The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/01/02</td>
<td>1.0 Xilinx EDK 3.1 release</td>
</tr>
<tr>
<td>03/11/03</td>
<td>2.0 Xilinx EDK 3.2 release</td>
</tr>
<tr>
<td>09/24/03</td>
<td>3.0 Xilinx EDK 6.1 release</td>
</tr>
<tr>
<td>02/20/04</td>
<td>3.1 Xilinx EDK 6.2 release</td>
</tr>
<tr>
<td>08/24/04</td>
<td>4.0 Xilinx EDK 6.3 release</td>
</tr>
</tbody>
</table>
Table of Contents

Preface: About This Guide

Manual Contents .................................................. 7
Additional Resources .............................................. 7
Conventions .......................................................... 8
  Typographical ..................................................... 8
  Online Document ................................................. 9

Chapter 1: MicroBlaze Architecture

Overview ............................................................ 11
  Features .......................................................... 11
Data Types and Endianness ...................................... 12
Instructions ......................................................... 13
Registers ............................................................ 18
  General Purpose Registers ................................... 18
  Special Purpose Registers .................................... 19
Pipeline .............................................................. 23
  Pipeline Architecture ......................................... 23
  Branches .......................................................... 24
Memory Architecture ............................................. 24
Reset, Interrupts, Exceptions and Breaks ....................... 25
  Reset ............................................................. 25
  Interrupt ......................................................... 26
  User Vector (Exception) ....................................... 26
  Hardware Exceptions ......................................... 27
  Breaks ............................................................ 27
Instruction Cache ................................................ 28
  Overview ......................................................... 28
  Instruction Cache Organization .............................. 28
  General Instruction Cache Functionality ................... 28
  Instruction Cache Operation .................................. 29
  Instruction Cache Software Support .......................... 30
Data Cache .......................................................... 31
  Overview ......................................................... 31
  Data Cache Organization ...................................... 31
  General Data Cache Functionality ............................ 31
  Data Cache Operation ......................................... 32
  Data Cache Software Support ................................ 33
Fast Simplex Link (FSL) ......................................... 34
  Hardware Acceleration using FSL ............................ 34
Debug and Trace ................................................... 34
  Debug Overview ................................................ 34
  Trace Overview ................................................. 35
Chapter 2: MicroBlaze Signal Interface Description

Overview .................................................. 37
Features ..................................................... 37
MicroBlaze I/O Overview ................................ 37
On-Chip Peripheral Bus (OPB) Interface Description .............................................. 40
Local Memory Bus (LMB) Interface Description .......................................................... 43
  LMB Signal Interface ........................................ 43
  LMB Transactions ......................................... 45
  Read and Write Data Steering .............................. 47
Fast Simplex Link (FSL) Interface Description .......................................................... 48
  Master FSL Signal Interface ............................... 48
  Slave FSL Signal Interface ............................... 49
  FSL Transactions .......................................... 49
Xilinx CacheLink (XCL) Interface Description ....................................................... 49
  CacheLink Signal Interface ............................... 50
  CacheLink Transactions .................................. 51
Debug Interface Description .................................................................................. 52
Trace Interface Description ................................................................................ 53
MicroBlaze Core Configurability ............................................................................. 54

Chapter 3: MicroBlaze Application Binary Interface

Scope .................................................................. 59
Data Types ....................................................... 59
Register Usage Conventions ............................................................................. 59
Stack Convention ................................................ 61
  Calling Convention ........................................ 62
Memory Model .................................................. 62
  Small data area ............................................. 62
  Data area ....................................................... 63
  Common un-initialized area .............................. 63
  Literals or constants ...................................... 63
Interrupt and Exception Handling ................................................................. 63

Chapter 4: MicroBlaze Instruction Set Architecture

Summary .......................................................... 65
Notation ........................................................... 65
Formats ........................................................... 66
Instructions ....................................................... 66
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>
<tbody>
<tr>
<td>Tutorials</td>
<td>Tutorials covering Xilinx design flows, from design entry to verification and debugging <a href="http://support.xilinx.com/support/techsup/tutorials/index.htm">http://support.xilinx.com/support/techsup/tutorials/index.htm</a></td>
</tr>
<tr>
<td>Answer Browser</td>
<td>Database of Xilinx solution records                                               <a href="http://support.xilinx.com/xlnx/xil_ans_browser.jsp">http://support.xilinx.com/xlnx/xil_ans_browser.jsp</a></td>
</tr>
<tr>
<td>Application Notes</td>
<td>Descriptions of device-specific design techniques and approaches                 <a href="http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes">http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes</a></td>
</tr>
<tr>
<td>Data Book</td>
<td>Pages from The Programmable Logic Data Book, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging <a href="http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp">http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp</a></td>
</tr>
</tbody>
</table>
### Conventions

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

#### Typographical

The following typographical conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning or Use</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier font</td>
<td>Messages, prompts, and program files that the system displays</td>
<td>speed grade:  - 100</td>
</tr>
<tr>
<td>Courier bold</td>
<td>Literal commands that you enter in a syntactical statement</td>
<td>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

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

MicroBlaze Architecture

Overview

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

Features

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

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- Separate 32-bit instruction and data buses that conform to IBM's OPB (On-chip Peripheral Bus) specification
- Separate 32-bit instruction and data buses with direct connection to on-chip block RAM through a LMB (Local Memory Bus)
- 32-bit address bus
- Single issue pipeline
- Instruction cache
- Data cache
- Hardware debug logic
- Fast Simplex Link (FSL) support
- Hardware multiplier (in Virtex-II and subsequent devices)
- Hardware exception handling
- Dedicated Cache Link interface for enhanced cache performance
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 1-1: 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 significance</td>
<td>MSByte</td>
<td>MSByte</td>
<td>LSByte</td>
<td></td>
</tr>
<tr>
<td>Bit label</td>
<td>0</td>
<td></td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Bit significance</td>
<td>MSBit</td>
<td>LSBit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1-2: Half Word Data Type**

<table>
<thead>
<tr>
<th>Byte address</th>
<th>n</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte label</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Byte significance</td>
<td>MSByte</td>
<td>LSByte</td>
</tr>
<tr>
<td>Bit label</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Bit significance</td>
<td>MSBit</td>
<td>LSBit</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 1-5 lists the MicroBlaze instruction set. Refer to Chapter 4, “MicroBlaze Instruction Set Architecture”, for more information on these instructions. Table 1-4 describes the instruction set nomenclature used in the semantics of each instruction.

<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>Imm</td>
<td>16 bit immediate value</td>
</tr>
<tr>
<td>Immx</td>
<td>x bit immediate value</td>
</tr>
<tr>
<td>FSLx</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>
</tbody>
</table>
### Table 1-5: 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>00000000001</td>
<td>Rd := Rb + Rb + 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 + Rb + 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 + Rb + 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 + Rb + 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 + Rb + C</td>
</tr>
<tr>
<td>CMP Rd, Ra, Rb</td>
<td>001011</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>001011</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000000</td>
<td>Rd := Rb + Ra + 1(unsigned)</td>
</tr>
<tr>
<td>ADDI Rd, Ra, Imm</td>
<td>001000</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra</td>
</tr>
<tr>
<td>RSUBI Rd, Ra, Imm</td>
<td>001001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra + 1</td>
</tr>
<tr>
<td>ADDIC Rd, Ra, Imm</td>
<td>001010</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra + C</td>
</tr>
<tr>
<td>RSUBIC Rd, Ra, Imm</td>
<td>001011</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra + C</td>
</tr>
<tr>
<td>ADDIK Rd, Ra, Imm</td>
<td>001100</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra</td>
</tr>
<tr>
<td>RSUBIK Rd, Ra, Imm</td>
<td>001101</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra + 1</td>
</tr>
<tr>
<td>ADDIKC Rd, Ra, Imm</td>
<td>001110</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra + C</td>
</tr>
<tr>
<td>RSUBIKC Rd, Ra, Imm</td>
<td>001111</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000</td>
<td>Rd := s(Imm) + Ra + C</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>010001</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>BSSL Rd, Ra, Rb</td>
<td>010001</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>00000000000 &amp; Imm5</td>
<td>Rd := Ra &gt;&gt; Imm5</td>
</tr>
<tr>
<td>BSRLI Rd, Ra, Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000000000 &amp; Imm5</td>
<td>Rd := Ra &gt;&gt; Imm5</td>
</tr>
<tr>
<td>BSRAI Rd, Ra, Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00000010000 &amp; Imm5</td>
<td>Rd := Ra[0], (Ra &gt;&gt; Imm5)</td>
</tr>
<tr>
<td>BSLLI Rd, Ra, Imm</td>
<td>011001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>00001000000 &amp; Imm5</td>
<td>Rd := Ra[0], (Ra &gt;&gt; Imm5)</td>
</tr>
<tr>
<td>IDIV Rd, Ra, Rb</td>
<td>010010</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>010010</td>
<td>Rd</td>
<td>Ra</td>
<td>Rb</td>
<td>00000000001</td>
<td>Rd := Rb/Ra, unsigned</td>
</tr>
<tr>
<td>GET Rd, FSLx</td>
<td>011011</td>
<td>Rd</td>
<td>00000000000 &amp; FSLx</td>
<td>00000000000 &amp; FSLx</td>
<td>Rd := FSLx (blocking data read)</td>
<td>MSR[FSL] := FSLx_S_Control</td>
</tr>
<tr>
<td>Instruction</td>
<td>Semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUT Ra,FSLx</td>
<td>FSLx := Ra (blocking data write)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NGET Rd,FSLx</td>
<td>Rd := FSLx (non-blocking data read) MSR[FSL] := FSLx_S_Control MSR[C] := not FSLx_S_Exists</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NPUT Ra,FSLx</td>
<td>FSLx := Ra (non-blocking data write) MSR[C] := FSLx_M_Full</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGET Rd,FSLx</td>
<td>Rd := FSLx (blocking control read) MSR[FSL] := not FSLx_S_Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUT Ra,FSLx</td>
<td>FSLx := Ra (blocking control write)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCGET Rd,FSLx</td>
<td>Rd := FSLx (non-blocking control read) MSR[FSL] := not FSLx_S_Control MSR[C] := not FSLx_S_Exists</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCPUT Ra,FSLx</td>
<td>FSLx := Ra (non-blocking control write) MSR[C] := FSLx_M_Full</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR Rd,Ra,Rb</td>
<td>Rd := Ra or Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND Rd,Ra,Rb</td>
<td>Rd := Ra and Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR Rd,Ra,Rb</td>
<td>Rd := Ra xor Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANDN Rd,Ra,Rb</td>
<td>Rd := Ra and Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRA Rd,Ra</td>
<td>Rd := Ra[0], (Ra &gt;&gt; 1); C := Ra[31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC Rd,Ra</td>
<td>Rd := C, (Ra &gt;&gt; 1); C := Ra[31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL Rd,Ra</td>
<td>Rd := 0, (Ra &gt;&gt; 1); C := Ra[31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEXT16 Rd,Ra</td>
<td>Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIC Ra,Rb</td>
<td>ICache_Tag := Ra, ICache_Data := Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDC Ra,Rb</td>
<td>DCache_Tag := Ra, DCache_Data := Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTS Sd,Ra</td>
<td>Sd := Ra , where Sd=001 is MSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MFS Rd,Sa</td>
<td>Rd := Sa , where Sa=000 is PC, 001 is MSR, 011 is EAR, and 101 is ESR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSRCLR Rd,Imm</td>
<td>Rd := MSR; MSR := MSR ^ Imm14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSRSET Rd,Imm</td>
<td>Rd := MSR; MSR := MSR ^ Imm14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR Rb</td>
<td>PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRD Rb</td>
<td>PC := PC + Rb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1-5: **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>BRLD Rd,Rb</td>
<td>100110</td>
<td>Rd</td>
<td>10100</td>
<td>Rb</td>
<td>00000000000</td>
<td>PC := PC + Rb; Rd := PC</td>
</tr>
<tr>
<td>BRA Rd</td>
<td>100110</td>
<td>00000</td>
<td>01000</td>
<td>Rb</td>
<td>000000000000</td>
<td>PC := Rb</td>
</tr>
<tr>
<td>BRAD Rd</td>
<td>100110</td>
<td>00000</td>
<td>11000</td>
<td>Rb</td>
<td>0000000000000</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>0000000000000</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>000000000000000</td>
<td>PC := Rb; Rd := PC; MSR[BIP] := 1</td>
</tr>
<tr>
<td>BEQ Ra,Rb</td>
<td>100111</td>
<td>00000</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BNE Ra,Rb</td>
<td>100111</td>
<td>00001</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra /= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLT Ra,Rb</td>
<td>100111</td>
<td>00010</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra &lt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLE Ra,Rb</td>
<td>100111</td>
<td>00010</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra &lt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGT Ra,Rb</td>
<td>100111</td>
<td>00100</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra &gt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGE Ra,Rb</td>
<td>100111</td>
<td>00101</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra &gt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BEQD Ra,Rb</td>
<td>100111</td>
<td>10000</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra = 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BNED Ra,Rb</td>
<td>100111</td>
<td>10001</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra /= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLTD Ra,Rb</td>
<td>100111</td>
<td>10100</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra &lt; 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BLED Ra,Rb</td>
<td>100111</td>
<td>10111</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</td>
<td>if Ra &lt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>BGTD Ra,Rb</td>
<td>100111</td>
<td>10100</td>
<td>Ra</td>
<td>Rb</td>
<td>000000000000000</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>000000000000000</td>
<td>if Ra &gt;= 0: PC := PC + Rb</td>
</tr>
<tr>
<td>ORI Rd,Ra,Imm</td>
<td>101000</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra or s(Imm)</td>
<td></td>
</tr>
<tr>
<td>ANDI Rd,Ra,Imm</td>
<td>101001</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra and s(Imm)</td>
<td></td>
</tr>
<tr>
<td>XORI Rd,Ra,Imm</td>
<td>101010</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra xor s(Imm)</td>
<td></td>
</tr>
<tr>
<td>ANDNI Rd,Ra,Imm</td>
<td>101101</td>
<td>Rd</td>
<td>Ra</td>
<td>Imm</td>
<td>Rd := Ra and s(Imm)</td>
<td></td>
</tr>
<tr>
<td>IMM Imm</td>
<td>101100</td>
<td>00000</td>
<td>00000</td>
<td>Imm</td>
<td>Imm[0:15] := 1imm</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[EE] := 1, MSR[EIP] := 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>11100</td>
<td>Imm</td>
<td>PC := s(Imm); Rd := PC</td>
<td></td>
</tr>
</tbody>
</table>
### Table 1-5: 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>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>Ra</td>
<td>00000</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>Ra</td>
<td>00001</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>Ra</td>
<td>00010</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>Ra</td>
<td>00011</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>Ra</td>
<td>00100</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>Ra</td>
<td>00101</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>Ra</td>
<td>10000</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>Ra</td>
<td>10001</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>Ra</td>
<td>10010</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>Ra</td>
<td>10011</td>
<td>Imm</td>
<td>if Ra &lt;= 0: PC := PC + s(Imm)</td>
<td></td>
</tr>
<tr>
<td>BGTID Ra,Imm</td>
<td>101111</td>
<td>Ra</td>
<td>10100</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>Ra</td>
<td>10101</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>
</tbody>
</table>
**Registers**

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

**General Purpose Registers**

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

### Table 1-5: 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>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>

### Table 1-6: General Purpose Registers (R0-R31)

<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>R0</td>
<td>R0 is defined to always have the value of zero. Anything written to R0 is discarded.</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0:31</td>
<td>R1 through R31</td>
<td>R1 through R31 are 32-bit general purpose registers</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0:31</td>
<td>R14</td>
<td>32-bit used to store return addresses for interrupts</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0:31</td>
<td>R15</td>
<td>32-bit general purpose register</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>
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 $S_a = 00000$, or $S_a = rpc$.

<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>R16</td>
<td>32-bit used to store return addresses for breaks</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0:31</td>
<td>R17</td>
<td>If MicroBlaze is configured to support hardware exceptions, this register is loaded with HW exception return address; if not it is a general purpose register</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0:31</td>
<td>R18 through R31</td>
<td>R18 through R31 are 32-bit general purpose registers.</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Table 1-6: General Purpose Registers (R0-R31) (Continued)

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 to with an MTS instruction or with the dedicated MSRSET and MSRCLR instructions. Writes to MSR are delayed one clock cycle. When writing to MSR using MTS, the value written takes effect one clock cycle after executing the MTS instruction. Any value written to bit 0 is discarded. When used with an MTS or MFS instruction the MSR register is specified by setting $S_x = 00001$, or $S_x = rmsr$.

<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>

Table 1-7: Program Counter (PC)

Figure 1-3: PC
### Table 1-8: Machine Status Register (MSR)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
</table>
| 0    | CC   | Arithmetic Carry Copy  
|      |      | Copy of the Arithmetic Carry (bit 29). Read-only. | 0 |
| 1:21 | Reserved | | |
| 22   | EIP  | Exception In Progress | 0 |
|      |      | 0 No hardware exception in progress  
|      |      | 1 Hardware exception in progress Read/Write | |
| 23   | EE   | Exception Enable | 0 |
|      |      | 0 Hardware exceptions disabled  
|      |      | 1 Hardware exceptions enabled Read/Write | |
| 24   | DCE  | Data Cache Enable | 0 |
|      |      | 0 Data Cache is Disabled  
|      |      | 1 Data Cache is Enabled Read/Write | |
| 25   | DZ   | Division by Zero¹ | 0 |
|      |      | 0 No division by zero has occurred  
|      |      | 1 Division by zero has occurred Read-only | |
| 26   | ICE  | Instruction Cache Enable | 0 |
|      |      | 0 Instruction Cache is Disabled  
|      |      | 1 Instruction Cache is Enabled Read/Write | |
| 27   | FSL  | FSL Error | 0 |
|      |      | 0 FSL get/put had no error  
|      |      | 1 FSL get/put had mismatch in instruction type and value type Read-only | |

¹ Division by zero is a hard fault and results in a reset.

**Figure 1-4: MSR**
Table 1-8: Machine Status Register (MSR) (Continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<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-only</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>C</td>
<td>Arithmetic Carry</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 No Carry (Borrow)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Carry (No Borrow)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>IE</td>
<td>Interrupt Enable</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Interrupts disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Interrupts enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>BE</td>
<td>Buslock Enable$^2$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 Buslock disabled on data-side OPB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Buslock enabled on data-side OPB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Buslock Enable does not affect operation of IXCL, DXCL, ILMB, DLMB, or IOPB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read/Write</td>
<td></td>
</tr>
</tbody>
</table>

1. This bit is not connected to the optional divide by zero exception handling. It 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 unaligned access exception. The contents of this register is undefined for all other exceptions. The register can be read with an MFS instruction. When used with the MFS instruction the EAR register is specified by setting Sa = 00011, or Sa = rear.
Chapter 1: MicroBlaze Architecture

Exception Status Register (ESR)

The Exception Status Register contains status bits for the processor. It can be read with an MFS instruction. When used with the MFS instruction the ESR register is specified by setting Sa = 00101, or Sa = resr.

Table 1-9: Machine Status 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>

Table 1-10: 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></td>
<td>Reserved</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 1-11.</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>Read-only</td>
<td></td>
</tr>
</tbody>
</table>
This section describes the MicroBlaze pipeline architecture.

**Pipeline Architecture**

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

- Fetch
- Decode
- 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>Exception Cause</th>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
</table>
| Unaligned Data Access    | 20   | W                        | Word Access Exception
  - 0 unaligned halfword access
  - 1 unaligned word access | 0          |
|                          | 21   | S                        | Store Access Exception
  - 0 unaligned load access
  - 1 unaligned store access | 0          |
|                          | 22:26| Rx                       | Source/Destination Register
  General purpose register used as source (Store) or destination (Load) in unaligned access | 0          |
| Illegal Instruction      | 20:26| Reserved                 |                                                                             | 0           |
| Instruction bus error    | 20:26| Reserved                 |                                                                             | 0           |
| Data bus error           | 20:26| Reserved                 |                                                                             | 0           |
| Divide by zero           | 20:26| Reserved                 |                                                                             | 0           |
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 e.g., software debugging.

Both instruction and data interfaces of MicroBlaze are 32 bit wide and use big endian (reverse bit order) 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 (available in MicroBlaze v3.00a and higher). 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, only available in v3.00a or higher). 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”.

1-800-255-7778 UG081 (v4.0) August 24, 2004
Reset, Interrupts, Exceptions and Break

All versions of MicroBlaze supports reset, interrupt, user exception and break. Starting with version 3.00a, MicroBlaze can also be configured to support 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 1-12 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 R14</td>
<td></td>
</tr>
<tr>
<td>Break: Non-maskable hardware</td>
<td>0x00000018 - 0x0000001C</td>
<td>R16</td>
</tr>
<tr>
<td>Break: Hardware Break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Break: Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Exception</td>
<td>0x00000020 - 0x00000024 R17</td>
<td></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).

Equivalent Pseudocode

```plaintext
PC ← 0x00000000
```

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

Interrupts are ignored by the processor if the break in progress (BIP) bit in the MSR register 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 1-13 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 1-13: 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 divider</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

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

User Vector (Exception)

The user exception vector is located at address 0x8. A user exception is easiest caused by inserting a 'BRAILD 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

```plaintext
rx ← PC
PC ← 0x00000008
```
Hardware Exceptions

MicroBlaze v3.00a and higher can be configured to detect different internal error conditions: illegal instruction, instruction and data bus error, unaligned access and divide by zero. 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.

Equivalent Pseudocode

\[
\begin{align*}
  r17 & \leftarrow \text{PC} \\
  \text{PC} & \leftarrow 0x00000020 \\
  \text{MSR}[\text{EE}] & \leftarrow 0 \\
  \text{MSR}[\text{EIP}] & \leftarrow 1
\end{align*}
\]

Breaks

There are two kinds of breaks:

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

Software Breaks

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

Hardware Breaks

Hardware breaks are performed by asserting the external break signal (i.e. the \texttt{Ext\_BRK} and \texttt{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 \texttt{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 \texttt{Ext\_NM\_BRK} input port) will always be handled immediately.

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 1-13 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

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

Overview

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

The instruction cache has the following features:
- Direct mapped (1-way associative)
- User selectable cacheable memory area
- Configurable cache 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 new bit in the MSR register
- 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 (only available in MicroBlaze v3.00a and higher). 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.
- 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 is split into two segments - a cacheable segment and a non-cacheable segment. The cacheable segment is determined by two parameters, C_ICACHE_BASEADDR and C_ICACHE_HIGHADDR. All addresses within this range correspond to the cacheable address space segment. All other addresses are non-cacheable.
Instruction Cache 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 7 to 14 bits. This results in cache sizes ranging from 512B to 64 kB\(^1\). 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_ICACHE_BYTE_SIZE=4096; the cacheable byte address range is 16 bits, and the cache word address range is 12 bits (i.e. a 10 bit cache word address), thus the required address tag is: 16-12=4 bits.

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 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 instruction is in the cache. The lookup is successful when both the valid bit is set and the tag address is the same as the tag address segment of the instruction address.

---

1. The size of the cache is FPGA architecture dependent. The MicroBlaze instruction cache can be configured to use between 1 and 32 RAMB primitives. The actual cache size therefore depends on the RAMB size in the targeted architecture.
If the instruction is in the cache, the cache will drive the ready signal (Cache_Hit) for MicroBlaze and the instruction data for the address. If the instruction is not in the cache, the cache will not drive the ready signal but will wait until the OPB fulfills the request and updates the cache with the new information.

Instruction Cache Software Support

MSR Bit

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

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

WIC Instruction

The WIC instruction may be used to update the instruction cache from a software program. For a detailed description, please refer to Chapter 4, “MicroBlaze Instruction Set Architecture”.

HW Debug Logic

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

Lock Bit

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

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

Overview

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

The data cache has the following features:

- 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 new bit in the MSR register
- Instructions to write to the data cache
- Memory is organized into a cacheable and a non-cacheable segments

Data Cache Organization

MicroBlaze can be configured to cache data over either the OPB interface, or the dedicated Xilinx CacheLink interface (only available in MicroBlaze v3.00a and higher). 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). OPB caches use single word cache lines.
- 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 in split into two segments - a cacheable segment and a non-cacheable segment. The cacheable area is determined by two parameters, C_DCACHE_BASEADDR and C_DCACHE_HIGHADDR. All addresses within this range correspond to the cacheable address space. 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 9 to 14 bits. This results in a cache sizes ranging from 2 kB to 64 kB\(^1\). 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 \( \text{C\_DCACHE\_BASEADDR}=0x00400000, \text{C\_DCACHE\_HIGHADDR}=0x00403fff, \text{and} \text{C\_DCACHE\_BYTE\_SIZE}=2048; \) the cacheable byte address range is 14 bits, and the cache word address range is 11 bits (i.e. a 9 bit cache word address), thus the required address tag is 14-11=3 bits.

**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 loaded 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 secondly if the address is in the data cache. If that case, the data is fetch from the data 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.
Data Cache

If the read data is in the cache, the cache will drive the ready signal (Cache_Hit) for MicroBlaze and the data for the address. If the read data is not in the cache, the cache will not drive the ready signal but will:

- for OPB caching; wait until the OPB fulfills the speculative read request
- for CacheLink caching; send a cache line request over the CacheLink interface.

**Data Cache Software Support**

**MSR Bit**

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

The contents of the cache are preserved by default when the cache is disabled. The cache cannot be turned on/off from an interrupt handler routine as the changes to the MSR is lost once the interrupt is handled (the MSR state is restored after interrupt handling).

**WDC Instruction**

The WDC instruction may be used to update the data cache from a software program. For a detailed description, please refer to Chapter 4, “MicroBlaze Instruction Set Architecture”.

**HW Debug Logic**

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

**Lock Bit**

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

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

MicroBlaze contains eight Fast Simplex Link (FSL) interfaces, each consisting of one input and one output port. The FSL channels are dedicated uni-directional 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 MicroBlaze ISA used to transfer information from an FSL port to a general purpose register. The put instruction is used for transfer 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 1-11.

Example code:

```c
// Configure f_x
put FSLx, Rc
// Store operands
put FSLx, Ra // op 1
put FSLx, Rb // op 2
// Load result
get FSLx, Rt
```

Figure 1-11: 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 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 MicroBlaze
• Read and write memory and all registers including PC and MSR
• Support for multiple processors
• Write to Instruction and data cache

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 tree memory interfaces are supported: Local Memory Bus (LMB), IBM’s On-chip Peripheral Bus (OPB) and Xilinx CacheLink (XCL, only in MicroBlaze v3.00a and higher). The LMB provides single-cycle access to on-chip dual-port block RAM. The OPB interface provides a connection to both on-and off-chip peripherals and memory. 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 controller
- 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:

- DOPB: Data interface, On-chip Peripheral Bus
- DLMB: Data interface, Local Memory Bus (BRAM only)
- IOPB: Instruction interface, On-chip Peripheral Bus
- ILMB: Instruction interface, Local Memory Bus (BRAM only)
- MFSL 0..7: FSL master interface
- SFSL 0..7: FSL slave interface
- IXCL: Instruction side Xilinx CacheLink interface (FSL master/slave pair)
- DXCL: Data side Xilinx CacheLink interface (FSL master/slave pair)
- Core: Miscellaneous signals for clock, reset, debug and trace
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 buslock</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 timeout error</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>
<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 buslock</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>
</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_RNW</td>
<td>IOPB</td>
<td>O</td>
<td>Instruction interface OPB read, not write (tied to ‘0’)</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 LB address bus</td>
</tr>
<tr>
<td>Byte_Enable[0:3]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB byte enables</td>
</tr>
<tr>
<td>Data_Write[0:31]</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB write data bus</td>
</tr>
<tr>
<td>D_AS</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB address strobe</td>
</tr>
<tr>
<td>Read_Strobe</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB read strobe</td>
</tr>
<tr>
<td>Write_Strobe</td>
<td>DLMB</td>
<td>O</td>
<td>Data interface LB write strobe</td>
</tr>
<tr>
<td>Data_Read[0:31]</td>
<td>DLMB</td>
<td>I</td>
<td>Data interface LB read data bus</td>
</tr>
<tr>
<td>DReady</td>
<td>DLMB</td>
<td>I</td>
<td>Data interface LB data ready</td>
</tr>
<tr>
<td>Instr_Addr[0:31]</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LB address bus</td>
</tr>
<tr>
<td>I_AS</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LB address strobe</td>
</tr>
<tr>
<td>IFetch</td>
<td>ILMB</td>
<td>O</td>
<td>Instruction interface LB instruction fetch</td>
</tr>
<tr>
<td>Instr[0:31]</td>
<td>ILMB</td>
<td>I</td>
<td>Instruction interface LB read data bus</td>
</tr>
<tr>
<td>IReady</td>
<td>ILMB</td>
<td>I</td>
<td>Instruction interface LB 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>
<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>
</tbody>
</table>
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 OPB data bus interconnects are illustrated in Figure 2-2. The write data bus (from masters and bridges) is separated from the read data bus (from slaves and bridges) to break up the bus OR logic. In minimal cases this can completely eliminate the OR logic for the read or write data buses. Optionally, you can "OR" together the read and write buses to create the correct functionality for the OPB bus monitor. Note that the instruction-side OPB contains a write data bus (tied to 0x00000000) and a RNW signal (tied to logic 1) so that its interface remains consistent with the data-side OPB. These signals are constant and generally are minimized in implementation.

A multi-ported slave is used instead of a bridge in the example shown in Figure 2-3. This could represent a memory controller with a connection to both the IOPB and the DOPB. In this case, the bus multiplexing and prioritization must be done in the slave. The advantage of this approach is that a separate I-to-D bridge and an OPB arbiter on the instruction side are not required. The arbiter function must still exist in the slave device.
Figure 2-2: OPB Interconnection (breaking up read and write buses)
Figure 2-3: OPB Interconnection (with multi-ported slave and no bridge)
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 is 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>not used</td>
<td>O</td>
<td>Byte enables</td>
</tr>
<tr>
<td>Data_Write[0:31]</td>
<td>Data_Write[0:31]</td>
<td>not used</td>
<td>O</td>
<td>Write data bus</td>
</tr>
<tr>
<td>AS</td>
<td>D_AS</td>
<td>I_AS</td>
<td>O</td>
<td>Address strobe</td>
</tr>
<tr>
<td>Read_Strobe</td>
<td>Read_Strobe</td>
<td>IFetch</td>
<td>O</td>
<td>Read in progress</td>
</tr>
<tr>
<td>Write_Strobe</td>
<td>Write_Strobe</td>
<td>not used</td>
<td>O</td>
<td>Write in progress</td>
</tr>
<tr>
<td>Data_Read[0:31]</td>
<td>Data_Read[0:31]</td>
<td>Instr[0:31]</td>
<td>I</td>
<td>Read data bus</td>
</tr>
<tr>
<td>Ready</td>
<td>DReady</td>
<td>IReady</td>
<td>I</td>
<td>Ready for next transfer</td>
</tr>
<tr>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>I</td>
<td>Bus clock</td>
</tr>
</tbody>
</table>

#### Addr[0:31]

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

#### Byte_Enable[0:3]

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2: MicroBlaze Signal Interface Description

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]

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.

| Table 2-3: Valid Values for Byte_Enable[0:3] |
| Byte Lanes Used |
|------------------|------------------|------------------|------------------|------------------|
| 1000             | x                |                 |                 |                 |
| 0011             | x                | x               | x               |                 |
| 1100             | x                |                 | x               | x               |
| 1111             | x                | x               | x               | x               |

<table>
<thead>
<tr>
<th>Bit Values</th>
<th>Byte Lanes Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data[0:7]</td>
</tr>
<tr>
<td>1</td>
<td>Data[8:15]</td>
</tr>
<tr>
<td>2</td>
<td>Data[16:23]</td>
</tr>
<tr>
<td>3</td>
<td>Data[24:31]</td>
</tr>
</tbody>
</table>
LMB Transactions

The following diagrams provide examples of LMB bus operations.

Generic Write Operation

![Diagram of LMB Generic Write Operation]

**Figure 2-4:** LMB Generic Write Operation

Generic Read Operation

![Diagram of LMB Generic Read Operation]

**Figure 2-5:** LMB Generic Read Operation
Back-to-Back Write Operation (Typical LMB access - 2 clocks per write)

![LMB Back-to-Back Write Operation](image)

Single Cycle Back-to-Back Read Operation (Typical I-side access - 1 clock per read)

![LMB Single Cycle Back-to-Back Read Operation](image)
Back-to-Back Mixed Read/Write Operation (Typical D-side timing)

![Back-to-Back Mixed Read/Write Operation](image)

**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>Address [30:31]</th>
<th>Byte_Enable [0:3]</th>
<th>Transfer Size</th>
<th>Register rD Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0001</td>
<td>byte</td>
<td>Byte3</td>
</tr>
<tr>
<td>10</td>
<td>0010</td>
<td>byte</td>
<td>Byte2</td>
</tr>
<tr>
<td>01</td>
<td>0100</td>
<td>byte</td>
<td>Byte1</td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>byte</td>
<td>Byte0</td>
</tr>
<tr>
<td>10</td>
<td>0011</td>
<td>halfword</td>
<td>Byte2</td>
</tr>
<tr>
<td>00</td>
<td>1100</td>
<td>halfword</td>
<td>Byte0</td>
</tr>
<tr>
<td>00</td>
<td>1111</td>
<td>word</td>
<td>Byte0 Byte1 Byte2 Byte3</td>
</tr>
</tbody>
</table>

**Figure 2-8: Back-to-Back Mixed Read/Write Operation**
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.

### 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>
<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 that indicating data is ready to be written to the output FSL when set</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>

<table>
<thead>
<tr>
<th>Address [30:31]</th>
<th>Byte_Enable [0:3]</th>
<th>Transfer Size</th>
<th>Write Data Bus Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0001</td>
<td>byte</td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>10</td>
<td>0010</td>
<td>byte</td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>01</td>
<td>0100</td>
<td>byte</td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>byte</td>
<td>rD[24:31]</td>
</tr>
<tr>
<td>10</td>
<td>0011</td>
<td>halfword</td>
<td>rD[16:23] rD[24:31]</td>
</tr>
<tr>
<td>00</td>
<td>1100</td>
<td>halfword</td>
<td>rD[16:23] rD[24:31]</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 contents of the register file of MicroBlaze 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 into the register file of MicroBlaze. 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. It is available on MicroBlaze version v3.00a or higher. The CacheLink signalling protocol is implemented over a master/slave pair of fast simplex links (FSLs) for direct and streamed access to memory controllers supporting this new interface.
This interface is only available on MicroBlaze when caches are enabled, and supports the same Harvard architecture as the regular OPB caches. The selection between OPB and CacheLink cache controllers is individually controlled for instruction and data side caches using two new parameters: C_ICACHE_USE_FSL and C_DCACHE_USE_FSL. 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.

The CacheLink cache controllers handles 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>
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).

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>
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 debuggers simultaneously.

---

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 (v3.00a) 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 instruction</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 register 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>
<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 part of multi-cycle instruction execution</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>

Table 2-10: MicroBlaze Trace signals
Parameters valid for MicroBlaze v3.00a 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</td>
<td>virtex2</td>
<td>yes</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qvirtex2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>spartan2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>spartan2e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>spartan3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtex</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtex2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtex2p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtex4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtexe</td>
<td></td>
<td></td>
<td></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</td>
<td>microblaze</td>
<td>yes</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name</td>
<td></td>
<td></td>
<td></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>Barrel Shifter</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_USE_DIV</td>
<td>Divide Unit</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_USE_MSR_INSTR</td>
<td>Enable use of instructions:</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>MSRSET and MSRCLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_UNALIGNED_EXCEPTION</td>
<td>Enable exception handling for unaligned</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>data accesses</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_ILL_OPCODE_EXCEPTION</td>
<td>Enable exception handling for illegal</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>op-code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_IOPB_BUS_EXCEPTION</td>
<td>Enable exception handling for IOPB bus</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_DOPB_BUS_EXCEPTION</td>
<td>Enable exception handling for DOPB bus</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_DIV_ZERO_EXCEPTION</td>
<td>Enable exception handling for division by</td>
<td>0, 1</td>
<td>0</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>zero</td>
<td></td>
<td></td>
<td></td>
<td></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>
</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_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>0x3FFFFFF</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>0-24</td>
<td>7</td>
<td>yes</td>
<td>integer</td>
</tr>
<tr>
<td>C_CACHE_BYTE_SIZE</td>
<td>Instruction cache size</td>
<td>512, 1024, 2048, 4096, 8192, 16384, 32768, 65536</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192</td>
<td></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>
<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>0x3FFFFFF</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></td>
<td>integer</td>
</tr>
<tr>
<td>C_ALLOW_DCACHE_WR</td>
<td>Data cache write enable</td>
<td>0,1</td>
<td>1</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>C_DCACHE_ADDR_TAG</td>
<td>Data cache address tags</td>
<td>0-24</td>
<td>7</td>
<td>yes</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_BYTE_SIZE</td>
<td>Data cache size</td>
<td>2048, 4096, 8192, 16384, 32768, 65536&lt;sup&gt;2&lt;/sup&gt;</td>
<td>8192</td>
<td></td>
<td>integer</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></td>
<td>integer</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.
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.

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>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 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.
- Register R13 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.
- 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

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Enforcement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>Dedicated HW</td>
<td>Value 0</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>Dedicated SW</td>
<td>Stack Pointer</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>Dedicated SW</td>
<td>Read-only small data area anchor</td>
<td></td>
</tr>
<tr>
<td>R3-R4</td>
<td>VOLatile SW</td>
<td>Return Values/Temporaries</td>
<td></td>
</tr>
<tr>
<td>R5-R10</td>
<td>VOLatile SW</td>
<td>Passing parameters/Temporaries</td>
<td></td>
</tr>
<tr>
<td>R11-R12</td>
<td>VOLatile SW</td>
<td>Temporaries</td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>Dedicated SW</td>
<td>Read-write small data area anchor</td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>Dedicated HW</td>
<td>Return address for Interrupt</td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td>Dedicated SW</td>
<td>Return address for Sub-routine</td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td>Dedicated HW</td>
<td>Return address for Trap (Debugger)</td>
<td></td>
</tr>
<tr>
<td>R17</td>
<td>Dedicated HW</td>
<td>Return Address for Exceptions</td>
<td></td>
</tr>
<tr>
<td>R18</td>
<td>Dedicated SW</td>
<td>Reserved for Assembler</td>
<td></td>
</tr>
<tr>
<td>R19-R31</td>
<td>Non-volatile SW</td>
<td>Must be saved across function calls. Callee-save</td>
<td></td>
</tr>
<tr>
<td>RPC</td>
<td>Special HW</td>
<td>Program counter</td>
<td></td>
</tr>
<tr>
<td>RMSR</td>
<td>Special HW</td>
<td>Machine Status Register</td>
<td></td>
</tr>
</tbody>
</table>
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 has certain special registers such as a program counter (rpc) and machine status register (rmsr). These registers are not mapped directly to the register file and hence the usage of these registers is different from the general purpose registers. The value from rmsr and rpc can be transferred to general purpose registers by using \texttt{mts} and \texttt{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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Parameters for called sub-routine</td>
<td>(Arg n ..Arg1)</td>
</tr>
<tr>
<td>(Optional: Maximum number of arguments required for any called procedure from</td>
<td>current procedure.)</td>
</tr>
<tr>
<td>Old Stack Pointer</td>
<td>Link Register (R15)</td>
</tr>
<tr>
<td>Callee Saved Register (R31...R19)</td>
<td>(Optional: Only those registers which are used by the current procedure are saved)</td>
</tr>
<tr>
<td>Local Variables for Current Procedure</td>
<td>(Optional: Present only if Locals defined in the procedure)</td>
</tr>
<tr>
<td>Functional Parameters (Arg n .. Arg 1)</td>
<td>(Optional: Maximum number of arguments required for any called procedure from the current 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 to the comm area and can be accessed either using the absolute address or using the read-write small data area anchor R13.

**Literals or constants**

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

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

## Interrupt and Exception Handling

MicroBlaze assumes certain address locations for handling interrupts and exceptions as indicated in Table 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. In case of programs compiled without the `-xl-mode-xmdstub` compiler option, the *crt0.o* initialization file is passed by the mb-gcc compiler to the *mb-ld* linker for linking. This file sets the appropriate addresses of the exception handlers.

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

**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
```
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 1.

Table 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>¬</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

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.

**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.

### 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.

### Formats

**Table 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>
</tbody>
</table>

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

**Type B**

Type B is used for register-immediate instructions. It contains the opcode, one destination and one source registers, and a source 16-bit immediate value.

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

Instructions

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

**add**

**Arithmetic Add**

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

**Description**

The sum of the contents of registers rA and rB, is placed into register rD.

Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic addk. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic addc. Both bits are set to a one for the mnemonic addkc.

When an add instruction has bit 3 set (addk, addkc), the carry flag will keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (add, addc), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (addc, addkc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (add, addk), the content of the carry flag does not affect the execution of the instruction (providing a normal addition).

**Pseudocode**

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

**Registers Altered**

- rD
- MSR[C]

**Latency**

1 cycle

**Note**

The C bit in the instruction opcode is not the same as the carry bit in the MSR register.
**addi**

**Arithmetic Add Immediate**

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>format</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>rD, rA, IMM</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>addic</td>
<td>rD, rA, IMM</td>
<td>Add Immediate with Carry</td>
</tr>
<tr>
<td>addik</td>
<td>rD, rA, IMM</td>
<td>Add Immediate and Keep Carry</td>
</tr>
<tr>
<td>addikc</td>
<td>rD, rA, IMM</td>
<td>Add Immediate with Carry and Keep Carry</td>
</tr>
</tbody>
</table>

**Description**

The sum of the contents of registers rA and the value in the IMM field, sign-extended to 32 bits, is placed into register rD. Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic addik. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic addic. Both bits are set to a one for the mnemonic addikc.

When an addi instruction has bit 3 set (addik, addikc), the carry flag will Keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (addi, addic), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (addic, addikc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (addi, addik), the content of the carry flag does not affect the execution of the instruction (providing a normal addition).

**Pseudocode**

```plaintext
if C = 0 then
    (rD) ← (rA) + sext(IMM)
else
    (rD) ← (rA) + sext(IMM) + MSR[C]
if K = 0 then
    MSR[C] ← CarryOut
```

**Registers Altered**

- rD
- MSR[C]

**Latency**

1 cycle

**Notes**

The C bit in the instruction opcode is not the same as the carry bit in the MSR register.

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

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

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

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

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

Registers Altered
- \( rD \)

Latency
1 cycle
**andi**

**Logial AND with Immediate**

`andi rD, rA, IMM`

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

**Description**

The 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

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

<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>
<tr>
<td>21</td>
<td>31</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

\textbf{andni} \hspace{1em} rD, rA, IMM

\begin{tabular}{|c|c|c|c|}
\hline
1 & 0 & 1 & 0 & 1 & 1 & rD & rA
\hline
0 & 6 & 11 & 16 & IMM
\hline
\end{tabular}

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

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

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

beq rA, rB Branch if Equal
beqd rA, rB Branch if Equal with Delay

Description

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

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

Pseudocode

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

Registers Altered

• PC

Latency

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

Branch Immediate if Equal

**beqi** \( rA, \text{IMM} \)  Branch Immediate if Equal

**beqid** \( rA, \text{IMM} \)  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>0</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 \( \text{PC} + \text{IMM} \).

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

**Pseudocode**

If \( rA = 0 \) then

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

else

\[
\text{PC} \leftarrow \text{PC} + 4
\]

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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bge rA, rB</td>
<td>Branch if Greater or Equal</td>
</tr>
<tr>
<td>bged rA, rB</td>
<td>Branch if Greater or Equal with Delay</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)
bgei

**Branch Immediate if Greater or Equal**

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

<table>
<thead>
<tr>
<th>0 1 0 1 1 1</th>
<th>D 0 1 0 1</th>
<th>rA 16</th>
<th>IMM 31</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 greater or equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

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

**Pseudocode**

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

**Registers Altered**

- PC

**Latency**

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

**Note**

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

bgt

Branch if Greater Than

bgt    rA, rB         Branch if Greater Than
bgtd   rA, rB         Branch if Greater Than with Delay

Description

Branch if rA is greater than 0, to the instruction located in the offset value of rB. The target of the branch will be the instruction at address PC + rB.

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

Pseudocode

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

Registers Altered

• PC

Latency

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

Branch Immediate if Greater Than

bgti rA, IMM     Branch Immediate if Greater Than
bgtid rA, IMM    Branch Immediate if Greater Than with Delay

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

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

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

Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
ble 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>Opcode</th>
<th>Roadmap</th>
<th>rA</th>
<th>rB</th>
<th>D</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>001011</td>
<td>000001</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</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 \( rA, \text{IMM} \) Branch Immediate if Less or Equal
bleid \( rA, \text{IMM} \) Branch Immediate if Less or Equal with Delay

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

Description

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

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 \( rA \leq 0 \) then
\[
\text{PC} \leftarrow \text{PC} + \text{sext(IMM)}
\]
else
\[
\text{PC} \leftarrow \text{PC} + 4
\]
if \( D = 1 \) then
allow following instruction to complete execution

Registers Altered

- PC

Latency

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

Note

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

blt

Description

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

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

Pseudocode

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

Registers Altered

• PC

Latency

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

**Branch Immediate if Less Than**

- **blti**  
  
  rA, IMM  
  
  Branch Immediate if Less Than

- **bltid**  
  
  rA, IMM  
  
  Branch Immediate if Less Than with Delay

#### Description

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

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

#### Pseudocode

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

#### Registers Altered

- PC

#### Latency

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

#### Note

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

Branch if Not Equal

\[
\begin{align*}
\text{bne} & \quad rA, rB \quad \text{Branch if Not Equal} \\
\text{bned} & \quad rA, rB \quad \text{Branch if Not Equal with Delay}
\end{align*}
\]

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

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

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)
**bnei**  
Branch Immediate if Not Equal

**bnei**  
\[ \text{rA, IMM} \]  
Branch Immediate if Not Equal

**bneid**  
\[ \text{rA, IMM} \]  
Branch Immediate if Not Equal with Delay

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>D</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>rA</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

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

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

**Pseudocode**

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

**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)

**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

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>bld</td>
<td>rD, rB</td>
<td>Branch and Link with Delay</td>
</tr>
<tr>
<td>brald</td>
<td>rD, 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 bld 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, bld 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 bra1 are not available.
Chapter 4: MicroBlaze Instruction Set Architecture

### bri

**Unconditional Branch Immediate**

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

#### Description

Branch to the instruction located at address determined by IMM, sign-extended to 32 bits.

The mnemonics brlid and bralid will set the L bit. If the L bit is set, linking will be performed. The current value of PC will be stored in rD.

The mnemonics brai, braid and bralid will set the A bit. If the A bit is set, it means that the branch is to an absolute value and the target is the value in IMM, otherwise, it is a relative branch and the target will be PC + IMM.

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

#### Pseudocode

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

#### Registers Altered

- rD
- PC

#### Latency

2 cycles (if the D bit is set) or 3 cycles (if the D bit is not set)
Notes

The instructions brli and brali are not available.

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

**Break**

`brk` rD, rB

<table>
<thead>
<tr>
<th>1 0 0 1 1 0</th>
<th>rD</th>
<th>0 1 1 0 0</th>
<th>rB</th>
<th>0 0 0 0</th>
<th>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>
<td>31</td>
</tr>
</tbody>
</table>

**Description**

Branch and link to the instruction located at address value in rB. The current value of PC will be stored in rD. The BIP flag in the MSR will be set.

**Pseudocode**

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

**Registers Altered**

- rD
- PC
- MSR[BIP]

**Latency**

3 cycles
Instructions

brki

Break Immediate

`brki` rD, IMM

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</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>
<td></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.
Barrel Shift

bs

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>rD</th>
<th>rA</th>
<th>rB</th>
<th>S</th>
<th>T</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>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>
</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 bssl 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.
Instructions

bsi

Barrel Shift Immediate

bsrli rD, rA, IMM  Barrel Shift Right Logical Immediate
bsrai rD, rA, IMM  Barrel Shift Right Arithmetical Immediate
bslli rD, rA, IMM  Barrel Shift Left Logical Immediate

Description

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

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

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

Pseudocode

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.
**cmp**  

**Integer Compare**

\[
\begin{align*}
\text{cmp} & \quad rD, rA, rB \quad \text{compare rB with rA (signed)} \\
\text{cmpu} & \quad rD, rA, rB \quad \text{compare rB with rA (unsigned)}
\end{align*}
\]

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

**Description**

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

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

**Pseudocode**

\[
\begin{align*}
(rD) & \leftarrow (rB) + (\neg rA) + 1 \\
(rD)(\text{MSB}) & \leftarrow (rA) > (rB)
\end{align*}
\]

**Registers Altered**

- rD

**Latency**

1 cycle
Instructions

get
get from fsl interface

get rD, FSLx get data from FSL x (blocking)
nget rD, FSLx get data from FSL x (non-blocking)
cget rD, FSLx get control from FSL x (blocking)
ncget rD, FSLx get control from FSL x (non-blocking)

Description

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

The get instruction has four variants.

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

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

Pseudocode

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

if \( (n = 1) \) then

\[\begin{align*}
\text{MSR}[\text{Carry}] & \leftarrow \text{not (FSLx Exists bit)} \\
\text{if ((FSLx Control bit) == c)} & \text{then} \\
\text{MSR}[\text{FSL_Error}] & \leftarrow 0 \\
\text{else} & \\
\text{MSR}[\text{FSL_Error}] & \leftarrow 1
\end{align*}\]

Registers Altered

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

Latency

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

Note

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

## Integer Divide

**idiv**

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

**idivu**

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

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

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

## 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.
Instructions

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.
**lbu**

**Load Byte Unsigned**

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

| 1 | 1 | 0 | 0 | 0 | 0 | rD | rA | rB | 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**

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**

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

**Registers Altered**

- \( rD \)

**Latency**

2 cycles
lbui

Load Byte Unsigned Immediate

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

<table>
<thead>
<tr>
<th>1 1 1 0 0 0</th>
<th>(rD)</th>
<th>(rA)</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 6 11 16</td>
<td></td>
<td></td>
<td>31</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
\[
\text{Addr} \leftarrow (rA) + \text{sext}(\text{IMM})
\]
\[
(rD)[24:31] \leftarrow \text{Mem(Addr)}
\]
\[
(rD)[0:23] \leftarrow 0
\]

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 \text{imm} instruction. See the \text{imm} instruction for details on using 32-bit immediate values.
**lhu**

**Load Halfword Unsigned**

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>(rA) + (rB)</td>
<td>Addr[31] ← 0</td>
<td>(rD)[16:31] ← Mem(Addr)</td>
<td>(rD)[0:15] ← 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

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

**Pseudocode**

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

**Registers Altered**

- rD

**Latency**

2 cycles
### lhui

** Load Halfword Unsigned Immediate  

** lhui ** rD, rA, IMM  

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

**Description**

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

**Pseudocode**

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

**Registers Altered**

- rD

**Latency**

2 cycles

**Note**

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

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

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

Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles
**lwi**

**Load Word Immediate**

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

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

**Description**

_loads a word (32 bits) from the word aligned memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits. The data is placed in register rD._

**Pseudocode**

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

**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**

\[
mfs \quad rD, rS
\]

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

**Description**

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

**Pseudocode**

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

**Registers Altered**

- \( rD \)

**Latency**

1 cycle

**Note**

To refer to special purpose registers in assembly language, use `rpc` for PC, `rmsr` for MSR, `rear` for EAR, and `resr` for ESR. Source registers EAR and ESR are only valid in MicroBlaze v3.00a and higher.
msrclr

Read MSR and clear bits in MSR

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

<table>
<thead>
<tr>
<th>1 0 0 1 0 1</th>
<th>rD</th>
<th>0 0 0 0 1 0 0</th>
<th>Imm14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 6 11 16 17 18</td>
<td>31</td>
<td></td>
<td></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

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

Registers Altered

- rD
- MSR

Latency

1 cycle

Note

This instruction is only valid if C_USE_MSR_INSTR is set for MicroBlaze.
The immediate values has to be less than 2^{14}. Only bits 18 to 31 of the MSR can be cleared.
This instruction only exists in version 2.10.a and above.
**msrset**

Read MSR and set bits in MSR

```
msrset rD, Imm
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>rD</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th></th>
<th>Imm14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>18</td>
<td>31</td>
<td></td>
<td></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. The immediate values has to be less than 2^14. Only bits 18 to 31 of the MSR can be set. This instruction only exists in version 2.10.a and above.
mts

Move To Special Purpose Register

mts rS, rA

Description
Copies the contents of register rD into the MSR register.

Pseudocode
(rS) ← (rA)

Registers Altered
• rS

Latency
1 cycle

Notes
You cannot write to the PC using the MTS instruction.

When writing to MSR using MTS, the value written will take effect one clock cycle after executing the MTS instruction.

To refer to special purpose registers in assembly language, use rmsr for MSR. PC, ESR and EAR can not be written by the MTS instruction.
mul

Multiply

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

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

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

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

Registers Altered

- rD

Latency

3 cycles

Note

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

**Multiply Immediate**

`muli rD, rA, IMM`

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

**Description**

Multiplies the contents of registers rA and the value IMM, sign-extended to 32 bits; and puts the result in register rD. This is a 32-bit by 32-bit multiplication that will produce a 64-bit result. The least significant word of this value is placed in rD. The most significant word is discarded.

**Pseudocode**

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

**Registers Altered**

- rD

**Latency**

3 cycles

**Notes**

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

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

or rD, rA, rB

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

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

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.
**put**

**put to fsl interface**

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

**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**

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

**Registers Altered**

- MSR[Carry]

**Latency**

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

Arithmetic Reverse Subtract

**rsub**

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

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

### Registers Altered

- rD
- MSR[C]

### Latency

1 cycle

### Notes

In subtractions, Carry = (Borrow). When the Carry is set by a subtraction, it means that there is no Borrow, and when the Carry is cleared, it means that there is a Borrow.
**rsubi**

**Arithmetic Reverse Subtract Immediate**

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>format</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsubi</td>
<td>rD, rA, IMM</td>
<td>Subtract Immediate</td>
</tr>
<tr>
<td>rsubic</td>
<td>rD, rA, IMM</td>
<td>Subtract Immediate with Carry</td>
</tr>
<tr>
<td>rsubik</td>
<td>rD, rA, IMM</td>
<td>Subtract Immediate and Keep Carry</td>
</tr>
<tr>
<td>rsubikc</td>
<td>rD, rA, IMM</td>
<td>Subtract Immediate with Carry and Keep Carry</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 rsubit. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic rsubik. Both bits are set to a one for the mnemonic rsubit.

When an rsubi instruction has bit 3 set (rsubit, rsubit), the carry flag will keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (rsubi, rsubit), 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, rsubit), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (rsubi, rsubit), 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 = \(\overline{\text{Borrow}}\). When the Carry is set by a subtraction, it means that there is no Borrow, and when the Carry is cleared, it means that there is a Borrow.

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

Return from Break

rtbd rA, IMM

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

Description

Return from break will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits. It will also enable breaks after execution by clearing the BIP flag in the MSR.

This instruction always has a delay slot. The instruction following the RTBD is always executed before the branch target. That delay slot instruction has breaks disabled.

Pseudocode

PC ← (rA) + sext(IMM)
allow following instruction to complete execution
MSR[BIP] ← 0

Registers Altered

- PC
- MSR[BIP]

Latency

2 cycles
### rtid: Return from Interrupt

**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
**rted**

**Return from Exception**

**rtsd**

**rA, IMM**

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

**Description**

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**

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

**Registers Altered**

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

**Latency**

2 cycles

**Notes**

This instruction is only available in MicroBlaze v3.00a or higher.
rtsd

Return from Subroutine

rtsd rA, IMM

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

Description

Return from subroutine will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits.

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

Pseudocode

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

Registers Altered

- PC

Latency

2 cycles
sb

**Store Byte**

sb \( rD, rA, rB \)

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

**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**

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

**Registers Altered**

- None

**Latency**

2 cycles
sbi

**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**

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

**Registers Altered**
- None

**Latency**
2 cycles

**Note**
By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.
sext16  

**Sign Extend Halfword**

\[
\text{sext16} \quad rD, rA
\]

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

**Description**

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

**Pseudocode**

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

**Registers Altered**

- rD

**Latency**

1 cycle
sext8

**Sign Extend Byte**

**sext8**  
**rD, rA**

### Description

This instruction sign-extends a byte (8 bits) into a word (32 bits). Bit 24 in rA will be copied into bits 0-23 of rD. Bits 24-31 in rA will be copied into bits 24-31 of rD.

### Pseudocode

\[
\text{(rD)}[0:23] \leftarrow (\text{rA})[24] \\
\text{(rD)}[24:31] \leftarrow (\text{rA})[24:31]
\]

### Registers Altered

- **rD**

### Latency

1 cycle
sh  Store Halfword

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

<table>
<thead>
<tr>
<th>1 1 0 1 0 1</th>
<th>( rD )</th>
<th>( rA )</th>
<th>( rB )</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</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
\[ \text{Addr} \leftarrow (rA) + (rB) \]
\[ \text{Addr}[31] \leftarrow 0 \]
\[ \text{Mem(Addr)} \leftarrow (rD)[16:31] \]

Registers Altered
- None

Latency
2 cycles
**shi Store Halfword Immediate**

**shi** rD, rA, IMM

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

**Description**
Stores the contents of the least significant halfword of register rD, into the halfword aligned memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits.

**Pseudocode**

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

**Registers Altered**
- None

**Latency**
2 cycles

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

**Shift Right Arithmetic**

\[
sra \quad rD, rA
\]

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

**Description**

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

**Pseudocode**

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

**Registers Altered**

- rD
- MSR[C]

**Latency**

1 cycle
srl
Shift Right Logical

srl    rD, rA

|    1 0 0 1 0 0   | rD |    0 0 0 0 0 0 0 0 1 0 0 0 0 0 1    |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|    0 |    6 |    11 |    16 |    31 |

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

Pseudocode
(rD)[0] ← 0
(rD)[1:31] ← (rA)[0:30]
MSR[C] ← (rA)[31]

Registers Altered
• rD
• MSR[C]

Latency
1 cycle
SW
Store Word

sw \quad rD, rA, rB

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>01</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></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
Addr \leftarrow (rA) + (rB)
Addr[30:31] \leftarrow 00
Mem(Addr) \leftarrow (rD)[0:31]

Registers Altered
• None

Latency
2 cycles
swi

Store Word Immediate

\textbf{swi} \quad rD, rA, \text{IMM}

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

\textbf{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.

\textbf{Pseudocode}

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

\textbf{Register Altered}

- None

\textbf{Latency}

2 cycles

\textbf{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 \texttt{imm} instruction. See the \texttt{imm} instruction for details on using 32-bit immediate values.
wdc  Write to Data Cache

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

Description

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

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

Pseudocode

(DCache Tag) ← (rA)
(DCache Data) ← (rB)

Registers Altered

- None

Latency

1 cycle
**wic**

Write to Instruction Cache

\[
wic \quad rA, rB
\]

```
0  1  0  0  1  0  0  rA  rA  rB  0  0  0  0  1  1  0  1  0  0  0
0  6  11 16
```

**Description**

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

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

**Pseudocode**

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

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

**Registers Altered**

- None

**Latency**

1 cycle
**xor**

**Logical Exclusive OR**

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

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

**Description**

The contents of register \(r_A\) are XORed with the contents of register \(r_B\); the result is placed into register \(r_D\).

**Pseudocode**

\[(r_D) \leftarrow (r_A) \oplus (r_B)\]

**Registers Altered**

- \(r_D\)

**Latency**

1 cycle
**Logical Exclusive OR with Immediate**

### xori

**Syntax:**

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

### Description

The IMM field is extended to 32 bits by concatenating 16 0-bits on the left. The contents of register rA are XORed with the extended IMM field; the result is placed into register rD.

### Pseudocode

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

### 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.