The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/01/02</td>
<td>1.0</td>
<td>Xilinx EDK 3.1 release</td>
</tr>
<tr>
<td>03/11/03</td>
<td>2.0</td>
<td>Xilinx EDK 3.2 release</td>
</tr>
<tr>
<td>09/24/03</td>
<td>3.0</td>
<td>Xilinx EDK 6.1 release</td>
</tr>
<tr>
<td>02/20/04</td>
<td>3.1</td>
<td>Xilinx EDK 6.2 release</td>
</tr>
<tr>
<td>08/24/04</td>
<td>4.0</td>
<td>Xilinx EDK 6.3 release</td>
</tr>
<tr>
<td>09/21/04</td>
<td>4.1</td>
<td>Minor corrections for EDK 6.3 SP1 release</td>
</tr>
<tr>
<td>11/18/04</td>
<td>4.2</td>
<td>Minor corrections for EDK 6.3 SP2 release</td>
</tr>
<tr>
<td>01/20/05</td>
<td>5.0</td>
<td>Xilinx EDK 7.1 release</td>
</tr>
<tr>
<td>04/02/05</td>
<td>5.1</td>
<td>Minor corrections for EDK 7.1 SP1 release</td>
</tr>
<tr>
<td>05/09/05</td>
<td>5.2</td>
<td>Minor corrections for EDK 7.1 SP2 release</td>
</tr>
</tbody>
</table>
## Table of Contents

### Preface: About This Guide

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Contents</td>
<td>7</td>
</tr>
<tr>
<td>Additional Resources</td>
<td>7</td>
</tr>
<tr>
<td>Conventions</td>
<td>8</td>
</tr>
<tr>
<td>Typographical</td>
<td>8</td>
</tr>
<tr>
<td>Online Document</td>
<td>9</td>
</tr>
</tbody>
</table>

### Chapter 1: MicroBlaze Architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>11</td>
</tr>
<tr>
<td>Features</td>
<td>11</td>
</tr>
<tr>
<td>Data Types and Endianness</td>
<td>12</td>
</tr>
<tr>
<td>Instructions</td>
<td>13</td>
</tr>
<tr>
<td>Registers</td>
<td>19</td>
</tr>
<tr>
<td>General Purpose Registers</td>
<td>19</td>
</tr>
<tr>
<td>Special Purpose Registers</td>
<td>20</td>
</tr>
<tr>
<td>Pipeline Architecture</td>
<td>25</td>
</tr>
<tr>
<td>Branches</td>
<td>26</td>
</tr>
<tr>
<td>Memory Architecture</td>
<td>26</td>
</tr>
<tr>
<td>Reset, Interrupts, Exceptions</td>
<td>27</td>
</tr>
<tr>
<td>Hardware Exceptions</td>
<td>28</td>
</tr>
<tr>
<td>Breaks</td>
<td>28</td>
</tr>
<tr>
<td>Interrupt</td>
<td>29</td>
</tr>
<tr>
<td>User Vector (Exception)</td>
<td>30</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>30</td>
</tr>
<tr>
<td>Overview</td>
<td>30</td>
</tr>
<tr>
<td>Instruction Cache Organization</td>
<td>31</td>
</tr>
<tr>
<td>General Instruction Cache</td>
<td>31</td>
</tr>
<tr>
<td>Instruction Cache Operation</td>
<td>32</td>
</tr>
<tr>
<td>Instruction Cache Software</td>
<td>32</td>
</tr>
<tr>
<td>Data Cache</td>
<td>33</td>
</tr>
<tr>
<td>Overview</td>
<td>33</td>
</tr>
<tr>
<td>Data Cache Organization</td>
<td>33</td>
</tr>
<tr>
<td>General Data Cache Functionality</td>
<td>34</td>
</tr>
<tr>
<td>Data Cache Operation</td>
<td>35</td>
</tr>
<tr>
<td>Data Cache Software</td>
<td>35</td>
</tr>
<tr>
<td>Floating Point Unit (FPU)</td>
<td>36</td>
</tr>
<tr>
<td>Overview</td>
<td>36</td>
</tr>
<tr>
<td>Format</td>
<td>36</td>
</tr>
<tr>
<td>Rounding</td>
<td>37</td>
</tr>
<tr>
<td>Operations</td>
<td>37</td>
</tr>
<tr>
<td>Exceptions</td>
<td>38</td>
</tr>
<tr>
<td>Fast Simplex Link (FSL)</td>
<td>38</td>
</tr>
<tr>
<td>Hardware Acceleration using FSL</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 2: MicroBlaze Signal Interface Description

Overview .............................................................................. 41
Features .............................................................................. 41
MicroBlaze I/O Overview .................................................. 41
On-Chip Peripheral Bus (OPB) Interface Description ................. 44
Local Memory Bus (LMB) Interface Description ......................... 45
LMB Signal Interface ......................................................... 45
LMB Transactions ............................................................. 47
Read and Write Data Steering ............................................. 49
Fast Simplex Link (FSL) Interface Description ......................... 50
Master FSL Signal Interface ............................................... 50
Slave FSL Signal Interface ................................................ 51
FSL Transactions ............................................................. 51
Xilinx CacheLink (XCL) Interface Description ......................... 51
CacheLink Signal Interface ............................................... 53
CacheLink Transactions ................................................... 54
Debug Interface Description .............................................. 55
Trace Interface Description .............................................. 56
MicroBlaze Core Configurability ......................................... 57

Chapter 3: MicroBlaze Application Binary Interface

Scope ............................................................................. 61
Data Types ....................................................................... 61
Register Usage Conventions ............................................. 61
Stack Convention ............................................................. 63
Calling Convention ........................................................ 64
Memory Model ................................................................ 64
Small data area .................................................................. 65
Data area ......................................................................... 65
Common un-initialized area ............................................. 65
Literals or constants ......................................................... 65
Interrupt and Exception Handling ..................................... 65

Chapter 4: MicroBlaze Instruction Set Architecture

Summary ........................................................................ 67
Notation ........................................................................... 67
Formats ........................................................................... 68
Instructions ..................................................................... 68
Preface

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>
</table>
| Tutorials           | Tutorials covering Xilinx design flows, from design entry to verification and debugging.  
                      | [http://support.xilinx.com/support/techsup/tutorials/index.htm](http://support.xilinx.com/support/techsup/tutorials/index.htm) |
| Answer Browser      | Database of Xilinx solution records                                             
                      | [http://support.xilinx.com/xlnx/xil_ans_browser.jsp](http://support.xilinx.com/xlnx/xil_ans_browser.jsp) |
| Application Notes   | Descriptions of device-specific design techniques and approaches                
| Data Book           | 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  
                      | [http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp](http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp) |
### 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>ngdbuild design_name</td>
</tr>
<tr>
<td>Helvetica bold</td>
<td>Commands that you select from a menu</td>
<td>File → Open</td>
</tr>
<tr>
<td></td>
<td>Keyboard shortcuts</td>
<td>Ctrl+C</td>
</tr>
<tr>
<td>Italic font</td>
<td>Variables in a syntax statement for which you must supply values</td>
<td>ngdbuild design_name</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 not connected.</td>
</tr>
<tr>
<td>Square brackets [ ]</td>
<td>An optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.</td>
<td>ngdbuild [option_name] design_name</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>
</table>
| Vertical ellipsis| Repetitive material that has been omitted                                      | IOB #1: Name = QOUT’  
|                  |                                                                                | IOB #2: Name = CLKIN’  
|                  |                                                                                | .                                                                       |
|                  |                                                                                | .                                                                       |
|                  |                                                                                | .                                                                       |
| Horizontal ellipsis| Repetitive material that has been omitted                                    | allow block block_name  
|                  |                                                                                | loc1 loc2 ... locn;                                                    |

Online Document

The following conventions are used in this document:

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

MicroBlaze Architecture

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.

![MicroBlaze Core Block Diagram](image)

Figure 1-1: MicroBlaze Core Block Diagram

Features

The MicroBlaze embedded soft core is highly configurable, allowing users to select a specific set of features required by their design. The processor’s fixed feature set includes the following:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline
In addition to these static features the MicroBlaze processor is parametrized to allow selective enabling of additional features. Older (deprecated) versions of MicroBlaze support a subset of the optional features described in this manual. Only the latest (active) version of MicroBlaze (v4.00a) supports all optional features.

Xilinx recommends that all new designs use the active version of the MicroBlaze processor.

### Table 1-1: Configurable Feature Overview by MicroBlaze Version

<table>
<thead>
<tr>
<th>Feature</th>
<th>MicroBlaze Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v2.00a</td>
</tr>
<tr>
<td>Version Status</td>
<td>deprecated</td>
</tr>
<tr>
<td>On-chip Peripheral Bus (OPB) data side interface</td>
<td>option</td>
</tr>
<tr>
<td>On-chip Peripheral Bus (OPB) instruction side interface</td>
<td>option</td>
</tr>
<tr>
<td>Local Memory Bus (LMB) data side interface</td>
<td>option</td>
</tr>
<tr>
<td>Local Memory Bus (LMB) instruction side interface</td>
<td>option</td>
</tr>
<tr>
<td>Hardware barrel shifter</td>
<td>option</td>
</tr>
<tr>
<td>Hardware divider</td>
<td>option</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>option</td>
</tr>
<tr>
<td>Data cache</td>
<td>option</td>
</tr>
<tr>
<td>Hardware debug logic</td>
<td>option</td>
</tr>
<tr>
<td>Fast Simplex Link (FSL) interfaces</td>
<td>0-7</td>
</tr>
<tr>
<td>Machine status set and clear instructions</td>
<td>-</td>
</tr>
<tr>
<td>CacheLink support</td>
<td>-</td>
</tr>
<tr>
<td>Hardware exception support</td>
<td>-</td>
</tr>
<tr>
<td>Pattern compare instructions</td>
<td>-</td>
</tr>
<tr>
<td>Floating point unit (FPU)</td>
<td>-</td>
</tr>
<tr>
<td>Disable hardware multiplier&lt;sup&gt;1&lt;/sup&gt;</td>
<td>-</td>
</tr>
<tr>
<td>Hardware debug readable ESR and EAR</td>
<td>-</td>
</tr>
</tbody>
</table>

1. Used in Virtex-II and subsequent families, for saving MUL18 or DSP48 primitives

### Data Types and Endianness

MicroBlaze uses Big-Endian, bit-reversed format to represent data. The hardware supported data types for MicroBlaze are word, half word, and byte. The bit and byte organization for each type is shown in the following tables.

### Table 1-2: Word Data Type

<table>
<thead>
<tr>
<th>Byte address</th>
<th>n</th>
<th>n+1</th>
<th>n+2</th>
<th>n+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte label</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Byte</td>
<td>MSByte</td>
<td></td>
<td></td>
<td>LSByte</td>
</tr>
</tbody>
</table>
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 1-6 lists the MicroBlaze instruction set. Refer to Chapter 4, “MicroBlaze Instruction Set Architecture”, for more information on these instructions. Table 1-5 describes the instruction set nomenclature used in the semantics of each instruction.

Table 1-5: 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>MSR</td>
<td>Machine Status Register</td>
</tr>
</tbody>
</table>

Table 1-2: Word Data Type

| Bit label | 0 | 31 |
| Bit significance | MSBit | LSBit |

Table 1-3: Half Word Data Type

| Byte address | n | n+1 |
| Byte label | 0 | 1 |
| Byte significance | MSByte | LSByte |

| Bit label | 0 | 15 |
| Bit significance | MSBit | LSBit |

Table 1-4: Byte Data Type

| Byte address | n |
| Byte label | 0 |
| Byte significance | MSByte |

| Bit label | 0 | 7 |
| Bit significance | MSBit | LSBit |

Table 1-6: MicroBlaze Instruction Set Architecture
Table 1-5: Instruction Set Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESR</td>
<td>Exception Status Register</td>
</tr>
<tr>
<td>EAR</td>
<td>Exception Address Register</td>
</tr>
<tr>
<td>FSR</td>
<td>Floating Point Unit Status Register</td>
</tr>
<tr>
<td>PC</td>
<td>Execute stage Program Counter</td>
</tr>
<tr>
<td>$x[y]$</td>
<td>Bit $y$ of register $x$</td>
</tr>
<tr>
<td>$x[y:z]$</td>
<td>Bit range $y$ to $z$ of register $x$</td>
</tr>
<tr>
<td>$\bar{x}$</td>
<td>Bit inverted value of register $x$</td>
</tr>
<tr>
<td>Imm</td>
<td>16 bit immediate value</td>
</tr>
<tr>
<td>Imm$x$</td>
<td>$x$ bit immediate value</td>
</tr>
<tr>
<td>FSL$x$</td>
<td>3 bit Fast Simplex Link (FSL) port designator where $x$ is the port number</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>
<tr>
<td>&amp;</td>
<td>Concatenate. E.g. “0000100 &amp; Imm7” is the concatenation of the fixed field “0000100” and a 7 bit immediate value.</td>
</tr>
<tr>
<td>signed</td>
<td>Operation performed on signed integer data type</td>
</tr>
<tr>
<td>unsigned</td>
<td>Operation performed on unsigned integer data type</td>
</tr>
<tr>
<td>float</td>
<td>Operation performed on floating point data type</td>
</tr>
</tbody>
</table>

Table 1-6: MicroBlaze Instruction Set Summary

<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>ADD Rd,Ra,Rb</td>
<td>000000</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra</td>
</tr>
<tr>
<td>RSUB Rd,Ra,Rb</td>
<td>000001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + 1</td>
</tr>
<tr>
<td>ADDC Rd,Ra,Rb</td>
<td>000010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + C</td>
</tr>
<tr>
<td>RSUBC Rd,Ra,Rb</td>
<td>000011</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + C</td>
</tr>
<tr>
<td>ADDK Rd,Ra,Rb</td>
<td>000100</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra</td>
</tr>
<tr>
<td>RSUBK Rd,Ra,Rb</td>
<td>000101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + 1</td>
</tr>
<tr>
<td>ADDKC Rd,Ra,Rb</td>
<td>000110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + C</td>
</tr>
<tr>
<td>RSUBKC Rd,Ra,Rb</td>
<td>000111</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + C</td>
</tr>
<tr>
<td>CMP Rd,Ra,Rb</td>
<td>000101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000001</td>
<td>Rd := Rb + Ra + 1 (signed)</td>
</tr>
<tr>
<td>CMPU Rd,Ra,Rb</td>
<td>000101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000011</td>
<td>Rd := Rb + Ra + 1 (unsigned)</td>
</tr>
</tbody>
</table>
### Table 1-6: 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>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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td>MUL Rd,Ra,Rb</td>
<td>010000</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Ra * Rb</td>
</tr>
<tr>
<td>BSRL Rd,Ra,Rb</td>
<td>010001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Ra &gt;&gt; Rb</td>
</tr>
<tr>
<td>BSRA Rd,Ra,Rb</td>
<td>010010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>01000000000</td>
<td>Rd := Ra[0], (Ra &gt;&gt; Rb)</td>
</tr>
<tr>
<td>BSLL Rd,Ra,Rb</td>
<td>010011</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>10000000000</td>
<td>Rd := Ra &lt;&lt; Rb</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>
</tr>
<tr>
<td>BSRLI Rd,Ra,Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm5</td>
<td>Rd := Ra &gt;&gt; Imm5</td>
<td></td>
</tr>
<tr>
<td>BSRAI Rd,Ra,Imm</td>
<td>011010</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm5</td>
<td>Rd := Ra[0], (Ra &gt;&gt; Imm5)</td>
<td></td>
</tr>
<tr>
<td>BSLLI Rd,Ra,Imm</td>
<td>011011</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm5</td>
<td>Rd := Ra &lt;&lt; Imm5</td>
<td></td>
</tr>
<tr>
<td>IDIV Rd,Ra,Rb</td>
<td>010100</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb/Ra, signed</td>
</tr>
<tr>
<td>IDIVU Rd,Ra,Rb</td>
<td>010101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000010</td>
<td>Rd := Rb/Ra, unsigned</td>
</tr>
<tr>
<td>FADD Rd,Ra,Rb</td>
<td>010110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb+Ra, float1</td>
</tr>
<tr>
<td>FRSUB Rd,Ra,Rb</td>
<td>010111</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00010000000</td>
<td>Rd := Rb-Ra, float1</td>
</tr>
<tr>
<td>FMUL Rd,Ra,Rb</td>
<td>010110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00100000000</td>
<td>Rd := Rb*Ra, float1</td>
</tr>
<tr>
<td>FDIV Rd,Ra,Rb</td>
<td>010111</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00110000000</td>
<td>Rd := Rb/Ra, float1</td>
</tr>
<tr>
<td>FCMP.UN Rd,Ra,Rb</td>
<td>010100</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>01000000000</td>
<td>Rd := 1 when (Rb = NaN or Ra = NaN, float1) else Rd := 0</td>
</tr>
<tr>
<td>FCMP.LT Rd,Ra,Rb</td>
<td>010110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>01000100000</td>
<td>Rd := 1 when (Rb &lt; Ra, float1) else Rd := 0</td>
</tr>
<tr>
<td>FCMP.EQ Rd,Ra,Rb</td>
<td>010111</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>01000100000</td>
<td>Rd := 1 when (Rb = Ra, float1) else Rd := 0</td>
</tr>
<tr>
<td>FCMP.LE Rd,Ra,Rb</td>
<td>010110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>01000110000</td>
<td>Rd := 1 when (Rb &lt;= Ra, float1) else Rd := 0</td>
</tr>
<tr>
<td>FCMP.GT Rd,Ra,Rb</td>
<td>010111</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>01001000000</td>
<td>Rd := 1 when (Rb &gt; Ra, float1) else Rd := 0</td>
</tr>
</tbody>
</table>
### Table 1-6: MicroBlaze Instruction Set Summary (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>A</th>
<th>B</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type A</strong></td>
<td>0-5</td>
<td>6-10</td>
<td>11-15</td>
</tr>
<tr>
<td>FCMP.NE Rd,Ra,Rb</td>
<td>010110</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>FCMP.GE Rd,Ra,Rb</td>
<td>010110</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>GET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000</td>
</tr>
<tr>
<td>PUT Ra,FSLx</td>
<td>011011</td>
<td>Ra</td>
<td>00000</td>
</tr>
<tr>
<td>NGET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000</td>
</tr>
<tr>
<td>NPUT Ra,FSLx</td>
<td>011011</td>
<td>Ra</td>
<td>00000</td>
</tr>
<tr>
<td>CGET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000</td>
</tr>
<tr>
<td>CPUT Ra,FSLx</td>
<td>011011</td>
<td>Ra</td>
<td>00000</td>
</tr>
<tr>
<td>NCGET Rd,FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000</td>
</tr>
<tr>
<td>NCPUT Ra,FSLx</td>
<td>011011</td>
<td>Ra</td>
<td>00000</td>
</tr>
<tr>
<td>OR Rd,Ra,Rb</td>
<td>100000</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>AND Rd,Ra,Rb</td>
<td>100001</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>XOR Rd,Ra,Rb</td>
<td>100010</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>ANDN Rd,Ra,Rb</td>
<td>100011</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>PCMPBF Rd,Ra,Rb</td>
<td>100000</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>PCMPEQ Rd,Ra,Rb</td>
<td>100010</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>PCMPNE Rd,Ra,Rb</td>
<td>100011</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>SRA Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
</tr>
<tr>
<td>SRC Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
</tr>
</tbody>
</table>
### Instructions

**Table 1-6: 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>Type B</td>
<td>0-5</td>
<td>6-10</td>
<td>11-15</td>
<td>16-31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL Rd,Ra</td>
<td>100100</td>
<td>Rd</td>
<td>Ra</td>
<td>0000000001000001</td>
<td>Rd := 0, (Ra &gt;&gt; 1) C := Ra[31]</td>
<td></td>
</tr>
<tr>
<td>WIC Ra,Rb</td>
<td>100100</td>
<td>Ra</td>
<td>Ra</td>
<td>Rb</td>
<td>01101000</td>
<td>ICache_Tag := Ra, ICache_Data := Rb</td>
</tr>
<tr>
<td>WDC Ra,Rb</td>
<td>100100</td>
<td>Ra</td>
<td>Ra</td>
<td>Rb</td>
<td>01101000</td>
<td>DCache_Tag := Ra, DCache_Data := Rb</td>
</tr>
<tr>
<td>MTS Sd,Ra</td>
<td>100101</td>
<td>00000</td>
<td>Ra</td>
<td>110000000000 &amp; Sd</td>
<td>Sd := Ra, where Sd=001 is MSR, and Sd=111 is FSR</td>
<td></td>
</tr>
<tr>
<td>MFS Rd,Sa</td>
<td>100101</td>
<td>Rd</td>
<td>00000</td>
<td>100000000000 &amp; Sa</td>
<td>Rd := Sa, where Sa=000 is PC, 001 is MSR, 011 is EAR, 101 is ESR, and 111 is FSR</td>
<td></td>
</tr>
<tr>
<td>MSRCLR Rd,Imm</td>
<td>100101</td>
<td>Rd</td>
<td>00001</td>
<td>00 &amp; Imm14</td>
<td>Rd := MSR MSR := MSR and Imm14</td>
<td></td>
</tr>
<tr>
<td>MSRSET Rd,Imm</td>
<td>100101</td>
<td>Rd</td>
<td>00000</td>
<td>00 &amp; Imm14</td>
<td>Rd := MSR MSR := MSR or Imm14</td>
<td></td>
</tr>
<tr>
<td>BR Rb</td>
<td>100110</td>
<td>00000</td>
<td>00000</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := PC + Rb</td>
</tr>
<tr>
<td>BRD Rb</td>
<td>100110</td>
<td>00000</td>
<td>10000</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := PC + Rb</td>
</tr>
<tr>
<td>BRLD Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>10100</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := PC + Rb Rd := PC</td>
</tr>
<tr>
<td>BRA Rb</td>
<td>100110</td>
<td>00000</td>
<td>01000</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := Rb</td>
</tr>
<tr>
<td>BRAD Rb</td>
<td>100110</td>
<td>00000</td>
<td>11000</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := Rb</td>
</tr>
<tr>
<td>BRALD Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>11100</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := Rb; Rd := PC</td>
</tr>
<tr>
<td>BRK Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>01100</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := Rb; Rd := PC MSR[BIP] := 1</td>
</tr>
<tr>
<td>BEQ Ra,Rb</td>
<td>10011</td>
<td>00000</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BNE Ra,Rb</td>
<td>10011</td>
<td>00001</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra /= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLT Ra,Rb</td>
<td>10011</td>
<td>00010</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &lt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLE Ra,Rb</td>
<td>10011</td>
<td>00011</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &lt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGT Ra,Rb</td>
<td>10011</td>
<td>00100</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &gt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGE Ra,Rb</td>
<td>10011</td>
<td>00101</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &gt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BEQD Ra,Rb</td>
<td>10011</td>
<td>10000</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BEQD Ra,Rb</td>
<td>10011</td>
<td>10000</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BNE Ra,Rb</td>
<td>10011</td>
<td>10000</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra /= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLT Ra,Rb</td>
<td>10011</td>
<td>10001</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &lt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLE Ra,Rb</td>
<td>10011</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>BGT Ra,Rb</td>
<td>10011</td>
<td>10011</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &gt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGE Ra,Rb</td>
<td>10011</td>
<td>10100</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000</td>
<td>if Ra &gt;= 0: PC := PC + Rb</td>
</tr>
</tbody>
</table>
### MicroBlaze Instruction Set Summary (Continued)

<table>
<thead>
<tr>
<th>Table 1-6: MicroBlaze Instruction Set Summary (Continued)</th>
</tr>
</thead>
</table>

<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>BGTD Ra,Rb</td>
<td>$100111$</td>
<td>$10100$</td>
<td>Ra</td>
<td>Rb</td>
<td>$00000000000$</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>$00000000000$</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>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>RTSD 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>RTED Ra,Imm</td>
<td>$101101$</td>
<td>$10010$</td>
<td>Ra</td>
<td>Imm</td>
<td>PC := Ra + s(Imm) MSR[IE] := 1, MSR[EIP] := 0 ESR := 0</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>BRI Imm</td>
<td>$101110$</td>
<td>$00000$</td>
<td>$00000$</td>
<td>Imm</td>
<td>PC := PC + s(Imm)</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>BRLID 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>$11000$</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>$00100$</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>$00001$</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>$10000$</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>
</tbody>
</table>
MicroBlaze is a fully orthogonal architecture. It has thirty-two 32-bit general purpose registers and five 32-bit special purpose registers.

### General Purpose Registers

The thirty-two 32-bit General Purpose Registers are numbered R0 through R31. The register file is reset on bit stream download. It is not reset by the external reset inputs: reset and debug_rst.

---

**Table 1-6: MicroBlaze Instruction Set Summary (Continued)**

<table>
<thead>
<tr>
<th>Type</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>BGTID 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>Addr := Ra + Rb Rd[0:23] := 0, Rd[24:31] := *Addr</td>
<td></td>
</tr>
<tr>
<td>LHU Rd,Ra,Rb</td>
<td>110001</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>Addr := Ra + Rb Rd[0:15] := 0, Rd[16:31] := *Addr</td>
<td></td>
</tr>
<tr>
<td>LW Rd,Ra,Rb</td>
<td>110010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>Addr := Ra + Rb Rd := *Addr</td>
<td></td>
</tr>
<tr>
<td>SB Rd,Ra,Rb</td>
<td>110100</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>Addr := Ra + Rb *Addr := Rd[24:31]</td>
<td></td>
</tr>
<tr>
<td>SH Rd,Ra,Rb</td>
<td>110101</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>Addr := Ra + Rb *Addr := Rd[16:31]</td>
<td></td>
</tr>
<tr>
<td>SW Rd,Ra,Rb</td>
<td>110110</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>Addr := Ra + Rb *Addr := Rd</td>
<td></td>
</tr>
<tr>
<td>LBUI Rd,Ra,Imm</td>
<td>111000</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Addr := Ra + s(Imm) Rd[0:23] := 0, Rd[24:31] := *Addr</td>
<td></td>
</tr>
<tr>
<td>LHUI Rd,Ra,Imm</td>
<td>111001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Addr := Ra + s(Imm) Rd[0:15] := 0, Rd[16:31] := *Addr</td>
<td></td>
</tr>
<tr>
<td>LWI Rd,Ra,Imm</td>
<td>111010</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Addr := Ra + s(Imm) Rd := *Addr</td>
<td></td>
</tr>
<tr>
<td>SBI Rd,Ra,Imm</td>
<td>111100</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Addr := Ra + s(Imm) *Addr := Rd[24:31]</td>
<td></td>
</tr>
<tr>
<td>SHI Rd,Ra,Imm</td>
<td>111101</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Addr := Ra + s(Imm) *Addr := Rd[16:31]</td>
<td></td>
</tr>
<tr>
<td>SWI Rd,Ra,Imm</td>
<td>111110</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Addr := Ra + s(Imm) *Addr := Rd</td>
<td></td>
</tr>
</tbody>
</table>

1. Due to the many different corner cases involved in floating point arithmetic, only the normal behavior is described. A full description of the behavior can be found in: Chapter 4, “MicroBlaze Instruction Set Architecture.”
Please refer to Table 3-2 for software conventions on general purpose register usage.

**Special Purpose Registers**

**Program Counter (PC)**

The Program Counter is the 32-bit address of the execution instruction. It can be read with an MFS instruction. It cannot be written to using an MTS instruction. When used with the MFS instruction the PC register is specified by setting Sa = 000, or Sa = rpc.
Machine Status Register (MSR)

The Machine Status Register contains control and status bits for the processor. It can be read with an MFS instruction. When reading the MSR, bit 29 is replicated in bit 0 as the carry copy. MSR can be written using either an MTS instruction or the dedicated MSRSET and MSRCLR instructions.

When writing to the MSR, some of the bits will take effect immediately (e.g. Carry) and the remaining bits take effect one clock cycle later. Any value written to bit 0 is discarded. When used with an MTS or MFS instruction the MSR is specified by setting $s_x = 001$, or $s_x = \text{rmsr}$.

Table 1-8: Program Counter (PC)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>PC</td>
<td>Program Counter Address of executing instruction, i.e. “mfs r2 rpc” will store the address of the mfs instruction itself in R2</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Figure 1-3: PC

Figure 1-4: MSR
### Machine Status Register (MSR)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC</td>
<td>Arithmetic Carry Copy</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy of the Arithmetic Carry (bit 29). CC is always the same as bit C.</td>
<td></td>
</tr>
<tr>
<td>1:21</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>EIP</td>
<td>Exception In Progress</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 No hardware exception in progress</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Hardware exception in progress</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>EE</td>
<td>Exception Enable</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Hardware exceptions disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Hardware exceptions enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>DCE</td>
<td>Data Cache Enable</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Data Cache is Disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Data Cache is Enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>DZ</td>
<td>Division by Zero(^1)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 No division by zero has occurred</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Division by zero has occurred</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>ICE</td>
<td>Instruction Cache Enable</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Instruction Cache is Disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Instruction Cache is Enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>FSL</td>
<td>FSL Error</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 FSL get/put had no error</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 FSL get/put had mismatch in instruction type and value type</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>BIP</td>
<td>Break in Progress</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 No Break in Progress</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Break in Progress</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source of break can be software break instruction or hardware break from</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ext_Brk or Ext_NM_Brk pin.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
</tbody>
</table>
Table 1-9: Machine Status Register (MSR) (Continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
</table>
| 29   | C    | Arithmetic Carry  
|      |      | 0 No Carry (Borrow)  
|      |      | 1 Carry (No Borrow)  
|      |      | Read/Write | 0 |
| 30   | IE   | Interrupt Enable  
|      |      | 0 Interrupts disabled  
|      |      | 1 Interrupts enabled | 0 |
| 31   | BE   | Buslock Enable²  
|      |      | 0 Buslock disabled on data-side OPB  
|      |      | 1 Buslock enabled on data-side OPB | 0 |
|      |      | Buslock Enable does not affect  
|      |      | operation of IXCL, DXCL, ILMB,  
|      |      | DLMB, or IOPB.  
|      |      | Read/Write |

1. This bit is only used for integer divide-by-zero signaling. There is a floating point equivalent in the FSR. The DZ-bit will flag divide by zero conditions regardless if the processor is configured with exception handling or not.

Exception Address Register (EAR)

The Exception Address Register stores the full load/store address that caused the exception. For an unaligned access exception that means the unaligned access address, and for an DOPB exception, the failing OPB data access address. The contents of this register is undefined for all other exceptions. When read with the MFS instruction the EAR is specified by setting Sa = 011, or Sa = rear.

---

Table 1-10: Exception Address Register (EAR)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>EAR</td>
<td>Exception Address Register</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

---

Figure 1-5: EAR
Exception Status Register (ESR)

The Exception Status Register contains status bits for the processor. When read with the MFS instruction the ESR is specified by setting Sa = 101, or Sa = resr.

![ESR Diagram](image)

**Table 1-11: Exception Status Register (ESR)**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:19</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20:26</td>
<td>ESS</td>
<td>Exception Specific Status</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For details refer to Table 1-12.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>27:31</td>
<td>EC</td>
<td>Exception Cause</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00001 Unaligned data access exception</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00010 Illegal op-code exception</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00011 Instruction bus error exception</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00100 Data bus error exception</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00101 Divide by zero exception</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00110 Floating point unit exception</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1-12: Exception Specific Status (ESS)**

<table>
<thead>
<tr>
<th>Exception Cause</th>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unaligned Data Access</td>
<td>20</td>
<td>W</td>
<td>Word Access Exception</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 unaligned halfword access</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 unaligned word access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>S</td>
<td>Store Access Exception</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 unaligned load access</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 unaligned store access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22:26</td>
<td>Rx</td>
<td>Source/Destination Register</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>General purpose register used as source (Store) or destination (Load) in unaligned access</td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Architecture

MicroBlaze uses a pipelined instruction execution. The pipeline is divided into three stages:

- Fetch
- Decode

Floating Point Status Register (FSR)

The Floating Point Status Register contains status bits for the floating point unit. It can be read with an MFS, and written with an MTS instruction. When read or written, the register is specified by setting Sa = 111, or Sa = rfsr.

Table 1-12: Exception Specific Status (ESS)

<table>
<thead>
<tr>
<th>Exception Cause</th>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal Instruction</td>
<td>20:26</td>
<td>Reserved</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Instruction bus error</td>
<td>20:26</td>
<td>Reserved</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Data bus error</td>
<td>20:26</td>
<td>Reserved</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Divide by zero</td>
<td>20:26</td>
<td>Reserved</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Floating point unit</td>
<td>20:26</td>
<td>Reserved</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1-13: Floating Point Status Register (FSR)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:26</td>
<td>Reserved</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>IO</td>
<td>Invalid operation</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>DZ</td>
<td>Divide-by-zero</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>OF</td>
<td>Overflow</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>UF</td>
<td>Underflow</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>DO</td>
<td>Denormalized operand error</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1-7: FSR
• Execute

For most instructions, each stage takes one clock cycle to complete. Consequently, it takes three clock cycles for a specific instruction to complete, while one instruction is completed on every cycle. A few instructions require multiple clock cycles in the execute stage to complete. This is achieved by stalling the pipeline.

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
<th>cycle 5</th>
<th>cycle 6</th>
<th>cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 2</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Execute</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 3</td>
<td>Fetch</td>
<td>Decode</td>
<td>Stall</td>
<td>Stall</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When executing from slower memory, instruction fetches may take multiple cycles. This additional latency will directly affect the efficiency of the pipeline. MicroBlaze implements an instruction prefetch buffer that reduces the impact of such multi-cycle instruction memory latency. While the pipeline is stalled by a multi-cycle instruction in the execution stage the prefetch buffer continues to load sequential instructions. Once the pipeline resumes execution the fetch stage can load new instructions directly from the prefetch buffer rather than having to wait for the instruction memory access to complete.

Branches

Normally the instructions in the fetch and decode stages (as well as prefetch buffer) are flushed when executing a taken branch. The fetch pipeline stage is then reloaded with a new instruction from the calculated branch address. A taken branch in MicroBlaze takes three clock cycles to execute, two of which are required for refilling the pipeline. To somewhat mitigate this latency overhead, MicroBlaze supports branches with delay slots.

Delay Slots

When executing a taken branch with delay slot, only the fetch pipeline stage in MicroBlaze is flushed. The instruction in the decode stage (branch delay slot) is allowed to complete. This technique effectively reduces the branch penalty from two clock cycles to one. Branch instructions with delay slots have a D appended to the instruction mnemonic. For example, the BNE instruction will not execute the subsequent instruction (does not have a delay slot), whereas BNED will execute the next instruction before control is transferred to the branch location.

Memory Architecture

MicroBlaze has a Harvard memory architecture, i.e. instruction and data accesses are done in separate address spaces. Each address space has a 32 bit range (i.e. handles up to 4 GByte of instructions and data memory respectively). The instruction and data memory ranges can be made to overlap by mapping them both to the same physical memory. This is useful for software debugging.

Both instruction and data interfaces of MicroBlaze are 32 bit wide and use big endian, bit-reversed format. MicroBlaze supports word, halfword, and byte accesses to data memory. Data accesses must be aligned (i.e. word accesses must be on word boundaries, halfword on halfword boundaries), unless the processor is configured to support unaligned exceptions. All instruction accesses must be word aligned.
MicroBlaze does not separate between data accesses to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory accesses: Local Memory Bus (LMB), On-Chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). The memory maps on these interfaces are mutually exclusive.

MicroBlaze uses speculative accesses to reduce latency over slower memory interfaces. This means that the processor will initiate each memory access on all available interfaces. When the correct interface has been resolved (i.e. matched against the interface address map) in the subsequent cycle, the other accesses are aborted.

For details on these different memory interfaces please refer to Chapter 2, “MicroBlaze Signal Interface Description”.

Reset, Interrupts, Exceptions and Break

MicroBlaze supports reset, interrupt, user exception, break and hardware exceptions. The following section describes the execution flow associated with each of these events.

The relative priority starting with the highest is:

1. Reset
2. Hardware Exception
3. Non-maskable Break
4. Break
5. Interrupt
6. User Vector (Exception)

Table 1-14 defines the memory address locations of the associated vectors and the hardware enforced register file locations for return address. Each vector allocates two addresses to allow full address range branching (requires an IMM followed by a BRAI instruction).

<table>
<thead>
<tr>
<th>Event</th>
<th>Vector Address</th>
<th>Register File Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0x00000000 - 0x00000004</td>
<td>-</td>
</tr>
<tr>
<td>User Vector (Exception)</td>
<td>0x00000008 - 0x0000000C</td>
<td>-</td>
</tr>
<tr>
<td>Interrupt</td>
<td>0x00000010 - 0x00000014</td>
<td>R14</td>
</tr>
<tr>
<td>Break: Non-maskable hardware</td>
<td>0x00000018 - 0x0000001C</td>
<td>R16</td>
</tr>
<tr>
<td>Break: Hardware</td>
<td>0x0000000018 - 0x00000001C</td>
<td>R17</td>
</tr>
<tr>
<td>Break: Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Exception</td>
<td>0x00000020 - 0x00000024</td>
<td>R17</td>
</tr>
</tbody>
</table>
Reset

When a Reset or Debug_Rst\(^1\) occurs, MicroBlaze will flush the pipeline and start fetching instructions from the reset vector (address 0x0). Both external reset signals are active high, and should be asserted for a minimum of 16 cycles.

Equivalent Pseudocode

\[
\begin{align*}
PC & \leftarrow 0x00000000 \\
MSR & \leftarrow 0 \\
EAR & \leftarrow 0 \\
ESR & \leftarrow 0
\end{align*}
\]

Hardware Exceptions

MicroBlaze can be configured to trap the following internal error conditions: illegal instruction, instruction and data bus error, unaligned access, and divide by zero. When configured with a hardware floating point unit, it can also trap the following floating point specific exceptions: underflow, overflow, division-by-zero, invalid operation, and denormalized operand error.

On a hardware exception MicroBlaze will flush the pipeline and branch to the hardware exception vector (address 0x20). The exception will also load the decode stage program counter value into the general purpose register R17. The execution stage instruction in the exception cycle is not executed.

The EE and EIP bits in MSR are automatically reverted when executing the RTED instruction.

An exception in a branch delay-slot instruction is non-recoverable. For this reason the MicroBlaze compiler will never place a load/store instruction, or a floating point instruction in a branch delay slot, because these require recoverable exceptions (e.g. unaligned access and underflow).

Equivalent Pseudocode

\[
\begin{align*}
r17 & \leftarrow PC \\
PC & \leftarrow 0x00000020 \\
MSR[EE] & \leftarrow 0 \\
MSR[EIP] & \leftarrow 1 \\
ESR[EC] & \leftarrow \text{exception specific value} \\
ESR[ESS] & \leftarrow \text{exception specific value} \\
EAR & \leftarrow \text{exception specific value} \\
FSR & \leftarrow \text{exception specific value}
\end{align*}
\]

Breaks

There are two kinds of breaks:

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

---

1. Reset input controlled by the XMD debugger via MDM
Hardware Breaks

Hardware breaks are performed by asserting the external break signal (i.e. the Ext_BRK and Ext_NM_BRK input ports). On a break the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the break vector (address 0x18). The break return address (the PC associated with the instruction in the decode stage at the time of the break) is automatically loaded into general purpose register R16. MicroBlaze also sets the Break In Progress (BIP) flag in the Machine Status Register (MSR).

A normal hardware break (i.e the Ext_BRK input port) is only handled when there is no break in progress (i.e MSR[BIP] is set to 0). The Break In Progress flag also disables interrupts and exceptions. A non-maskable break (i.e the Ext_NM_BRK input port) will always be handled immediately.

The BIP bit in the MSR is automatically cleared when executing the RTBD instruction.

Software Breaks

To perform a software break, use the brake and brki instructions. Refer to Chapter 4, “MicroBlaze Instruction Set Architecture” for detailed information on software breaks.

Latency

The time it will take MicroBlaze to enter a break service routine from the time the break occurs, depends on the instruction currently in the execution stage.

Table 1-15 shows the different scenarios for breaks. The cycle count includes the cycles for completing the current instruction, and branching to the service routine vector.

Equivalent Pseudocode

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

Interrupt

MicroBlaze supports one external interrupt source (connecting to the Interrupt input port). The processor will only react to interrupts if the Interrupt Enable (IE) bit in the Machine Status Register (MSR) is set to 1. On an interrupt the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the interrupt vector (address 0x10). The interrupt return address (the PC associated with the instruction in the decode stage at the time of the interrupt) is automatically loaded into general purpose register R14. In addition, the processor also disables future interrupts by clearing the IE bit in the MSR. The IE bit is automatically set again when executing the RTID instruction.

Interrupts are ignored by the processor if the break in progress (BIP) bit in the MSR is set 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 1-15 shows the different scenarios for interrupts. The cycle count includes the cycles for completing the current instruction, and branching to the service routine vector.

**Table 1-15: Interrupt and Break latencies**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>LMB Memory Vector</th>
<th>OPB Memory Vector</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 divider¹</td>
<td>37 cycles</td>
<td>39 cycles</td>
</tr>
</tbody>
</table>

¹. This does not take into account blocking FSL instructions which can stall indefinitely

**Equivalent Pseudocode**

```
r14 ← PC
PC ← 0x00000010
MSR[IE] ← 0
```

**User Vector (Exception)**

The user exception vector is located at address 0x8. A user exception is caused by inserting a ‘BRALID Rx, 0x8’ instruction in the software flow. Although Rx could be any general purpose register Xilinx recommends using R15 for storing the user exception return address, and to use the RTSD instruction to return from the user exception handler.

**Pseudocode**

```
rx ← PC
PC ← 0x00000008
```

**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:

- Direct mapped (1-way associative)
- User selectable cacheable memory area
- Configurable cache and tag size
- Configurable caching over OPB or CacheLink
- 4 word cache-line (only with CacheLink)
- Individual cache line lock capability
- Cache on and off controlled using a bit in the MSR
- Instructions to write to the instruction cache
- Memory is organized into a cacheable and a non-cacheable segment
Instruction Cache Organization

MicroBlaze can be configured to cache instructions over either the OPB interface or the dedicated Xilinx CacheLink interface. The choice is determined by the setting of the two parameters: C_USE_ICACHE and C_ICACHE_USE_FSL (for details see: “MicroBlaze Core Configurability” in Chapter 2). The main differences between the two solutions are:

- Caching over CacheLink uses 4 word cache lines (critical word first). OPB caches use single word cache lines
- CacheLink uses dedicated interface, instead of the OPB interface, for memory accesses. This reduces the traffic on the OPB
- The CacheLink interface requires a specialized memory controller interface. The OPB interface uses standard OPB memory controllers

For details on the CacheLink interface on MicroBlaze, please refer to “Xilinx CacheLink (XCL) Interface Description” in Chapter 2.

General Instruction Cache Functionality

When the instruction cache is used, the memory address space in 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.

Cacheable instruction addresses are further split into two segments: a cache word address segment, and a tag address segment. The size of the two segments is configured through MicroBlaze parameters. The size\(^{(1)}\) of the cache word address can be between 9 to 14 bits. This results in cache sizes ranging from 2kB to 64 kB. The tag address should be sized so that it matches the complete range of cacheable memory in the design. E.g. assuming a configuration of C_ICACHE_BASEADDR=0x00300000, C_ICACHE_HIGHADDR=0x0030ffff, and C_CACHE_BYTE_SIZE=4096; the cacheable...
byte address range is 16 bits, and the cache byte address range is 12 bits (i.e. a 10 bit cache word address), thus the required address tag is: 16-12=4 bits.

The total number of primitives required in the example above is: 2 RAMB16 for storing the 1024 instructions, and 1 RAMB16 for tag and status (4 bits of tag + 1 valid bit + 1 lock bit), i.e. a total of 3 RAMB16 primitives.

**Instruction Cache Operation**

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 controller ignores the instruction and allows the OPB to fulfill the request. If the address is cacheable, a lookup is performed on the tag memory to check if the requested address is currently cached. The lookup is successful if: the valid bit is set, and the tag address matches the instruction address tag segment.

If the instruction is in the cache (see Figure 1-8), the cache controller will drive the ready signal (Cache_Hit) and the cached instruction (Cache_instruction_data) to the pipeline.

On a cache miss, the cache controller will wait until the missing instruction has been retrieved (either over the OPB, or the CacheLink interface depending on the caching scheme used), and then store it and its associated tag bits in the corresponding cache location.

**Instruction Cache Software Support**

**MSR Bit**

The ICE bit 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.

1. The size of the cache is FPGA architecture dependent. The MicroBlaze instruction cache can be configured to use up to 32 RAMB primitives for data. The actual cache size therefore depends on the RAMB size in the targeted architecture. For older architectures: Virtex, VirtexE, Spartan and Spartan2E the RAMB size is only 512B.
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. For a detailed description, please refer to Chapter 4, “MicroBlaze Instruction Set Architecture”.

**HW Debug Logic**

The HW debug logic can be used to perform a similar operation to 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 a cacheline can result in decreased cache performance, because the lock prevents the caching of all other instructions that map to the same cache location. In most cases adding instruction side LMB memory is a better choice for guaranteed access to certain code segments than cacheline locking. The access latency of LMB BRAM is the same as for a cache hit.

---

**Data Cache**

**Overview**

MicroBlaze may be used with an optional data cache for improved performance. The cached memory range must not include addresses in the LMB address range.

The data cache has the following features

- Direct mapped (1-way associative)
- Write-through
- User selectable cacheable memory area
- Configurable cache size and tag size
- Configurable caching over OPB or CacheLink
- 4 word cache-line (only with CacheLink)
- Individual cache line lock capability
- Cache on and off controlled using a bit in the MSR
- Instructions to write to the data cache
- Memory is organized into a cacheable and a non-cacheable segment

**Data Cache Organization**

MicroBlaze can be configured to cache data over either the OPB interface, or the dedicated Xilinx CacheLink interface. The choice is determined by the setting of the two parameters: C_USE_DCACHE and C_DCACHE_USE_FSL (for details see: “MicroBlaze Core Configurability” in Chapter 2). The main differences between the two solutions are:
• Caching over CacheLink uses 4 word cache lines (critical word first) for read misses. OPB caches use single word cache lines.
• CacheLink allows posted write accesses on write-misses. OPB caches require the write access to be completed before execution is resumed.
• CacheLink uses dedicated interface, instead of the OPB interface, for memory accesses. This reduces the traffic on the OPB.
• The CacheLink interface requires a specialized memory controller interface. The OPB interface uses standard OPB memory controllers.

For details on the CacheLink interface on MicroBlaze, please refer to “Xilinx CacheLink (XCL) Interface Description” in Chapter 2.

General Data Cache Functionality

When the data cache is used, the memory address space is 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. All other addresses are non-cacheable.

All cacheable data addresses are further split into two segments: a cache word address segment and a tag address segment. The size of the two segments can be configured by the user. The size of the cache word address can be between 11 to 14 bits. This results in a cache sizes ranging from 8 kB to 64 kB. The tag address should be sized so that it matches the complete range of cacheable memory in the design. E.g. assuming a configuration of C_DCACHE_BASEADDR=0x00400000, C_DCACHE_HIGHADDR=0x00403fff, and C_DCACHE_BYTE_SIZE=2048; the cacheable byte address range is 14 bits, and the cache

1. The size of the cache is FPGA architecture dependent. The MicroBlaze data cache can be configured to use between 4 and 32 RAMB primitives. The actual cache size therefore depends on the RAMB size in the targeted architecture. For older architectures: Virtex, VirtexE and Spartan2E the RAMB size is only 512B.
Data 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, i.e. the cache is not updated on a write miss.

When MicroBlaze executes a load instruction, the address is first checked to see if the address is within the cacheable area, and if so, whether the address is currently cached. In that case, the data is retrieved from the cache.

On a read request, if the read data is in the cache (see Figure 1-10), the cache will drive the ready signal (Cache_Hit) for MicroBlaze and the data for the address (Cache_data).

On a cache miss, the cache controller will wait until the missing data has been retrieved (either over the OPB, or the CacheLink interface depending on the caching scheme used), and then store it and its associated tag bits in the corresponding cache location.

Data Cache Software Support

MSR Bit

The DCE bit 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 cache cannot be turned on or off from an interrupt handler routine as the changes to the MSR are 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. For a detailed description, please refer to Chapter 4, “MicroBlaze Instruction Set Architecture”.

**HW Debug Logic**

The HW debug logic can perform a similar operation to 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 a cacheline can result in decreased cache performance, because the lock prevents the caching of all other instructions that map to the same cache location. In most cases adding data side LMB memory is a better choice for guaranteed access to certain data segments than cacheline locking. The access latency of LMB BRAM is the same as for a cache hit.

**Floating Point Unit (FPU)**

**Overview**

The MicroBlaze floating point unit is based on the IEEE 754 standard:

- Uses IEEE 754 single precision floating point format, including definitions for infinity, not-a-number (NaN), and zero
- Supports addition, subtraction, multiplication, division, and comparison instructions
- Implements round-to-nearest mode
- Generates sticky status bits for: underflow, overflow, and invalid operation

For improved performance, the following non-standard simplifications are made:

- Denormalized operands are not supported. A hardware floating point operation on a denormalized number will return a quiet NaN and set the denormalized operand error bit in FSR
- A denormalized result is stored as a signed 0 with the underflow bit set in FSR. This method is commonly referred to as Flush-to-Zero (FTZ)
- An operation on a quiet NaN will return the fixed NaN: 0xFFC00000, rather than one of the NaN operands
- Overflow as a result of a floating point operation will always return signed $\infty$, even when the exception is trapped.

**Format**

An IEEE 754 single precision floating point number is composed of the following three fields:

1. 1-bit sign

---

1. Basically numbers that are so close to 0, that they cannot be represented with full precision, i.e. any number $n$ that falls in the following ranges: $(1.17549 \times 10^{-38} > n > 0)$, or $(0 > n > -1.17549 \times 10^{-38})$
Floating Point Unit (FPU)

2. 8-bit biased exponent
3. 23-bit fraction (a.k.a. mantissa or significand)

The fields are stored in a 32 bit word as defined in Figure 1-12:

\[
\begin{array}{ccccccccccccccccc}
\text{sign} & \text{exponent} & \text{fraction} \\
\end{array}
\]

Figure 1-12: IEEE 754 Single Precision format

The value of a floating point number \( v \) in MicroBlaze has the following interpretation:

1. If \( \text{exponent} = 255 \) and \( \text{fraction} \neq 0 \), then \( v = \text{NaN} \), regardless of the \( \text{sign} \) bit
2. If \( \text{exponent} = 255 \) and \( \text{fraction} = 0 \), then \( v = (-1)^{\text{sign}} \times \infty \)
3. If \( 0 < \text{exponent} < 255 \), then \( v = (-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1.\text{fraction}) \)
4. If \( \text{exponent} = 0 \) and \( \text{fraction} \neq 0 \), then \( v = (-1)^{\text{sign}} \times 2^{-126} \times (0.\text{fraction}) \)
5. If \( \text{exponent} = 0 \) and \( \text{fraction} = 0 \), then \( v = (-1)^{\text{sign}} \times 0 \)

For practical purposes only 3 and 5 are really useful, while the others all represent either an error or numbers that can no longer be represented with full precision in a 32 bit format.

Rounding

The MicroBlaze FPU only implements the default rounding mode, “Round-to-nearest”, specified in IEEE 754. By definition, the result of any floating point operation should return the nearest single precision value to the infinitely precise result. If the two nearest representable values are equally near, then the one with its least significant bit zero is returned.

Operations

All MicroBlaze FPU operations use the processors general purpose registers rather than a dedicated floating point register file, see “General Purpose Registers”.

Arithmetic

The FPU implements the following floating point operations:

- addition, fadd
- subtraction, fsub
- multiplication, fmul
- division, fdiv

Comparison

The FPU implements the following floating point comparisons:

- compare less-than, fcmp.lt
- compare equal, fcmp.eq
- compare less-or-equal, fcmp.le
• compare greater-than, fcmp.gt
• compare not-equal, fcmp.ne
• compare greater-or-equal, fcmp.ge
• compare unordered, fcmp.un (used for NaN)

Exceptions

The floating point unit uses the regular hardware exception mechanism in MicroBlaze. When enabled, exceptions are thrown for all the IEEE standard conditions: underflow, overflow, divide-by-zero, and illegal operation, as well as for the MicroBlaze specific exception: denormalized operand error.

A floating point exception will inhibit the write to the destination register (Rd). This allows a floating point exception handler to operate on the uncorrupted register file.

Fast Simplex Link (FSL)

MicroBlaze can be configured with up to eight Fast Simplex Link (FSL) interfaces, each consisting of one input and one output port. The FSL channels are dedicated unidirectional point-to-point data streaming interfaces. For detailed information on the FSL interface, please refer to the FSL Bus data sheet (DS449).

The FSL interfaces on MicroBlaze are 32 bits wide. A separate bit indicates whether the sent/received word is of control or data type. The get instruction in the MicroBlaze ISA is used to transfer information from an FSL port to a general purpose register. The put instruction is used to transfer data in the opposite direction. Both instructions come in 4 flavours: blocking data, non-blocking data, blocking control, and non-blocking control. For a detailed description of the get and put instructions please refer to Chapter 4, “MicroBlaze Instruction Set Architecture”.

Hardware Acceleration using FSL

Each FSL provides a low latency dedicated interface to the processor pipeline. Thus they are ideal for extending the processors execution unit with custom hardware accelerators. A simple example is illustrated in Figure 1-13.

**Example code:**

```c
// Configure f_x
put FSLx, Rc  // Configure FSLx

// Store operands
put FSLx, Ra // op 1
put FSLx, Rb // op 2

// Load result
get FSLx, Rt
```

*Figure 1-13: FSL used with HW accelerated function \( f_x \)*
This method is similar to extending the ISA with custom instructions, but has the benefit of not making the overall speed of the processor pipeline dependent on the custom function. Also, there are no additional requirements on the software tool chain associated with this type of functional extension.

## Debug and Trace

### Debug Overview

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) core, which interfaces with the JTAG port of Xilinx FPGAs. Multiple MicroBlaze instances can be interfaced with a single MDM to enable multiprocessor debugging. The debugging features include:

- Configurable number of hardware breakpoints and watchpoints and unlimited software breakpoints
- External processor control enables debug tools to stop, reset, and single step
- Read from and write to: memory, general purpose registers, and special purpose register, except ESR and EAR which can only be read
- Support for multiple processors
- Write to instruction and data caches

### Trace Overview

The MicroBlaze trace interface exports a number of internal state signals for performance monitoring and analysis. Xilinx recommends that users only use the trace interface through Xilinx developed analysis cores. This interface is not guaranteed to be backward compatible in future releases of MicroBlaze.
Chapter 2

MicroBlaze Signal Interface Description

Overview

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. The following three memory interfaces are supported: Local Memory Bus (LMB), IBM’s On-chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). The LMB provides single-cycle access to on-chip dual-port block RAM. The OPB interface provides a connection to both on-chip and off-chip peripherals and memory. The CacheLink interface is intended for use with specialized external memory controllers. MicroBlaze also supports up to 8 Fast Simplex Link (FSL) ports, each with one master and one slave FSL interface.

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
- FSL provides a fast non-arbitrated streaming communication mechanism
- XCL provides a fast slave-side arbitrated streaming interface between caches and specialized external memory controllers
- Debug interface for use with the Microprocessor Debug Module (MDM) core
- Trace interface for performance analysis

MicroBlaze I/O Overview

The core interfaces shown in Figure 2-1 and the following Table 2-1 are defined as follows:

<table>
<thead>
<tr>
<th>DOPB</th>
<th>Data interface, On-chip Peripheral Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLMB</td>
<td>Data interface, Local Memory Bus (BRAM only)</td>
</tr>
<tr>
<td>IOPB</td>
<td>Instruction interface, On-chip Peripheral Bus</td>
</tr>
<tr>
<td>ILMB</td>
<td>Instruction interface, Local Memory Bus (BRAM only)</td>
</tr>
<tr>
<td>MFSL 0..7</td>
<td>FSL master interfaces</td>
</tr>
<tr>
<td>SFSL 0..7</td>
<td>FSL slave interfaces</td>
</tr>
<tr>
<td>IXCL</td>
<td>Instruction side Xilinx CacheLink interface (FSL master/slave pair)</td>
</tr>
<tr>
<td>DXCL</td>
<td>Data side Xilinx CacheLink interface (FSL master/slave pair)</td>
</tr>
<tr>
<td>Core</td>
<td>Miscellaneous signals for: clock, reset, debug, and trace</td>
</tr>
</tbody>
</table>
Chapter 2: MicroBlaze Signal Interface Description

Figure 2-1: MicroBlaze Core Block Diagram

Table 2-1: Summary of MicroBlaze Core I/O

<table>
<thead>
<tr>
<th>Signal</th>
<th>Interface</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM_ABus[0:31]</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB address bus</td>
</tr>
<tr>
<td>DM_BE[0:3]</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB byte enables</td>
</tr>
<tr>
<td>DM_busLock</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB bus lock</td>
</tr>
<tr>
<td>DM_DBus[0:31]</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB write data bus</td>
</tr>
<tr>
<td>DM_request</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB bus request</td>
</tr>
<tr>
<td>DM_RNW</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB read, not write</td>
</tr>
<tr>
<td>DM_select</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB select</td>
</tr>
<tr>
<td>DM_seqAddr</td>
<td>DOPB</td>
<td>O</td>
<td>Data interface OPB sequential address</td>
</tr>
<tr>
<td>DOPB_DBus[0:31]</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB read data bus</td>
</tr>
<tr>
<td>DOPB_errAck</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB error acknowledge</td>
</tr>
<tr>
<td>DOPB_MGrant</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB bus grant</td>
</tr>
<tr>
<td>DOPB_retry</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB bus cycle retry</td>
</tr>
<tr>
<td>DOPB_timeout</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB bus cycle retry</td>
</tr>
<tr>
<td>DOPB_xferAck</td>
<td>DOPB</td>
<td>I</td>
<td>Data interface OPB transfer acknowledge</td>
</tr>
<tr>
<td>IM_ABus[0:31]</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB address bus</td>
</tr>
</tbody>
</table>
### Table 2-1: Summary of MicroBlaze Core I/O (Continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Interface</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM_BE[0:3]</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB byte enables</td>
</tr>
<tr>
<td>IM_busLock</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB bus lock</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>
</tr>
<tr>
<td>IM_request</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB bus request</td>
</tr>
<tr>
<td>IM_RNW</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB read, not write (tied to IM_select)</td>
</tr>
<tr>
<td>IM_select</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB select</td>
</tr>
<tr>
<td>IM_seqAddr</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB sequential address</td>
</tr>
<tr>
<td>IOPB_DBus[0:31]</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB read data bus</td>
</tr>
<tr>
<td>IOPB_errAck</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB error acknowledge</td>
</tr>
<tr>
<td>IOPB_MGrant</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB bus grant</td>
</tr>
<tr>
<td>IOPB_retry</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB bus cycle retry</td>
</tr>
<tr>
<td>IOPB_timeout</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB timeout error</td>
</tr>
<tr>
<td>IOPB_xferAck</td>
<td>IOPB</td>
<td>I</td>
<td>Instruction interface OPB transfer acknowledge</td>
</tr>
<tr>
<td>Data_Addr[0:31]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LMB address bus</td>
</tr>
<tr>
<td>Byte_Enable[0:3]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LMB byte enables</td>
</tr>
<tr>
<td>Data_Write[0:31]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LMB write data bus</td>
</tr>
<tr>
<td>D_AS</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LMB address strobe</td>
</tr>
<tr>
<td>Read_Strobe</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LMB read strobe</td>
</tr>
<tr>
<td>Write_Strobe</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LMB write strobe</td>
</tr>
<tr>
<td>Data_Read[0:31]</td>
<td>DLMB</td>
<td>I</td>
<td>Data interface LMB read data bus</td>
</tr>
<tr>
<td>DReady</td>
<td>DLMB</td>
<td>I</td>
<td>Data interface LMB data ready</td>
</tr>
<tr>
<td>Instr_Addr[0:31]</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LMB address bus</td>
</tr>
<tr>
<td>I_AS</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LMB address strobe</td>
</tr>
<tr>
<td>IFetch</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LMB instruction fetch</td>
</tr>
<tr>
<td>Instr[0:31]</td>
<td>ILMB</td>
<td>I</td>
<td>Instruction interface LMB read data bus</td>
</tr>
<tr>
<td>IReady</td>
<td>ILMB</td>
<td>I</td>
<td>Instruction interface LMB data ready</td>
</tr>
<tr>
<td>FSL0_M .. FSL7_M</td>
<td>MFSL</td>
<td>O</td>
<td>Master interface to output FSL channels</td>
</tr>
<tr>
<td>FSL0_S .. FSL7_S</td>
<td>SFSL</td>
<td>I</td>
<td>Slave interface to input FSL channels</td>
</tr>
<tr>
<td>ICache_FSL_in...</td>
<td>IXCL_S</td>
<td>IO</td>
<td>Instruction side CacheLink FSL slave interface</td>
</tr>
</tbody>
</table>
Chapter 2: MicroBlaze Signal Interface Description

On-Chip Peripheral Bus (OPB) Interface Description

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 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

### Table 2-1: Summary of MicroBlaze Core I/O (Continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Interface</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICache_FSL_out...</td>
<td>IXCL_M</td>
<td>IO</td>
<td>Instruction side CacheLink FSL master interface</td>
</tr>
<tr>
<td>DCache_FSL_in...</td>
<td>DXCL_S</td>
<td>IO</td>
<td>Data side CacheLink FSL slave interface</td>
</tr>
<tr>
<td>DCache_FSL_out...</td>
<td>DXCL_M</td>
<td>IO</td>
<td>Data side CacheLink FSL master interface</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Core</td>
<td>I</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Reset</td>
<td>Core</td>
<td>I</td>
<td>Core reset, active high. Should be held for at least 16 cycles</td>
</tr>
<tr>
<td>Clk</td>
<td>Core</td>
<td>I</td>
<td>Clock</td>
</tr>
<tr>
<td>Debug_Rst</td>
<td>Core</td>
<td>I</td>
<td>Reset signal from OPB JTAG UART, active high. Should be held for at least 16 cycles</td>
</tr>
<tr>
<td>Ext_BRK</td>
<td>Core</td>
<td>I</td>
<td>Break signal from OPB JTAG UART</td>
</tr>
<tr>
<td>Ext_NM_BRK</td>
<td>Core</td>
<td>I</td>
<td>Non-maskable break signal from OPB JTAG UART</td>
</tr>
<tr>
<td>Dbg....</td>
<td>Core</td>
<td>IO</td>
<td>Debug signals from OPB MDM</td>
</tr>
<tr>
<td>Valid_Instr</td>
<td>Core</td>
<td>O</td>
<td>Trace: Valid instruction in EX stage</td>
</tr>
<tr>
<td>PC_Ex</td>
<td>Core</td>
<td>O</td>
<td>Trace: Address for EX stage instruction</td>
</tr>
<tr>
<td>Reg_Write</td>
<td>Core</td>
<td>O</td>
<td>Trace: EX stage instruction writes to the register file</td>
</tr>
<tr>
<td>Reg_Addr</td>
<td>Core</td>
<td>O</td>
<td>Trace: Destination register</td>
</tr>
<tr>
<td>MSR_Reg</td>
<td>Core</td>
<td>O</td>
<td>Trace: Current MSR register value</td>
</tr>
<tr>
<td>New_Reg_Value</td>
<td>Core</td>
<td>O</td>
<td>Trace: Destination register write data</td>
</tr>
<tr>
<td>Pipe_Running</td>
<td>Core</td>
<td>O</td>
<td>Trace: Processor pipeline to advance</td>
</tr>
<tr>
<td>Interrup_Taken</td>
<td>Core</td>
<td>O</td>
<td>Trace: Unmasked interrupt has occurred</td>
</tr>
<tr>
<td>Jump_Taken</td>
<td>Core</td>
<td>O</td>
<td>Trace: Branch instruction evaluated true</td>
</tr>
<tr>
<td>Prefetch_Addr</td>
<td>Core</td>
<td>O</td>
<td>Trace: OF stage pointer into prefetch buffer</td>
</tr>
<tr>
<td>MB_Halted</td>
<td>Core</td>
<td>O</td>
<td>Trace: Pipeline is halted</td>
</tr>
<tr>
<td>Trace....</td>
<td>Core</td>
<td>O</td>
<td>Trace signals for real time HW analysis</td>
</tr>
</tbody>
</table>
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 so that its interface remains consistent with the data-side OPB. These signals are constant and generally are minimized in implementation.

Local Memory Bus (LMB) Interface Description

The 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 are accessed in a single clock cycle. LMB signals and definitions are shown in the following table. All LMB signals are active high.

LMB Signal Interface

Table 2-2: LMB Bus Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Data Interface</th>
<th>Instruction 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><em>not used</em></td>
<td>O</td>
<td>Byte enables</td>
</tr>
<tr>
<td>Data_Write[0:31]</td>
<td>Data_Write[0:31]</td>
<td><em>not used</em></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><em>not used</em></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>
<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></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>x</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>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1111</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

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 Transactions**

The following diagrams provide examples of LMB bus operations.

**Generic Write Operation**

![Diagram of LMB Generic Write Operation](image)

**Figure 2-2: LMB Generic Write Operation**

**Generic Read Operation**

![Diagram of LMB Generic Read Operation](image)

**Figure 2-3: LMB Generic Read Operation**
Chapter 2: MicroBlaze Signal Interface Description

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)
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 2-4, and data steering for write cycles is shown in Table 2-5.

**Table 2-4: 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>

![Figure 2-6: Back-to-Back Mixed Read/Write Operation](image)
Chapter 2: MicroBlaze Signal Interface Description

### Fast Simplex Link (FSL) Interface Description

The Fast Simplex Link bus provides a point-to-point communication channel between an output FIFO and an input FIFO. For details on the generic FSL protocol please refer to the “Fast Simplex Link (FSL) bus” data sheet (DS449).

### Master FSL Signal Interface

MicroBlaze may contain up to 8 master FSL interfaces. The master signals are depicted in Table 2-6.

#### Table 2-5: Write Data Steering (store from Register rD)

<table>
<thead>
<tr>
<th>Address [30:31]</th>
<th>Byte_Enable [0:3]</th>
<th>Transfer Size</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0001</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>10</td>
<td>0010</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>01</td>
<td>0100</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>10</td>
<td>0011</td>
<td>halfword</td>
<td></td>
<td></td>
<td></td>
<td>rD[16:23] rD[24:31]</td>
</tr>
<tr>
<td>00</td>
<td>1100</td>
<td>halfword</td>
<td></td>
<td></td>
<td></td>
<td>rD[16:23] rD[24:31]</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.

#### Table 2-6: Master 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_M_Clk</td>
<td>Clock</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>FSLn_M_Write</td>
<td>Write enable signal indicating that data is being written to the output FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_Data</td>
<td>Data value written to the output FSL</td>
<td>std_logic_vector</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_Control</td>
<td>Control bit value written to the output FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_M_Full</td>
<td>Full Bit indicating output FSL FIFO is full when set</td>
<td>std_logic</td>
<td>input</td>
</tr>
</tbody>
</table>
Slave FSL Signal Interface

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

Table 2-7: 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>input</td>
</tr>
<tr>
<td>FSLn_S_Read</td>
<td>Read acknowledge signal indicating that data has been read from the input FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>FSLn_S_Data</td>
<td>Data value currently available at the top of the input FSL</td>
<td>std_logic_vector</td>
<td>input</td>
</tr>
<tr>
<td>FSLn_S_Control</td>
<td>Control Bit value currently available at the top of the input FSL</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>FSLn_S_Exists</td>
<td>Flag indicating that data exists in the input FSL</td>
<td>std_logic</td>
<td>input</td>
</tr>
</tbody>
</table>

FSL Transactions

FSL BUS Write Operation

A write to the FSL bus is performed by MicroBlaze using one of the flavors of the put instruction. A write operations transfers the register contents to an output FSL bus. The transfer is typically completed in 2 clock cycles for blocking mode writes to the FSL (put and cput instructions) as long as the FSL FIFO does not become full. If the FSL FIFO is full, the processor stalls at this instruction until the FSL full flag is lowered. In the non-blocking mode (nput and ncput instructions), the transfer is completed in two clock cycles irrespective of whether or not the FSL was full. In the case the FSL was full, the transfer of data does not take place and the carry bit is set in the MSR.

FSL BUS Read Operation

A read from the FSL bus is performed by MicroBlaze using one of the flavors of the get instruction. A read operations transfers the contents of an input FSL to a general purpose register. The transfer is typically completed in 2 clock cycles for blocking mode reads from the FSL (get and cget instructions) as long as data exists in the FSL FIFO. If the FSL FIFO is empty, the processor stalls at this instruction until the FSL exists flag is set. In the non-blocking mode (nget and ncget instructions), the transfer is completed in two clock cycles irrespective of whether or not the FSL was empty. In the case the FSL was empty, the transfer of data does not take place and the carry bit is set in the MSR.

Xilinx CacheLink (XCL) Interface Description

Xilinx CacheLink (XCL) is a high performance solution for external memory accesses. The MicroBlaze CacheLink interface can either connect to an Fast Simplex Link (FSL).
interfaced memory controller via an explicitly instantiated FSL master/slave pair (see Figure 2-7),

Figure 2-7: CacheLink connection with explicit FSL buffers (only Instruction cache used in this example)

or it can connect directly to a memory controller with integrated FSL buffers, e.g. the MCH_OPB_SDRAM), which results in less latency and fewer instantiations (see Figure 2-8).

Figure 2-8: CacheLink connection with integrated FSL buffers (only Instruction cache used in this example)
The interface is only available on MicroBlaze when caches are enabled, and supports the same Harvard architecture as the regular OPB caches. The parameters: C_ICACHE_USE_FSL and C_DCACHE_USE_FSL select if caching is done over OPB or CacheLink for the instruction and data side respectively. It is possible to combine an OPB cache on one side with a CacheLink on the other. It is also allowed to use a CacheLink cache on one side without caching on the other. Memory locations outside the cacheable range are accessed through the OPB (or LMB).

The CacheLink cache controllers handle 4-word cache lines (critical word first), which increases hit rate. At the same time the separation from the OPB bus reduces contention for non-cached memory accesses. The CacheLink caches remain direct mapped, with single word write-through, and no fetch on write miss (identical to the OPB caches).

**CacheLink Signal Interface**

The CacheLink signals on MicroBlaze are listed in [Table 2-8](#).

### Table 2-8: MicroBlaze Cache Link signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_FSL_IN_Clk</td>
<td>Clock output to I-side return read data FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>ICACHE_FSL_IN_Read</td>
<td>Read signal to I-side return read data FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>ICACHE_FSL_IN_Data</td>
<td>Read data from I-side return read data FSL</td>
<td>std_logic_vector (0 to 31)</td>
<td>input</td>
</tr>
<tr>
<td>ICACHE_FSL_IN_Control</td>
<td>FSL control-bit from I-side return read data FSL. Reserved for future use</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>ICACHE_FSL_IN_Exists</td>
<td>More read data exists in I-side return FSL</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>ICACHE_FSL_OUT_Clk</td>
<td>Clock output to I-side read access FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>ICACHE_FSL_OUT_Write</td>
<td>Write new cache miss access request to I-side read access FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>ICACHE_FSL_OUT_Data</td>
<td>Cache miss access (=address) to I-side read access FSL</td>
<td>std_logic_vector (0 to 31)</td>
<td>output</td>
</tr>
<tr>
<td>ICACHE_FSL_OUT_Control</td>
<td>FSL control-bit to I-side read access FSL. Reserved for future use</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>ICACHE_FSL_OUT_Full</td>
<td>FSL access buffer for I-side read accesses is full</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>DCACHE_FSL_IN_Clk</td>
<td>Clock output to D-side return read data FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
</tbody>
</table>
### Table 2-8: MicroBlaze Cache Link signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCACHE_FSL_IN_Read</td>
<td>Read signal to D-side return read data FSL</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>DCACHE_FSL_IN_Data</td>
<td>Read data from D-side return read data FSL</td>
<td>std_logic_vector (0 to 31)</td>
<td>input</td>
</tr>
<tr>
<td>DCACHE_FSL_IN_Control</td>
<td>FSL control bit from D-side return read data FSL</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>DCACHE_FSL_IN_Exists</td>
<td>More read data exists in D-side return FSL</td>
<td>std_logic</td>
<td>input</td>
</tr>
<tr>
<td>DCACHE_FSL_OUT_Clk</td>
<td>Clock output to D-side read access FSL</td>
<td>std_logic;</td>
<td>output</td>
</tr>
<tr>
<td>DCACHE_FSL_OUT_Write</td>
<td>Write new cache miss access request to D-side read access FSL</td>
<td>std_logic;</td>
<td>output</td>
</tr>
<tr>
<td>DCACHE_FSL_OUT_Data</td>
<td>Cache miss access (read address or write address + write data + byte write enable) to D-side read access FSL</td>
<td>std_logic_vector (0 to 31)</td>
<td>output</td>
</tr>
<tr>
<td>DCACHE_FSL_OUT_Control</td>
<td>FSL control-bit to D-side read access FSL. Used with address bits [30 to 31] for read/write and byte enable encoding.</td>
<td>std_logic;</td>
<td>output</td>
</tr>
<tr>
<td>DCACHE_FSL_OUT_Full</td>
<td>FSL access buffer for D-side read accesses is full</td>
<td>std_logic;</td>
<td>input</td>
</tr>
</tbody>
</table>

### CacheLink Transactions

All individual CacheLink accesses follow the FSL FIFO based transaction protocol:

- Access information is encoded over the FSL data and control signals (e.g., DCACHE_FSL_OUT_Data, DCACHE_FSL_OUT_Control, ICACHE_FSL_IN_Data, and ICACHE_FSL_IN_Control).
- Information is sent (stored) by raising the write enable signal (e.g., DCACHE_FSL_OUT_Write).
- The sender is only allowed to write if the full signal from the receiver is inactive (e.g., DCACHE_FSL_OUT_Full = 0).
- Information is received (loaded) by raising the read signal (e.g., ICACHE_FSL_IN_Read).
- The receiver is only allowed to read as long as the sender signals that new data exists (e.g., ICACHE_FSL_IN_Exists = 1).

For details on the generic FSL protocol please refer to the “Fast Simplex Link (FSL) bus” data sheet (DS449).
The CacheLink solution uses one incoming (slave) and one outgoing (master) FSL per cache controller. The outgoing FSL is used to send access requests, while the incoming FSL is used for receiving the requested cache lines. CacheLink also uses a specific encoding of the transaction information over the FSL data and control signals.

The cache lines used for reads in the CacheLink protocol are 4 words long. Each cache line is expected to start with the critical word first. I.e. if an access to address 0x348 is a miss, then the returned cache line should have the following address sequence: 0x348, 0x34c, 0x340, 0x344. The cache controller will forward the first word to the execution unit as well as store it in the cache memory. This allows execution to resume as soon as the first word is back. The cache controller then follows through by filling up the cache line with the remaining 3 words as they are received.

All write operations to the data cache are single-word write-through.

Instruction and Data Cache Read Miss

On a read miss the cache controller will perform the following sequence:

1. If xCACHE_FSL_OUT_Full = 1 then stall until it goes low
2. Write the word aligned\(^1\) missed address to xCACHE_FSL_OUT_Data, with the control bit set low (xCACHE_FSL_OUT_Control = 0) to indicate a read access
3. Wait until xCACHE_FSL_IN_Exists goes high to indicate that data is available
4. Store the word from xCACHE_FSL_IN_Data to the cache
5. Forward the critical word to the execution unit in order to resume execution
6. Repeat 3 and 4 for the subsequent 3 words in the cache line

Data Cache Write

Note that writes to the data cache always are write-through, and thus there will be a write over the CacheLink regardless of whether there was a hit or miss in the cache. On a write the cache controller will perform the following sequence:

1. If DCACHE_FSL_OUT_Full = 1 then stall until it goes low
2. Write the missed address to DCACHE_FSL_OUT_Data, with the control bit set high (DCACHE_FSL_OUT_Control = 1) to indicate a write access
3. If DCACHE_FSL_OUT_Full = 1 then stall until it goes low
4. Write the data to be stored to DCACHE_FSL_OUT_Data. For byte and halfword accesses the data is mirrored accordingly onto byte-lanes. The control bit should be low (DCACHE_FSL_OUT_Control = 0) for a word or halfword access, and high for a byte access.

Debug Interface Description

The debug interface on MicroBlaze is designed to work with the Xilinx Microprocessor Debug Module (MDM) IP core. The MDM is controlled by the Xilinx Microprocessor Debugger (XMD) through the JTAG port of the FPGA. The MDM can control multiple

---

\(^1\) Byte and halfword read misses are naturally expected to return complete words, the cache controller then provides the execution unit with the correct bytes.
MicroBlaze processors at the same time. The debug signals on MicroBlaze are listed in Table 2-9.

**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_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>
<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>

**Trace Interface Description**

The MicroBlaze core exports a number of internal signals for trace purposes. This signal interface is not standardized and new revisions of the processor may not be backward compatible for signal selection or functionality. Users are recommended not to design custom logic for these signals, but rather to use them via Xilinx provided analysis IP. The current set of trace signals are listed in Table 2-10.

**Table 2-10: MicroBlaze Trace signals**

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid_Instr</td>
<td>Valid instruction in processor execute stage</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>PC_Ex</td>
<td>Program counter for processor execute stage</td>
<td>std_logic_vector (0 to 31)</td>
<td>output</td>
</tr>
<tr>
<td>Reg_Write</td>
<td>Execute-stage instruction writes to the register file</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Reg_Addr</td>
<td>Destination register for instruction in execute stage</td>
<td>std_logic_vector (0 to 4)</td>
<td>output</td>
</tr>
<tr>
<td>MSR_Reg</td>
<td>MSR contents before execution of current execute stage instruction</td>
<td>std_logic_vector (0 to 9)</td>
<td>output</td>
</tr>
<tr>
<td>New_Reg_Value</td>
<td>Destination register write data</td>
<td>std_logic_vector (0 to 31)</td>
<td>output</td>
</tr>
<tr>
<td>Pipe_Running</td>
<td>Processor pipeline to advance</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Interrup_Taken</td>
<td>Unmasked interrupt has occurred</td>
<td>std_logic</td>
<td>output</td>
</tr>
</tbody>
</table>
MicroBlaze Core Configurability

The MicroBlaze core has been developed to support a high degree of user configurability. This allows tailoring of the processor to meet specific cost/performance requirements.

Configuration is done via parameters that typically: enable, size, or select certain processor features. E.g. the instruction cache is enabled by setting the C_USE_ICACHE parameter. The size of the instruction cache, the cacheable memory range, and over which interface to cache, are all configurable using: C_CACHE_BYTE_SIZE, C_ICACHE_BASEADDR, C_ICACHE_HIGHADDR, and C_ICACHE_USE_FSL respectively.

Table 2-10: MicroBlaze Trace signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>VHDL Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump_Taken</td>
<td>Branch instruction evaluated true</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Prefetch_Addr</td>
<td>Which position in the prefetch buffer should be used for the decode stage in the next pipeline shift</td>
<td>std_logic_vector (0 to 3)</td>
<td>output</td>
</tr>
<tr>
<td>MB_Halted</td>
<td>Processor pipeline execution is halted</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_Branch_Instr</td>
<td>Instruction to be executed is a branch instruction</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_Delay_Slot</td>
<td>Current cycle is a branch delay slot</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_Data_Address</td>
<td>Address for D-side memory access</td>
<td>std_logic_vector (0 to 31)</td>
<td>output</td>
</tr>
<tr>
<td>Trace_AS</td>
<td>Trace_Data_Address is valid</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_Data_Read</td>
<td>D-side memory access is a read</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_Data_Write</td>
<td>D-side memory access is a write</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_DCache_Req</td>
<td>Data memory address is in D-Cache range</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_DCache_Hit</td>
<td>Data memory address is present in D-Cache</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_ICache_Req</td>
<td>Instruction memory address is in I-Cache range</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_ICache_Hit</td>
<td>Instruction memory address is present in I-Cache</td>
<td>std_logic</td>
<td>output</td>
</tr>
<tr>
<td>Trace_Instr_EX</td>
<td>Execute stage instruction code</td>
<td>std_logic_vector (0 to 31)</td>
<td>output</td>
</tr>
</tbody>
</table>
Parameters valid for MicroBlaze v4.00a are listed in Table 2-11. Note that not all of these are recognized by older versions of MicroBlaze, however the configurability is fully backward compatibility.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Feature/Description</th>
<th>Allowable Values</th>
<th>Default Value</th>
<th>EDK Tool Assigned</th>
<th>VHDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_FAMILY</td>
<td>Target Family</td>
<td>qrvirtex2, qvirtex2, spartan2, spartan2e, spartan3e, spartan3, virtex, virtex2, virtex2p, virtex4, virtexae</td>
<td>virtex2</td>
<td>yes</td>
<td>string</td>
</tr>
<tr>
<td>C_DATA_SIZE</td>
<td>Data Size</td>
<td>32</td>
<td>32</td>
<td>NA</td>
<td>integer</td>
</tr>
<tr>
<td>C_INSTANCE</td>
<td>Instance Name</td>
<td>Any instance name</td>
<td>microblaze</td>
<td>yes</td>
<td>string</td>
</tr>
<tr>
<td>C_D_OPB</td>
<td>Data side OPB interface</td>
<td>0, 1</td>
<td>1</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_D_LMB</td>
<td>Data side LMB interface</td>
<td>0, 1</td>
<td>1</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_I_OPB</td>
<td>Instruction side OPB interface</td>
<td>0, 1</td>
<td>1</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_I_LMB</td>
<td>Instruction side LMB interface</td>
<td>0, 1</td>
<td>1</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_USE_BARREL</td>
<td>Include barrel shifter</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_USE_DIV</td>
<td>Include hardware divider</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_USE_HW_MUL</td>
<td>Include hardware multiplier (Virtex2 and later)</td>
<td>0, 1</td>
<td>1</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_USE_FPU</td>
<td>Include hardware floating point unit</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_USE_MSR_INSTR</td>
<td>Enable use of instructions: MSRSET and MSRCLR</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_USE_PCMP_INSTR</td>
<td>Enable use of instructions: PCMPBF, PCMPEQ, and PCMPNE</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_UNALIGNED_EXCEPTION</td>
<td>Enable exception handling for unaligned data accesses</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_ILL_OPCODE_EXCEPTION</td>
<td>Enable exception handling for illegal op-code</td>
<td>0, 1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-11: MPD Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Feature/Description</th>
<th>Allowable Values</th>
<th>Default Value</th>
<th>EDK Tool Assigned</th>
<th>VHDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_IOPB_BUS_EXCEPTION</td>
<td>Enable exception handling for IOPB bus error</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_DOPB_BUS_EXCEPTION</td>
<td>Enable exception handling for DOPB bus error</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_DIV_ZERO_EXCEPTION</td>
<td>Enable exception handling for division by zero</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_FPU_EXCEPTION</td>
<td>Enable exception handling for hardware floating point unit exceptions</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_DEBUG_ENABLED</td>
<td>MDM Debug interface</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_NUMBER_OF_PC_BRK</td>
<td>Number of hardware breakpoints</td>
<td>0-8</td>
<td>1</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_NUMBER_OF_RD_ADDR_BRK</td>
<td>Number of read address watchpoints</td>
<td>0-4</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_NUMBER_OF_WR_ADDR_BRK</td>
<td>Number of write address watchpoints</td>
<td>0-4</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_INTERRUPT_IS_EDGE</td>
<td>Level/Edge Interrupt</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_EDGE_IS_POSITIVE</td>
<td>Negative/Positive Edge Interrupt</td>
<td>0, 1</td>
<td>1</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_FSL_LINKS</td>
<td>Number of FSL interfaces</td>
<td>0-8</td>
<td>0</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_FSL_DATA_SIZE</td>
<td>FSL data bus size</td>
<td>32</td>
<td>32</td>
<td>NA</td>
<td>integer</td>
</tr>
<tr>
<td>C_ICACHE_BASEADDR</td>
<td>Instruction cache base address</td>
<td>0x00000000 - 0xFFFFFFFF</td>
<td>0x00000000</td>
<td>std_logic_vector</td>
<td></td>
</tr>
<tr>
<td>C_ICACHE_HIGHADDR</td>
<td>Instruction cache high address</td>
<td>0x00000000 - 0xFFFFFFFF</td>
<td>0x3FFFF FFFF</td>
<td>std_logic_vector</td>
<td></td>
</tr>
<tr>
<td>C_USE_ICACHE</td>
<td>Instruction cache</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_ALLOW_ICACHE_WR</td>
<td>Instruction cache write enable</td>
<td>0, 1</td>
<td>1</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_ADDR_TAG_BITS</td>
<td>Instruction cache address tags</td>
<td>0-21</td>
<td>17</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_CACHE_BYTE_SIZE</td>
<td>Instruction cache size</td>
<td>1024, 2048, 4096, 8192, 16384, 32768, 65536</td>
<td>8192</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_ICACHE_USE_FSL</td>
<td>Cache over CacheLink instead of OPB for instructions</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
</tbody>
</table>
### Table 2-11: MPD Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Feature/Description</th>
<th>Allowable Values</th>
<th>Default Value</th>
<th>EDK Tool Assigned</th>
<th>VHDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_DCACHE_BASEADDR</td>
<td>Data cache base address</td>
<td>0x00000000 - 0xFFFFFFFF</td>
<td>0x00000000</td>
<td>std_logic_vector</td>
<td></td>
</tr>
<tr>
<td>C_DCACHE_HIGHADDR</td>
<td>Data cache high address</td>
<td>0x00000000 - 0xFFFFFFFF</td>
<td>0x3FFFFFFFF</td>
<td>std_logic_vector</td>
<td></td>
</tr>
<tr>
<td>C_USE_DCACHE</td>
<td>Data cache</td>
<td>0,1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_ALLOW_DCACHE_WR</td>
<td>Data cache write enable</td>
<td>0,1</td>
<td>1</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_DCACHE_ADDR_TAG</td>
<td>Data cache address tags</td>
<td>0-20</td>
<td>17</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_DCACHE_BYTE_SIZE</td>
<td>Data cache size</td>
<td>2048, 4096, 8192, 16384, 32768, 65536(^2)</td>
<td>8192</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>C_DCACHE_USE_FSL</td>
<td>Cache over CacheLink instead of OPB for data</td>
<td>0,1</td>
<td>0</td>
<td>integer</td>
<td></td>
</tr>
</tbody>
</table>

1. Not all sizes are permitted in all architectures. The cache will use between 1 and 32 RAMB primitives. In older architectures (Virtex, VirtexE, Spartan2, Spartan2E) this limits the maximum size to 16384kB.

2. Not all sizes are permitted in all architectures. The cache will use between 4 and 32 RAMB primitives. In older architectures (Virtex, VirtexE, Spartan2, Spartan2E) this limits the maximum size to 16384kB.
Chapter 3

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 3-1. Data types such as data8, data16, and data32 are used in place of the usual byte, halfword, and word. egister

Table 3-1: 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>float</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 3-2.
The architecture for MicroBlaze defines 32 general purpose registers (GPRs). These registers are classified as volatile, non-volatile and dedicated.

- The volatile registers (a.k.a caller-save) 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 (a.k.a callee-save). 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.

Register R18 is used as a temporary register for assembler operations.

MicroBlaze includes special purpose registers such as: program counter (rpc), machine status register (rmsr), exception status register (resr), exception address register (rear), and floating point status register (rfsr). 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 of a special purpose registers can be transferred to a general purpose register 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 3-1. The shaded area in Figure 3-1 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 3-1: Stack Convention

<table>
<thead>
<tr>
<th>High Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Parameters for called sub-routine (Arg n ..Arg1) (Optional: Maximum number of arguments required for any called procedure from the current procedure.)</td>
</tr>
<tr>
<td>Old Stack Pointer</td>
</tr>
<tr>
<td>Link Register (R15)</td>
</tr>
<tr>
<td>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>
</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 3-2. 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 3-2.

### 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 3-2. 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 in the common 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 3-3. At these locations, code is written to jump to the appropriate handlers.

<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>User exception</td>
<td>0x8</td>
<td>_exception_handler</td>
</tr>
<tr>
<td>Interrupt</td>
<td>0x10</td>
<td>_interrupt_handler</td>
</tr>
<tr>
<td>Hardware exception</td>
<td>0x20</td>
<td>_hw_exception_handler</td>
</tr>
</tbody>
</table>

The code expected at these locations is as shown in Figure 3-3. For programs compiled without the -xl-mode-xmdstub compiler option, the _cr0.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.

For programs compiled with the -xl-mode-xmdstub compiler option, the crt1.o initialization file is linked to the output program. This program has to be run with the xmdstub already loaded in the memory at address location 0x0. Hence at run-time, the initialization code in crt1.o writes the appropriate instructions to location 0x8 through 0x14 depending on the address of the exception and interrupt handlers.

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

0x00: bri _start1
0x04: nop
0x08: imm high bits of address (user exception handler)
0x0c: bri _exception_handler
0x10: imm high bits of address (interrupt handler)
0x14: bri _interrupt_handler
0x20: imm high bits of address (HW exception handler)
0x24: bri _hw_exception_handler

MicroBlaze allows exception and interrupt handler routines to be located at any address location addressable using 32 bits. The user exception handler code starts with the label _exception_handler, the hardware exception handler starts with _hw_exception_handler, while the interrupt handler code starts with the label _interrupt_handler.

In the current MicroBlaze system, there are dummy routines for interrupt and 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 in the document: UG111 Embedded System Tools Reference Manual.
Chapter 4

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 4-1.

Table 4-1: Symbol notation

<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>¬x</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 4: 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>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</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>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</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.

---

**Table 4-1: Symbol notation**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem(x)</td>
<td>Memory location at address x</td>
</tr>
<tr>
<td>FSLx</td>
<td>FSL interface x</td>
</tr>
<tr>
<td>LSW(x)</td>
<td>Least Significant Word of x</td>
</tr>
<tr>
<td>isDnz(x)</td>
<td>Floating point: true if x is denormalized</td>
</tr>
<tr>
<td>isInfinite(x)</td>
<td>Floating point: true if x is +(\infty) or -(\infty)</td>
</tr>
<tr>
<td>isPosInfinite(x)</td>
<td>Floating point: true if x is +(\infty)</td>
</tr>
<tr>
<td>isNegInfinite(x)</td>
<td>Floating point: true if x is -(\infty)</td>
</tr>
<tr>
<td>isNaN(x)</td>
<td>Floating point: true if x is a quiet or signalling NaN</td>
</tr>
<tr>
<td>isZero(x)</td>
<td>Floating point: true if x is +0 or -0</td>
</tr>
<tr>
<td>isQuietNaN(x)</td>
<td>Floating point: true if x is a quiet NaN</td>
</tr>
<tr>
<td>isSigNaN(x)</td>
<td>Floating point: true if x is a signaling NaN</td>
</tr>
<tr>
<td>signZero(x)</td>
<td>Floating point: return +0 for x &gt; 0, and -0 if x &lt; 0</td>
</tr>
<tr>
<td>signInfinity(x)</td>
<td>Floating point: return +(\infty) for x &gt; 0, and -(\infty) if x &lt; 0</td>
</tr>
</tbody>
</table>
add

Arithmetic Add

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Bit 3 (K)</th>
<th>Bit 4 (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addc</td>
<td>Add with Carry</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addk</td>
<td>Add and Keep Carry</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>addkc</td>
<td>Add with Carry and Keep Carry</td>
<td>0</td>
<td>1</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.
addi

**Add Immediate**

- **Syntax**: `addi rD, rA, IMM`
- **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.
- **Pseudocode**
  
  ```
  if C = 0 then
      (rD) ← (rA) + sext(IMM)
  else
      (rD) ← (rA) + sext(IMM) + MSR[C]
  if K = 0 then
      MSR[C] ← CarryOut
  ```

- **Registers Altered**
  - rD
  - MSR[C]
- **Latency**: 1 cycle
- **Notes**: The C bit in the instruction opcode is not the same as the carry bit in the MSR. 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

and 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</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

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

Pseudocode

(rD) ← (rA) ∧ (rB)

Registers Altered

- rD

Latency

1 cycle
andi Logial AND with Immediate

andi rD, rA, IMM

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

Description

The contents of register rA are ANDed with the value of the IMM field, sign-extended to 32 bits; the result is placed into register rD.

Pseudocode

(rD) ← (rA) ∧ sext(IMM)

Registers Altered

• rD

Latency

1 cycle

Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an IMM instruction. See the imm instruction for details on using 32-bit immediate values.
andn

Logical AND NOT

andn rD, rA, rB

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

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

Pseudocode
\[(rD) \leftarrow (rA) \land (\overline{rB})\]

Registers Altered
- rD

Latency
1 cycle
andni  Logical AND NOT with Immediate

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

Description

The IMM field is sign-extended to 32 bits. The contents of register rA are ANDed with the logical complement of the extended IMM field; the result is placed into register rD.

Pseudocode

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

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.
### beq

**Branch if Equal**

<table>
<thead>
<tr>
<th>beq</th>
<th>rA, rB</th>
<th>Branch if Equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>beqd</td>
<td>rA, rB</td>
<td>Branch if Equal with Delay</td>
</tr>
</tbody>
</table>

#### 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

```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)
beqi Branch Immediate if Equal

beqi rA, IMM Branch Immediate if Equal
beqid rA, IMM Branch Immediate if 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>0</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>
</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 beqi 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.
bge  Branch if Greater or Equal

bge    rA, rB  Branch if Greater or Equal
bged   rA, rB  Branch if Greater or Equal with Delay

<table>
<thead>
<tr>
<th>10011</th>
<th>D 0101</th>
<th>rA</th>
<th>rB</th>
<th>000000000000000000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

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)
Instructions

bgei

Branch Immediate if Greater or Equal

\texttt{bgei} \hspace{1em} \text{rA, IMM} \hspace{1em} \text{Branch Immediate if Greater or Equal}

\texttt{bgeid} \hspace{1em} \text{rA, IMM} \hspace{1em} \text{Branch Immediate if Greater or 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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 bgeid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

\begin{verbatim}
If rA >= 0 then
    PC ← PC + sext(IMM)
else
    PC ← PC + 4
if D = 1 then
    allow following instruction to complete execution
\end{verbatim}

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

\[
\begin{align*}
\text{bgt} & \quad rA, rB & \text{Branch if Greater Than} \\
\text{bgtd} & \quad rA, rB & \text{Branch if Greater Than with Delay}
\end{align*}
\]

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

\[
\begin{align*}
\text{If } rA & > 0 \text{ then} \\
& \quad \text{PC} \leftarrow \text{PC} + rB \\
\text{else} & \\
& \quad \text{PC} \leftarrow \text{PC} + 4 \\
& \quad \text{if } D = 1 \text{ then} \\
& \quad \quad \text{allow following instruction to complete execution}
\end{align*}
\]

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)
Instructions

bgti  
**Branch Immediate if Greater Than**

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

**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 bgti 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.
ble

Branch if Less or Equal

ble rA, rB Branch if Less or Equal
bled rA, rB Branch if Less or Equal with Delay

<table>
<thead>
<tr>
<th>1 0 0 1 1</th>
<th>D 0 0 1 1</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</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 is less or equal to 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

The mnemonic bled will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

If rA <= 0 then
  PC ← PC + rB
else
  PC ← PC + 4
if D = 1 then
  allow following instruction to complete execution

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)
2 cycles (if branch is taken and the D bit is set)
3 cycles (if branch is taken and the D bit is not set)
blei

Branch Immediate if Less or Equal

blei  rA, IMM  Branch Immediate if Less or Equal
bleid  rA, IMM  Branch Immediate if Less or Equal with Delay

Description

Branch if rA is less or equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

The mnemonic bleid will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

If rA <= 0 then
   PC ← PC + sext(IMM)
else
   PC ← PC + 4
if D = 1 then
   allow following instruction to complete execution

 Registers Altered

- PC

Latency

1 cycle (if branch is not taken)
2 cycles (if branch is taken and the D bit is set)
3 cycles (if branch is taken and the D bit is not set)

Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
blt

Branch if Less Than

<table>
<thead>
<tr>
<th>blt</th>
<th>rA, rB</th>
<th>Branch if Less Than</th>
</tr>
</thead>
<tbody>
<tr>
<td>bltd</td>
<td>rA, rB</td>
<td>Branch if Less Than with Delay</td>
</tr>
</tbody>
</table>

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

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)
Instructions

**blti**

**Branch Immediate if Less Than**

```plaintext
blti     rA, IMM  Branch Immediate if Less Than
bltid    rA, IMM  Branch Immediate if Less Than 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>1</th>
<th>0</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>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>0</td>
<td>6</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**

```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.
bne

Branch if Not Equal

\[
\text{bne} \quad rA, rB \quad \text{Branch if Not Equal} \\
\text{bned} \quad rA, rB \quad \text{Branch if Not Equal with Delay}
\]

Description

Branch if \( rA \) not equal to 0, to the instruction located in the offset value of \( rB \). The target of the branch will be the instruction at address \( PC + rB \).

The mnemonic bned will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

Pseudocode

\[
\begin{align*}
\text{If } & rA \neq 0 \text{ then} \\
& PC \leftarrow PC + rB \\
\text{else} & \\
& PC \leftarrow PC + 4 \\
& \text{if } D = 1 \text{ then} \\
& \text{allow following instruction to complete execution}
\end{align*}
\]

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

1  0  1  1  1 |   D  0  0  0  1 |    rA    |   IMM   |
0   6  11  16  31

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.
Unconditional Branch

<table>
<thead>
<tr>
<th>br</th>
<th>rb</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>bra</td>
<td>rb</td>
<td>Branch Absolute</td>
</tr>
<tr>
<td>brd</td>
<td>rb</td>
<td>Branch with Delay</td>
</tr>
<tr>
<td>brad</td>
<td>rb</td>
<td>Branch Absolute with Delay</td>
</tr>
<tr>
<td>brla</td>
<td>rb</td>
<td>Branch and Link with Delay</td>
</tr>
<tr>
<td>brald</td>
<td>rb</td>
<td>Branch Absolute and Link with Delay</td>
</tr>
</tbody>
</table>

### 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, brla and brald will set the D bit. The D bit determines whether there is a branch delay slot or not. If the D bit is set, it means that there is a delay slot and the instruction following the branch (i.e. in the branch delay slot) is allowed to complete execution before executing the target instruction. If the D bit is not set, it means that there is no delay slot, so the instruction to be executed after the branch is the target instruction.

### Pseudocode

```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.
**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 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**

```plaintext
define if L = 1 then
    rD ← PC
else
    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

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

| 1 0 0 1 1 0 | rD  | 0 1 1 0 0 | rB  | 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           | 6   | 11      | 16  | 21             | 31             |

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

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

Registers Altered

- \( rD \)
- \( PC \)
- \( \text{MSR}[\text{BIP}] \)

Latency

3 cycles
**brki**

**Break Immediate**

```
brki rD, IMM
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</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**

Branch and link to the instruction located at address value in IMM, sign-extended to 32 bits. The current value of PC will be stored in rD. The BIP flag in the MSR will be set.

**Pseudocode**

```
(rD) ← PC
PC ← sext(IMM)
MSR[BIP] ← 1
```

**Registers Altered**

- rD
- PC
- MSR[BIP]

**Latency**

3 cycles

**Note**

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
Instructions

**Barrel Shift**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bsrl</strong></td>
<td>Barrel Shift Right Logical</td>
<td></td>
</tr>
<tr>
<td><strong>bsra</strong></td>
<td>Barrel Shift Right Arithmetical</td>
<td></td>
</tr>
<tr>
<td><strong>bsll</strong></td>
<td>Barrel Shift Left Logical</td>
<td></td>
</tr>
</tbody>
</table>

### 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

```plaintext
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

<table>
<thead>
<tr>
<th>bsrli</th>
<th>rD, rA, IMM</th>
<th>Barrel Shift Right Logical Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsrai</td>
<td>rD, rA, IMM</td>
<td>Barrel Shift Right Arithmetical Immediate</td>
</tr>
<tr>
<td>bssl</td>
<td>rD, rA, IMM</td>
<td>Barrel Shift Left Logical Immediate</td>
</tr>
</tbody>
</table>

### Description

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

The mnemonic bssl sets the S bit (Side bit). If the S bit is set, the barrel shift is done to the left. The mnemonics bsr 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 bsr and bssl clear the T bit and the shift performed is Logical.

### Pseudocode

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

### 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.
**Instructions**

### cmp

#### Integer Compare

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>rD, rA, rB compare rB with rA (signed)</td>
</tr>
<tr>
<td>cmpu</td>
<td>rD, rA, rB compare rB with rA (unsigned)</td>
</tr>
</tbody>
</table>

#### Registers Altered
- rD

#### Latency
- 1 cycle.

#### Pseudocode

\[
(rD) \leftarrow (rB) + (rA) + 1 \\
(rD)\text{ (MSB)} \leftarrow (rA) > (rB)
\]

#### 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.
fadd

Floating Point Arithmetic Add

fadd rD, rA, rB Add

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

The floating point sum of registers rA and rB, is placed into register rD.

Pseudocode

```plaintext
if isDnz(rA) or isDnz(rB) then
    (rD) ← 0xFFC00000
    FSR[DO] ← 1
    ESR[EC] ← 00110
else
    if isSigNaN(rA) or isSigNaN(rB) or
       (isPosInfinite(rA) and isNegInfinite(rB)) or
       (isNegInfinite(rA) and isPosInfinite(rB)) then
        (rD) ← 0xFFC00000
        FSR[IO] ← 1
        ESR[EC] ← 00110
    else if isQuietNaN(rA) or isQuietNaN(rB) then
        (rD) ← 0xFFC00000
    else
        if isDnz((rA)+(rB)) then
            (rD) ← signZero((rA)+(rB))
            FSR[UF] ← 1
            ESR[EC] ← 00110
        else
            if isNaN((rA)+(rB)) and then
                (rD) ← signInfinite((rA)+(rB))
                FSR[OF] ← 1
                ESR[EC] ← 00110
            else
                (rD) ← (rA) + (rB)
```

Registers Altered

- rD, unless an FP exception is generated, in which case the register is unchanged
- ESR[EC]
- FSR[IO,UF,OF,DO]

Latency

6 cycles

Note

This instruction is only available when the MicroBlaze parameter C_USE_FPU is set to 1.
**frsub**

Reverse Floating Point Arithmetic Subtraction

`frsub`  
`rD, rA, rB`  
Reverse subtract

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

**Description**

The floating point value in `rA` is subtracted from the floating point value in `rB` and the result is placed into register `rD`.

**Pseudocode**

```plaintext
if isDnz(rA) or isDnz(rB) then  
  (rD) ← 0xFFC00000  
  FSR[DO] ← 1  
  ESR[EC] ← 00110  
else  
  if (isSigNaN(rA) or isSigNaN(rB)) or  
    (isPosInfinite(rA) and isPosInfinite(rB)) or  
    (isNegInfinite(rA) and isNegInfinite(rB)) then  
    (rD) ← 0xFFC00000  
    FSR[IO] ← 1  
    ESR[EC] ← 00110  
  else  
    if isQuietNaN(rA) or isQuietNaN(rB) then  
      (rD) ← 0xFFC00000  
    else  
      if isDnz((rB)-(rA)) then  
        (rD) ← signZero((rB)-(rA))  
        FSR[UF] ← 1  
        ESR[EC] ← 00110  
      else  
        if isNaN((rB)-(rA)) then  
          (rD) ← signInfinite((rB)-(rA))  
          FSR[OF] ← 1  
          ESR[EC] ← 00110  
        else  
          (rD) ← (rB) - (rA)
```

**Registers Altered**

- `rD`, unless an FP exception is generated, in which case the register is unchanged
- `ESR[EC]`
- `FSR[IO,UF,OF,DO]`

**Latency**

6 cycles

**Note**

This instruction is only available when the MicroBlaze parameter `C_USE_FPU` is set to 1.
Floating Point Arithmetic Multiplication

**fmul**

fmul rD, rA, rB Multiply

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

Description

The floating point value in rA is multiplied with the floating point value in rB and the result is placed into register rD.

Pseudocode

```c
if isDnz(rA) or isDnz(rB) then
    (rD) ← 0xFFC00000
    FSR[DO] ← 1
    ESR[EC] ← 00110
else
    if isSigNaN(rA) or isSigNaN(rB) or (isZero(rA) and isInfinite(rB)) or
        (isZero(rB) and isInfinite(rA)) then
        (rD) ← 0xFFC00000
        FSR[IO] ← 1
        ESR[EC] ← 00110
    else
        if isQuietNaN(rA) or isQuietNaN(rB) then
            (rD) ← 0xFFC00000
        else
            if isDnz((rB)*(rA)) then
                (rD) ← signZero((rA)*(rB))
                FSR[UF] ← 1
                ESR[EC] ← 00110
            else
                if isNaN((rB)*(rA)) and then
                    (rD) ← signInfinite((rB)*(rA))
                    FSR[OF] ← 1
                    ESR[EC] ← 00110
                else
                    (rD) ← (rB) * (rA)
```

Registers Altered

- rD, unless an FP exception is generated, in which case the register is unchanged
- ESR[EC]
- FSR[IO,UF,OF,DO]

Latency

6 cycles

Note

This instruction is only available when the MicroBlaze parameter C_USE_FPU is set to 1.
fdiv

Floating Point Arithmetic Division

fdiv rD, rA, rB  Divide

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

Description

The floating point value in rB is divided by the floating point value in rA and the result is placed into register rD.

Pseudocode

    if isDnz(rA) or isDnz(rB) then
        (rD) ← 0xFFC00000
        FSR[DO] ← 1
        ESR[EC] ← 00110
    else
        if isSigNaN(rA) or isSigNaN(rB) or (isZero(rA) and isZero(rB)) or
          (isInfinite(rA) and isInfinite(rB)) then
            (rD) ← 0xFFC00000
            FSR[IO] ← 1
            ESR[EC] ← 00110
        else
            if isQuietNaN(rA) or isQuietNaN(rB) then
                (rD) ← 0xFFC00000
            else
                if isZero(rA) and not isInfinite(rB) then
                    (rD) ← signInfinite((rB)/(rA))
                    FSR[DZ] ← 1
                    ESR[EC] ← 00110
                else
                    if isDnz((rB)/(rA)) then
                        (rD) ← signZero((rA)/(rB))
                        FSR[UF] ← 1
                        ESR[EC] ← 00110
                    else
                        if isNaN((rB)/(rA)) and then
                            (rD) ← signInfinite((rB)/(rA))
                            FSR[OF] ← 1
                            ESR[EC] ← 00110
                        else
                            (rD) ← (rB) / (rA)

Registers Altered

- rD, unless an FP exception is generated, in which case the register is unchanged
- ESR[EC]
- FSR[IO,UF,OF,DO,DZ]

Latency

30 cycles

Note

This instruction is only available when the MicroBlaze parameter C_USE_FPU is set to 1.
**fcmp**

Floating Point Number Comparison

- **fcmp.un** \( rD, rA, rB \) Unordered floating point comparison
- **fcmp.lt** \( rD, rA, rB \) Less-than floating point comparison
- **fcmp.eq** \( rD, rA, rB \) Equal floating point comparison
- **fcmp.le** \( rD, rA, rB \) Less-or-Equal floating point comparison
- **fcmp.gt** \( rD, rA, rB \) Greater-than floating point comparison
- **fcmp.ne** \( rD, rA, rB \) Not-Equal floating point comparison
- **fcmp.ge** \( rD, rA, rB \) Greater-or-Equal floating point comparison

<table>
<thead>
<tr>
<th>OpSel</th>
<th>0 1 0 1 0 1 1 0</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>0 1 0 0 0</th>
<th>0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 6 11 16 21 26 29 31</td>
<td>( rD )</td>
<td>( rA )</td>
<td>( rB )</td>
<td>( \text{OpSel} )</td>
<td>( 0 0 0 )</td>
</tr>
</tbody>
</table>

Description

The floating point value in \( rB \) is compared with the floating point value in \( rA \) and the comparison result is placed into register \( rD \). The \( \text{OpSel} \) field in the instruction code determines the type of comparison performed.

Pseudocode

```plaintext
if isDnz(rA) or isDnz(rB) then
    (rD) ← 0
    FSR[DO] ← 1
    ESR[EC] ← 00110
else
    (read out behavior from Table 4-2)
```

Table 4-2: Floating Point Comparison Operation

<table>
<thead>
<tr>
<th>Comparison Type</th>
<th>Operand Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>OpSel</td>
</tr>
<tr>
<td>Unordered</td>
<td>000</td>
</tr>
<tr>
<td>Less-than</td>
<td>001</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal</td>
<td>010</td>
</tr>
<tr>
<td>Less-or-equal</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4-2: Floating Point Comparison Operation

<table>
<thead>
<tr>
<th>Comparison Type</th>
<th>OpSel</th>
<th>(rB) &gt; (rA)</th>
<th>(rB) &lt; (rA)</th>
<th>(rB) = (rA)</th>
<th>isNaN(rA) or isNaN(rB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater-than</td>
<td>100</td>
<td>(rD) ← 1</td>
<td>(rD) ← 0</td>
<td>(rD) ← 0</td>
<td>(rD) ← 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not-equal</td>
<td>101</td>
<td>(rD) ← 1</td>
<td>(rD) ← 1</td>
<td>(rD) ← 0</td>
<td>(rD) ← 1</td>
</tr>
<tr>
<td>Greater-or-equal</td>
<td>110</td>
<td>(rD) ← 1</td>
<td>(rD) ← 0</td>
<td>(rD) ← 1</td>
<td>(rD) ← 0</td>
</tr>
</tbody>
</table>

**Registers Altered**

- rD, unless an FP exception is generated, in which case the register is unchanged
- ESR[EC]
- FSR[IO, DO]

**Latency**

3 cycles

**Note**

These instructions are only available when the MicroBlaze parameter C_USE_FPU is set to 1.
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. In case of an invalid access the destination register contents is undefined.

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) ← FSLx
if (n = 1) then
    MSR[Carry] ← not (FSLx Exists bit)
if ((FSLx Control bit) == c) then
    MSR[FSL_Error] ← 0
else
    MSR[FSL_Error] ← 1
```

Registers Altered

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

Latency

2 cycles. For blocking instructions, MicroBlaze will first stall until valid data is available.

Note

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

Integer Divide

\[
idiv \quad rD, rA, rB \quad \text{divide } rB \text{ by } rA \text{ (signed)}
\]

\[
idivu \quad rD, rA, rB \quad \text{divide } rB \text{ by } rA \text{ (unsigned)}
\]

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

\[
\begin{align*}
\text{if } (rA) &= 0 \text{then} \\
(rD) &\leftarrow 0 \\
\text{else} \\
(rD) &\leftarrow (rB) \div (rA)
\end{align*}
\]

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.
**imm**

### Immediate

<table>
<thead>
<tr>
<th>imm</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 0</td>
<td>0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

| 0 6 11 16 31 |

**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.
**Ibu**

**Load Byte Unsigned**

```c
Ibu rD, rA, rB
```

**Description**

Loads a byte (8 bits) from the memory location that results from adding the contents of registers rA and rB. The data is placed in the least significant byte of register rD and the other three bytes in rD are cleared.

**Pseudocode**

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

**Registers Altered**

- rD

**Latency**

2 cycles
lbui

Load Byte Unsigned Immediate

**lbui**  
**rD, rA, IMM**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></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**

Addr $\leftarrow (rA) + \text{sext}(\text{IMM})$

(rD)[24:31] $\leftarrow \text{Mem}(\text{Addr})$

(rD)[0:23] $\leftarrow 0$

**Registers Altered**

- rD

**Latency**

2 cycles

**Note**

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
Instructions

Note: Most entries in this document assume a binary little-endian architecture.

## lhu

### Load Halfword Unsigned

**Syntax:**

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

### Description

Loads a halfword (16 bits) from the halfword aligned memory location that results from adding the contents of registers rA and rB. The data is placed in the least significant halfword of register rD and the most significant halfword in rD is cleared.

### Pseudocode

```
Addr ← (rA) + (rB)
Addr[31] ← 0
(rD)[16:31] ← Mem(Addr)
(rD)[0:15] ← 0
```

### Registers Altered

- rD

### Latency

2 cycles
lhui

Load Halfword Unsigned Immediate

**Description**

Loads a halfword (16 bits) from the halfword aligned memory location that results from adding the contents of register rA and the value in IMM, sign-extended to 32 bits. The data is placed in the least significant halfword of register rD and the most significant halfword in rD is cleared.

**Pseudocode**

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

**Registers Altered**

- rD

**Latency**

2 cycles

**Note**

By default, Type B Instructions will take the 16-bit 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.
lw: Load Word

lw rD, rA, rB

Description
Load a word (32 bits) from the word aligned memory location that results from adding the contents of registers rA and rB. The data is placed in register rD.

Pseudocode
Addr ← (rA) + (rB)
Addr[30:31] ← 00
(rD) ← Mem(Addr)

Registers Altered
• rD

Latency
2 cycles
Load Word Immediate

**lwi**

**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**

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

**Registers Altered**
- rD

**Latency**
2 cycles

**Note**
By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
mfs  Move From Special Purpose Register

mfs  rD, rS

<table>
<thead>
<tr>
<th>1 0 0 1 0 1</th>
<th>rD</th>
<th>0 0 0 0 0</th>
<th>1 0 0 0 0 0 0 0 0 0</th>
<th>0 0 0 0</th>
<th>rS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>29</td>
<td>31</td>
</tr>
</tbody>
</table>

Description
Covers the contents of the special purpose register rS into register rD.

Pseudocode
(rD) ← (rS)

Registers Altered
• rD

Latency
1 cycle

Note
To refer to special purpose registers in assembly language, use rpc for PC, rmsr for MSR, rear for EAR, resr for ESR, and rfsr for FSR.

EAR and ESR are only valid as operands when atleast one of the MicroBlaze C_*_EXCEPTION parameters are set to 1.

FSR is only valid as an operand when the C_USE_FPU and C_FPU_EXCEPTION parameters are set to 1.
msrclr

Read MSR and clear bits in MSR

\[
\text{msrclr} \quad rD, \text{Imm}
\]

<table>
<thead>
<tr>
<th>1 0 0 1 0 1</th>
<th>rD</th>
<th>0 0 0 1 0 0</th>
<th>Imm14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16 17</td>
</tr>
</tbody>
</table>

Description

Copies the contents of the special purpose register MSR into register rD. Bit positions in the IMM value that are 1 are cleared in the MSR. Bit positions that are 0 in the IMM value are left untouched.

Pseudocode

\[
(rD) \leftarrow (\text{MSR})
\]

\[
(\text{MSR}) \leftarrow (\text{MSR}) \land (\text{IMM})
\]

Registers Altered

- rD
- MSR

Latency

1 cycle

Note

This instruction is only valid if C\_USE\_MSR\_INSTR is set for MicroBlaze.

MSRCLR will affect some MSR bits immediately (e.g. Carry) while the remaining bits will take effect one cycle after the instruction has been executed.

The immediate values has to be less than 2^{14}. Only bits 18 to 31 of the MSR can be cleared.
msrset

Read MSR and set bits in MSR

msrset rD, Imm

<table>
<thead>
<tr>
<th>1 0 0 1 0 1</th>
<th>rD</th>
<th>0 0 0 0 0 0</th>
<th>Imm14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>18</td>
</tr>
</tbody>
</table>

Description
Copies the contents of the special purpose register MSR into register rD. Bit positions in the IMM value that are 1 are set in the MSR. Bit positions that are 0 in the IMM value are left untouched.

Pseudocode
(rD) ← (MSR)
(MSR) ← (MSR) ∨ (IMM)

Registers Altered
- rD
- MSR

Latency
1 cycle

Note
This instruction is only valid if C_USE_MSR_INSTR is set for MicroBlaze.
MSRSET will affect some MSR bits immediately (e.g. Carry) while the remaining bits will take effect one cycle after the instruction has been executed.
The immediate values has to be less than 2^{14}. Only bits 18 to 31 of the MSR can be set.
### mts

**Move To Special Purpose Register**

```plaintext
mts rS, rA
```

| 1 0 0 1 0 1 | 0 0 0 0 0 | rA | 1 1 0 0 0 0 0 0 0 0 0 0 0 0 | rS |
| 0 | 6 | 11 | 16 | 29 | 31 |

#### Description

Copies the contents of register rD into the MSR or FSR.

#### Pseudocode

```plaintext
(rS) ← (rA)
```

#### Registers Altered

- rS

#### Latency

1 cycle

#### Notes

- When writing MSR using MTS, some bits take effect immediately (e.g. Carry) while the remaining bits take effect one cycle after the instruction has been executed.
- To refer to special purpose registers in assembly language, use rmsr for MSR and rfsr for FSR.
- The PC, ESR and EAR cannot be written by the MTS instruction.
- The FSR is only valid as a destination if the MicroBlaze parameter C_USE_FPU is set to 1.
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 multiplier primitives, and if present, the MicroBlaze parameter C_USE_HW_MUL is set to 1.
**muli**  

**Multiply Immediate**

```
muli rD, rA, IMM
```

<table>
<thead>
<tr>
<th>0</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**

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) ← LSW( (rA) × 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 multiplier primitives, and if present, the MicroBlaze parameter C_USE_HW_MUL is set to 1.
## or

### Logical OR

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

#### 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

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

(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.
Instructions

**pcmpbf**

**Pattern Compare Byte Find**

\[
\text{pcmpbf}\quad rD, rA, rB \quad \text{bytewise comparison returning position of first match}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 6 & 11 & 16 & 21 & 30 & 31 & 31
\end{array}
\]

**Description**

The contents of register \( rA \) is bytewise compared with the contents in register \( rB \).

- \( rD \) is loaded with the position of the first matching byte pair, starting with MSB as position 1, and comparing until LSB as position 4.
- If none of the byte pairs match, \( rD \) is set to 0.

**Pseudocode**

\[
\text{if } rB[0:7] = rA[0:7] \text{ then} \\
\hspace{1em} (rD) \leftarrow 1 \\
\text{else} \\
\hspace{1em} \text{if } rB[8:15] = rA[8:15] \text{ then} \\
\hspace{2em} (rD) \leftarrow 2 \\
\text{else} \\
\hspace{2em} \text{if } rB[16:23] = rA[16:23] \text{ then} \\
\hspace{3em} (rD) \leftarrow 3 \\
\text{else} \\
\hspace{3em} \text{if } rB[24:31] = rA[24:31] \text{ then} \\
\hspace{4em} (rD) \leftarrow 4 \\
\text{else} \\
\hspace{4em} (rD) \leftarrow 0
\]

**Registers Altered**

- \( rD \)

**Latency**

1 cycle

**Note**

This instruction is only available when the MicroBlaze parameter C_USE_PCMP_INSTR is set to 1.
pcmpeq

**Pattern Compare Equal**

pcmpeq  rD, rA, rB  equality comparison with a positive boolean result

|   | 1 | 0 | 0 | 0 | 1 | 0 | rD | rA | rB | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 6 | 11| 16| 21| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Description**

The contents of register rA is compared with the contents in register rB.

- rD is loaded with 1 if they match, and 0 if not

**Pseudocode**

```
if (rB) = (rA) then
    (rD) ← 1
else
    (rD) ← 0
```

**Registers Altered**

- rD

**Latency**

1 cycle

**Note**

This instruction is only available when the MicroBlaze parameter C_USE_PCMP_INSTR is set to 1
pcmpne  

Pattern Compare Not Equal

description

The contents of register rA is compared with the contents in register rB.

- rD is loaded with 0 if they match, and 1 if not

pseudocode

if (rB) = (rA) then
(rD) ← 0
else
(rD) ← 1

registers altered

- rD

latency

1 cycle

note

This instruction is only available when the MicroBlaze parameter C_USE_PCPM_INSTR is set to 1
put

**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’.

**Pseudocode**

\[
\begin{align*}
(FSLx) & \leftarrow (rA) \\
\text{if } (n = 1) \text{ then} & \\
\text{MSR}[\text{Carry}] & \leftarrow (FSLx \text{ Full bit}) \\
(FSLx \text{ Control bit}) & \leftarrow C
\end{align*}
\]

**Registers Altered**

- MSR[Carry]

**Latency**

2 cycles. For blocking accesses, MicroBlaze will first stall until space is available on the FSL interface.
rsub

Arithmetic Reverse Subtract

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsub</td>
<td>rD, rA, rB  Subtract</td>
</tr>
<tr>
<td>rsubc</td>
<td>rD, rA, rB  Subtract with Carry</td>
</tr>
<tr>
<td>rsubk</td>
<td>rD, rA, rB  Subtract and Keep Carry</td>
</tr>
<tr>
<td>rsubkc</td>
<td>rD, rA, rB  Subtract with Carry and Keep Carry</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

```
if C = 0 then
    (rD) ← (rB) + (rA) + 1
else
    (rD) ← (rB) + (rA) + MSR[C]
if K = 0 then
    MSR[C] ← CarryOut
```

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

In subtractions, Carry = \( \overline{\text{Borrow}} \). 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.
### rsubi

#### Arithmetic Reverse Subtract Immediate

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

#### Description

The contents of register rA is subtracted from the value of IMM, sign-extended to 32 bits, and the result is placed into register rD. Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic rsubik. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic rsubic. Both bits are set to a one for the mnemonic rsubikc.

When an rsubi instruction has bit 3 set (rsubik, rsubikc), the carry flag will keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (rsubi, rsubic), then the carry flag will be affected by the execution of the instruction. When bit 4 of the instruction is set to a one (rsubic, rsubikc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (rsubi, rsubik), the content of the carry flag does not affect the execution of the instruction (providing a normal subtraction).

#### Pseudocode

```plaintext
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 = (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

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

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

Registers Altered

- PC
- MSR[BIP]

Latency

2 cycles

Note

Convention is to use general purpose register r16 as rA.
rtid
Return from Interrupt

```
rtid       rA, IMM
```

<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

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

Registers Altered

- PC
- MSR[IE]

Latency

2 cycles

Note

Convention is to use general purpose register r14 as rA.
Return from Exception

**rted**

**Description**

Return from exception will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits. The instruction will also enable exceptions after execution.

This instruction always has a delay slot. The instruction following the RTED is always executed before the branch target.

**Pseudocode**

\[
\begin{align*}
\text{PC} & \leftarrow (\text{rA}) + \text{sext(IMM)} \\
& \text{allow following instruction to complete execution} \\
\text{MSR}[\text{EE}] & \leftarrow 1 \\
\text{MSR}[\text{EIP}] & \leftarrow 0 \\
\text{ESR} & \leftarrow 0
\end{align*}
\]

**Registers Altered**

- PC
- MSR[EE]
- MSR[EIP]
- ESR

**Latency**

2 cycles

**Note**

Convention is to use general purpose register r17 as rA. This instruction requires that one or more of the MicroBlaze parameters C\_\_\_EXCEPTION are set to 1.
### rtso

Return from Subroutine

**rtso** rA, IMM

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

\[
PC \leftarrow (rA) + \text{sext(IMM)} \\
\text{allow following instruction to complete execution}
\]

### Registers Altered

- PC

### Latency

2 cycles

### Note

Convention is to use general purpose register r15 as rA
sb  

**Store Byte**

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

### Description

Stores the contents of the least significant byte of register rD, into the memory location that results from adding the contents of registers rA and rB.

### Pseudocode

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

### Registers Altered

- None

### Latency

2 cycles
sb

**sbi** Store Byte Immediate

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

<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**

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

**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**  \( r_D, r_A \)

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

#### Description

This instruction sign-extends a halfword (16 bits) into a word (32 bits). Bit 16 in \( r_A \) will be copied into bits 0-15 of \( r_D \). Bits 16-31 in \( r_A \) will be copied into bits 16-31 of \( r_D \).

#### Pseudocode

\[
(\text{rD})[0:15] \leftarrow (\text{rA})[16] \\
(\text{rD})[16:31] \leftarrow (\text{rA})[16:31]
\]

#### Registers Altered

- \( r_D \)

#### Latency

1 cycle
**sext8**  
**Sign Extend Byte**

\[ \text{sext8} \quad rD, rA \]

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

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

**Registers Altered**

- rD

**Latency**

1 cycle
### sh

**Store Halfword**

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

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

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

**Registers Altered**
- None

**Latency**
- 2 cycles
## shi

**Store Halfword Immediate**

### Syntax

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

<table>
<thead>
<tr>
<th>1</th>
<th>1</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

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

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

\[
\text{Addr}[31] \leftarrow 0
\]

\[
\text{Mem(Addr)} \leftarrow (\text{rD})[16:31]
\]

### 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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</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

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

Registers Altered
- rD
- MSR[C]

Latency
1 cycle
Shift Right with Carry

src rD, rA

<table>
<thead>
<tr>
<th>1 0 0 1 0 0</th>
<th>rD</th>
<th>rA</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1</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

Shifts the contents of register rA, one bit to the right, and places the result in rD. The Carry flag is shifted in the shift chain and placed in the most significant bit of rD. The least significant bit coming out of the shift chain is placed in the Carry flag.

Pseudocode

\[
(rD)[0] \leftarrow \text{MSR}[C] \\
(rD)[1:31] \leftarrow (rA)[0:30] \\
\text{MSR}[C] \leftarrow (rA)[31]
\]

Registers Altered

- rD
- MSR[C]

Latency

1 cycle
srl

Shift Right Logical

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

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

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle
SW

Store Word

\texttt{sw} \quad \texttt{rD}, \texttt{rA}, \texttt{rB}

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

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

Registers Altered
- None

Latency
- 2 cycles
Instructions

swi

Store Word Immediate

swi 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></td>
<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 register rD, into the word aligned memory location that results from adding the contents of registers rA and the value IMM, sign-extended to 32 bits.

Pseudocode

Addr ← (rA) + sext(IMM)
Addr[30:31] ← 00
Mem(Addr) ← (rD)[0:31]

Register Altered

• None

Latency

2 cycles

Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
**wdc**

**Write to Data Cache**

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

<table>
<thead>
<tr>
<th>1 0 0 1 0 0</th>
<th>rA</th>
<th>rA</th>
<th>rB</th>
<th>0 0 0 0 1 1 0 0 1 0 0</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 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.

When caching over the CacheLink interface MicroBlaze uses a 4 word cache line. This means that only the valid and lock bits for a whole cache line can be written. WDC should not be used to initialise the cache when \text{C\_DCACHE\_USE\_FSL} is set to 1.

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

**Pseudocode**

\[
\begin{align*}
(D\text{Cache Tag}) & \leftarrow (rA) \\
(D\text{Cache Data}) & \leftarrow (rB)
\end{align*}
\]

**Registers Altered**

- None

**Latency**

1 cycle
**wic**  
Write to Instruction Cache

`wic` rA,rB

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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.

When caching over the CacheLink interface MicroBlaze uses a 4 word cache line. This means that only the valid and lock bits for a whole cache line can be written. WIC should not be used to initialize the cache when C_ICACHE_USE_FSL is set to 1.

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

**Pseudocode**

```
(ICache Tag) ← (rA)
(ICache Data) ← (rB)
```

**Registers Altered**

- None

**Latency**

1 cycle
## xor

**Logical Exclusive OR**

### xor

```
xor rD, rA, rB
```

### Description

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

### Pseudocode

```
(rD) ← (rA) ⊕ (rB)
```

### Registers Altered

- `rD`

### Latency

1 cycle
Instructions

xori  Logical Exclusive OR with Immediate

xori rA, rD, IMM

```
1 0 1 0 1 0  rD  rA  IMM
0 6 11 16 31
```

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.