, rB Barrel Shift Right Logical
bsra rD, rA, rB Barrel Shift Right Arithmetical
bsll rD, rA, rB Barrel Shift Left Logical

|   | rD | rA | rB | S | T | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 6  | 11 | 16 | 21| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 31|

Description

Shifts the contents of register rA by the amount specified in register rB and puts the result in register rD.

The mnemonic bsll sets the S bit (Side bit). If the S bit is set, the barrel shift is done to the left. The mnemonics bsrl and bsra clear the S bit and the shift is done to the right.

The mnemonic bsra will set the T bit (Type bit). If the T bit is set, the barrel shift performed is Arithmetical. The mnemonics bsrl and bsll clear the T bit and the shift performed is Logical.

Pseudocode

if S = 1 then
   (rD) ← (rA) << (rB)[27:31]
else
   if T = 1 then
      if ((rB)[27:31]) ≠ 0 then
         (rD)[0:(rB)[27:31]-1] ← (rA)[0]
         (rD)[(rB)[27:31]:31] ← (rA) >> (rB)[27:31]
      else
         (rD) ← (rA)
   else
      (rD) ← (rA) >> (rB)[27:31]

Registers Altered

• rD

Latency

2 cycles

Note

These instructions are optional. To use them, MicroBlaze has to be configured to use barrel shift instructions.
bsi

Barrel Shift Immediate

\[
\begin{array}{|c|c|}
\hline
\text{bsrli} & \text{rD, rA, IMM} \\
\text{bsrai} & \text{rD, rA, IMM} \\
\text{bslli} & \text{rD, rA, IMM} \\
\hline
\end{array}
\]

Description
Shifts the contents of register \( rA \) by the amount specified by IMM and puts the result in register \( rD \).

The mnemonic bsll sets the S bit (Side bit). If the S bit is set, the barrel shift is done to the left. The mnemonics bsrl and bsra clear the S bit and the shift is done to the right.

The mnemonic bsra will set the T bit (Type bit). If the T bit is set, the barrel shift performed is Arithmetical. The mnemonics bsrl and bsll clear the T bit and the shift performed is Logical.

Pseudocode
\[
\begin{align*}
\text{if } S & = 1 \text{ then} \\
& (rD) \leftarrow (rA) \ll \text{ IMM} \\
\text{else} \\
& \text{if } T = 1 \text{ then} \\
& \text{if } \text{ IMM} \neq 0 \text{ then} \\
& (rD)[0:\text{IMM-1}] \leftarrow (rA)[0] \\
& (rD)[\text{IMM:31}] \leftarrow (rA) \gg \text{ IMM} \\
& \text{else} \\
& (rD) \leftarrow (rA) \\
& \text{else} \\
& (rD) \leftarrow (rA) \gg \text{ IMM}
\end{align*}
\]

Registers Altered
- rD

Latency
2 cycles

Notes
These are not Type B Instructions. There is no effect from a preceding imm instruction.

These instructions are optional. To use them, MicroBlaze has to be configured to use barrel shift instructions.
cmp

Integer Compare

cmp  rD, rA, rB  compare rB with rA (signed)
cmpu rD, rA, rB  compare rB with rA (unsigned)

Description

The contents of register rA is subtracted from the contents of register rB and the result is placed into register rD.

The MSB bit of rD is adjusted to shown true relation between rA and rB. If the U bit is set, rA and rB is considered unsigned values. If the U bit is clear, rA and rB is considered signed values.

Pseudocode

\[
\text{if } (rA) = (rB) \text{ then} \\
\quad (rD) \leftarrow 0 \\
\text{else} \\
\quad (rD)\text{ (MSB)} \leftarrow (rA) > (rB)
\]

Registers Altered

- rD

Latency

1 cycle
Instructions

get

get from fsl interface

get \ rD, FSLx \ get data from FSL x (blocking)

nget \ rD, FSLx \ get data from FSL x (non-blocking)

cget \ rD, FSLx \ get control from FSL x (blocking)

ncget \ rD, FSLx \ get control from FSL x (non-blocking)

Description

MicroBlaze will read from the FSLx interface and place the result in register rD.

The get instruction has four variants.

The blocking versions (when ‘n’ bit is ‘0’) will stall microblaze until the data from the FSL interface is valid. The non-blocking versions will not stall microblaze and will set carry to ‘0’ if the data was valid and to ‘1’ if the data was invalid.

The get and nget instructions expect the control bit from the FSL interface to be ‘0’. If this is not the case, the instruction will set MSR[FSL_Error] to ‘1’. The cget and ncget instructions expect the control bit from the FSL interface to be ‘1’. If this is not the case, the instruction will set MSR[FSL_Error] to ‘1’.

Pseudocode

\[(rD) \leftarrow \text{FSLx} \]
\[\text{if } (n = 1) \text{ then} \]
\[\text{MSR}[\text{Carry}] \leftarrow \text{not (FSLx Exists bit)} \]
\[\text{if } ((\text{FSLx Control bit}) == c) \text{ then} \]
\[\text{MSR}[\text{FSL_Error}] \leftarrow 0 \]
\[\text{else} \]
\[\text{MSR}[\text{FSL_Error}] \leftarrow 1 \]

Registers Altered

- \ rD
- \ MSR[FSL_Error]
- \ MSR[Carry]

Latency

2 cycles if non-blocking or if data is valid at the FSL interface. For blocking instruction, MicroBlaze will stall until the data is valid

Note

For nget and ncget, a rsbce instruction can be used for counting down a index variable
idiv

Integer Divide

idiv  rd, ra, rb  divide rb by ra (signed)
idivu rd, ra, rb  divide rb by ra (unsigned)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rD</td>
<td>rA</td>
<td>rB</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
</tr>
</tbody>
</table>

Description

The contents of register rB is divided by the contents of register rA and the result is placed into register rD.

If the U bit is set, ra and rb is considered unsigned values. If the U bit is clear, ra and rb is considered signed values.

If the value of rA is 0, the divide_by_zero bit in MSR will be set and the value in rD will be 0.

Pseudocode

if (rA) = 0 then
  (rD) ← 0
else
  (rD) ← (rB) / (rA)

Registers Altered

- rD
- MSR[Divide_By_Zero]

Latency

2 cycles if (rA) = 0, otherwise 34 cycles

Note

This instruction is only valid if MicroBlaze is configured to use a hardware divider.
### 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

### Notes

The `imm` instruction and the Type B instruction following it are atomic, hence no interrupts are allowed between them.

The assembler provided by Xilinx automatically detects the need for `imm` instructions. When a 32-bit IMM value is specified in a Type B instruction, the assembler converts the IMM value to a 16-bit one to assemble the instruction and inserts an `imm` instruction before it in the executable file.

<table>
<thead>
<tr>
<th>imm</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110000000000000000000000000000</td>
<td>IMM</td>
</tr>
<tr>
<td>0 6 11 16 31</td>
<td></td>
</tr>
</tbody>
</table>
**lbu**

**Load Byte Unsigned**

\[
\text{lbu} \quad rD, rA, rB
\]

<table>
<thead>
<tr>
<th>1 1 0 0 0 0</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
\text{Addr} & \leftarrow (rA) + (rB) \\
(rD)[24:31] & \leftarrow \text{Mem}(\text{Addr}) \\
(rD)[0:23] & \leftarrow 0
\end{align*}
\]

**Registers Altered**

- \( rD \)

**Latency**

2 cycles
**lbui**  
**Load Byte Unsigned Immediate**

\[ \text{lbui} \quad rD, rA, \text{IMM} \]

<table>
<thead>
<tr>
<th></th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
\text{Addr} & \leftarrow (rA) + \text{sext}(\text{IMM}) \\
(rD)[24:31] & \leftarrow \text{Mem}(\text{Addr}) \\
(rD)[0:23] & \leftarrow 0
\end{align*}
\]

**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 the imm instruction for details on using 32-bit immediate values.
**lhu**

**Load Halfword Unsigned**

```
lhu  rD, rA, rB
```

<table>
<thead>
<tr>
<th>1 1 0 0 0 1</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 11 16 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\text{lhui} \quad rD, rA, \text{IMM}
\]

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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 \( \text{IMM} \), 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

\[
\begin{align*}
\text{Addr} & \leftarrow (rA) + \text{sext}(\text{IMM}) \\
\text{Addr}[31] & \leftarrow 0 \\
(rD)[16:31] & \leftarrow \text{Mem}(\text{Addr}) \\
(rD)[0:15] & \leftarrow 0
\end{align*}
\]

#### Registers Altered

- \( rD \)

#### Latency

2 cycles

#### Note

By default, Type B Instructions will take the 16-bit \( \text{IMM} \) 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 \( \text{imm} \) instruction. See the \( \text{imm} \) instruction for details on using 32-bit immediate values.
# Load Word

**lw**  

### 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

\[
\text{Addr} \leftarrow (rA) + (rB) \\
\text{Addr}[30:31] \leftarrow 00 \\
(rD) \leftarrow \text{Mem} (\text{Addr})
\]

### Registers Altered

- rD

### Latency

2 cycles
### lwI

**Load Word Immediate**

\[ \text{lwI} \quad \text{rD, rA, IMM} \]

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>IMM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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

\[
\begin{align*}
\text{Addr} & \leftarrow (\text{rA}) + \text{sext} (\text{IMM}) \\
\text{Addr}[30:31] & \leftarrow 00 \\
(\text{rD}) & \leftarrow \text{Mem}(\text{Addr})
\end{align*}
\]

#### 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 the imm instruction for details on using 32-bit immediate values.
**mfs**  
**Move From Special Purpose Register**

\[ \text{mfs} \quad rD, rS \]

<table>
<thead>
<tr>
<th>1 0 0 1 0 1</th>
<th>0 0 0 0 0 0</th>
<th>1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 rS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Copies the contents of the special purpose register \( rS \) into register \( rD \).

**Pseudocode**

\[
(rD) \leftarrow (rS)
\]

**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**

\[
\text{mts } \quad \text{rS, rA}
\]

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 31 |
| 0 | 6 | 11 | 16 | \(rS\) | \(rA\) |

**Description**

Copies the contents of register rD into the MSR register.

**Pseudocode**

\[
(rS) \leftarrow (rA)
\]

**Registers Altered**

- rS

**Latency**

1 cycle

**Notes**

You cannot 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. The most significant word is discarded.

Pseudocode

(rD) ← LSW( (rA) × (rB) )

Registers Altered

• rD

Latency

3 cycles

Note

This instruction is only valid if the target architecture has an embedded multiplier.
muli

Multiply Immediate

\[
muli \quad rD, rA, IMM
\]

<table>
<thead>
<tr>
<th></th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

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 \( rD \). The most significant word is discarded.

Pseudocode

\[
(rD) \leftarrow \text{LSW}( (rA) \times \text{sext}(IMM) )
\]

Registers Altered

- \( rD \)

Latency

3 cycles

Notes

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 the imm instruction for details on using 32-bit immediate values.

This instruction is only valid if the target architecture has an embedded multiplier.
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
\[(rD) \leftarrow (rA) \lor (rB)\]

Registers Altered
- rD

Latency
1 cycle
**ori**

**Logical OR with Immediate**

```plaintext
ori rD, rA, IMM
```

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

```plaintext
(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 the imm instruction for details on using 32-bit immediate values.
put

put to fsl interface

<table>
<thead>
<tr>
<th>put</th>
<th>rA, FSLx</th>
<th>put data to FSL x (blocking)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nput</td>
<td>rA, FSLx</td>
<td>put data to FSL x (non-blocking)</td>
</tr>
<tr>
<td>cput</td>
<td>rA, FSLx</td>
<td>put control to FSL x (blocking)</td>
</tr>
<tr>
<td>ncput</td>
<td>rA, FSLx</td>
<td>put control to FSL x (non-blocking)</td>
</tr>
</tbody>
</table>

Pseudocode

\[(FSLx) \leftarrow (rA)\]

if \( (n = 1) \) then

\[\text{MSR}[\text{Carry}] \leftarrow (FSLx \text{ Full bit})\]

\[(FSLx \text{ Control bit}) \leftarrow C\]

Description

MicroBlaze will write the value from register rA to the FSLx interface.

The put instruction has four variants.

The blocking versions (when ‘n’ is ‘0’) will stall microblaze until there is space available in the FSL interface. The non-blocking versions will not stall microblaze and will set carry to ‘0’ if space was available and to ‘1’ if no space was available.

The put and nput instructions will set the control bit to the FSL interface to ‘0’ and the cput and ncput instruction will set the control bit to ‘1’.

Registers Altered

- MSR[Carry]

Latency

2 cycles for non-blocking or if space is available on the FSL interface. For blocking, MicroBlaze stalls until space is available on the FSL interface.
Instructions

**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

<table>
<thead>
<tr>
<th>0 0 0 K C 1</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0</td>
<td>6 11 16 21</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

```plaintext
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, \( \text{Carry} = \text{(Borrow)} \). 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.
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 rsubikc. Both bits are set to a one for the mnemonic rsubikc.

When an rsubi instruction has bit 3 set (rsubi, 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, rsubik), 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 the imm instruction for details on using 32-bit immediate values.
rtbd

Return from Break

rtbd rA, IMM

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
PC \leftarrow (rA) + \text{sext}(\text{IMM}) \\
\text{allow following instruction to complete execution} \\
\text{MSR[BIP]} \leftarrow 0
\end{align*}
\]

Registers Altered

- PC
- MSR[BIP]

Latency

2 cycles
### rtid

#### Return from Interrupt

<table>
<thead>
<tr>
<th>1 0 1 1 0 1</th>
<th>1 0 0 0 1</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

#### 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

```markdown
PC ← (rA) + sext(IMM)
allow following instruction to complete execution
MSR[IE] ← 1
```

#### Registers Altered

- PC
- MSR[IE]

#### Latency

2 cycles
### Instructions

#### rtsd

**Return from Subroutine**

```plaintext
rtsd rA, IMM
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\text{PC} \leftarrow (\text{rA}) + \text{sext}(\text{IMM})
\]

allow following instruction to complete execution

**Registers Altered**

- PC

**Latency**

2 cycles
sb

**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
```

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\text{Addr} \leftarrow (rA) + \text{sext}(\text{IMM}) \\
\text{Mem(Addr)} \leftarrow (rD)[24:31]
\]

**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 the imm instruction for details on using 32-bit immediate values.
sext16

**Sign Extend Halfword**

```
sext16   rD, rA
```

<table>
<thead>
<tr>
<th></th>
<th>rD</th>
<th>rA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**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
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rD</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>rA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**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
Store Halfword

\[
\text{sh} \quad \text{rD, rA, rB}
\]

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>110101</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{Addr} & \leftarrow (\text{rA}) + (\text{rB}) \\
\text{Addr}[31] & \leftarrow 0 \\
\text{Mem(Addr)} & \leftarrow (\text{rD})[16:31]
\end{align*}
\]

Registers Altered

- None

Latency

- 2 cycles
**shi**

**Store Halfword Immediate**

\[ \text{shi} \quad rD, rA, \text{IMM} \]

<table>
<thead>
<tr>
<th></th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
</tbody>
</table>

**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**

\[
\text{Addr} \leftarrow (rA) + \text{sext}(\text{IMM})
\]
\[
\text{Addr}[31] \leftarrow 0
\]
\[
\text{Mem(Addr)} \leftarrow (rD)[16:31]
\]

**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 the imm instruction for details on using 32-bit immediate values.
sra  
**Shift Right Arithmetic**

sra  
rD, rA

<table>
<thead>
<tr>
<th></th>
<th>rD</th>
<th>rA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
(rD)[0] &\leftarrow (rA)[0] \\
(rD)[1:31] &\leftarrow (rA)[0:30] \\
\text{MSR}[C] &\leftarrow (rA)[31]
\end{align*}
\]

**Registers Altered**

- rD
- MSR[C]

**Latency**

1 cycle
Shift Right with Carry

**src**

\[ \text{src} \quad \text{rD, rA} \]

<table>
<thead>
<tr>
<th></th>
<th>rD</th>
<th>rA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
(rD)[0] & \leftarrow \text{MSR}[C] \\
(rD)[1:31] & \leftarrow (rA)[0:30] \\
\text{MSR}[C] & \leftarrow (rA)[31]
\end{align*}
\]

**Registers Altered**

- rD
- MSR[C]

**Latency**

1 cycle
srl

**Shift Right Logical**

\[
\text{srl} \quad rD, rA
\]

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>rD</th>
<th>rA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
(rD)[0] & \leftarrow 0 \\
(rD)[1:31] & \leftarrow (rA)[0:30] \\
\text{MSR}[C] & \leftarrow (rA)[31]
\end{align*}
\]

**Registers Altered**

- \( rD \)
- \( \text{MSR}[C] \)

**Latency**

1 cycle
**SW**

**Store Word**

\[
\text{sw} \quad rD, rA, rB
\]

<table>
<thead>
<tr>
<th>1 1 0 1 0</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 6 11 16 21 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

\[
\begin{align*}
\text{Addr} & \leftarrow (rA) + (rB) \\
\text{Addr}[30:31] & \leftarrow 00 \\
\text{Mem}(\text{Addr}) & \leftarrow (rD)[0:31]
\end{align*}
\]

**Registers Altered**

- None

**Latency**

2 cycles
**swi**  
*Store Word Immediate*

\[
\text{swi} \quad \text{rD, rA, IMM}
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>rD</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

**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**
\[
\text{Addr} \leftarrow (\text{rA}) + \text{sext}(\text{IMM})
\]
\[
\text{Addr}[30:31] \leftarrow 00
\]
\[
\text{Mem(Addr)} \leftarrow (\text{rD})[0:31]
\]

**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 the imm instruction for details on using 32-bit immediate values.
Instructions

wdc

Write to Data Cache

\textbf{wic} \quad rA, rB

<table>
<thead>
<tr>
<th></th>
<th>rA</th>
<th>rA</th>
<th>rB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

**Description**

Write into the data cache tag and data memory. Register \(rB\) contains the new data. Register \(rA\) contains the data address. Bit 30 in \(rA\) is the new valid bit and bit 31 is the new lock bit.

The instruction only works when the data cache has been disabled by clearing the Data cache enable bit in the MSR register.

**Pseudocode**

\[
\text{(DCache Tag) } \leftarrow (rA) \\
\text{(DCache Data) } \leftarrow (rB)
\]

**Registers Altered**

- None

**Latency**

1 cycle
### wic

**Write to Instruction Cache**

\[
\text{wic} \quad rA, rB
\]

<table>
<thead>
<tr>
<th>Bit</th>
<th>rA</th>
<th>rA</th>
<th>rB</th>
<th>0000 1101 1000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
</tr>
</tbody>
</table>

**Description**

Write into the instruction cache tag and data memory. Register \( rB \) contains the new instruction data. Register \( rA \) contains the instruction address. Bit 30 in \( rA \) is the new valid bit and bit 31 is the new lock bit.

The instruction only works when the instruction cache has been disabled by clearing the Instruction cache enable bit in the MSR register.

**Pseudocode**

\[
\text{(ICache Tag)} \leftarrow (rA) \\
\text{(ICache Data)} \leftarrow (rB)
\]

**Registers Altered**

- None

**Latency**

1 cycle
**xor**

**Logical Exclusive OR**

\[
xor \quad rD, rA, rB
\]

| \(1\) | \(0\) | \(0\) | \(1\) | \(0\) | \(rD\) | \(rA\) | \(rB\) | \(0\) | \(0\) | \(0\) | \(0\) | \(0\) | \(0\) | \(0\) | \(0\) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 6 | 11 | 16 | 21 | 31 |

**Description**

The contents of register \(rA\) are XORed with the contents of register \(rB\); the result is placed into register \(rD\).

**Pseudocode**

\[(rD) \leftarrow (rA) \oplus (rB)\]

**Registers Altered**

- \(rD\)

**Latency**

1 cycle
**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**

\[(rD) \leftarrow (rA) \oplus \text{sext}(\text{IMM})\]

**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 the imm instruction for details on using 32-bit immediate values.