MicroBlaze Processor Reference Guide

Embedded Development Kit EDK 6.3i

∑ XILINX ®

MicroBlaze Processor Reference Guide
UG081 (v4.0) August 24, 2004

The following table shows the revision history for this document.

	Version	Revision
10/01/02	1.0	Xilinx EDK 3.1 release
03/11/03	2.0	Xilinx EDK 3.2 release
09/24/03	3.0	Xilinx EDK 6.1 release
02/20/04	3.1	Xilinx EDK 6.2 release
08/24/04	4.0	Xilinx EDK 6.3 release

Table of Contents

Preface: About This Guide

Chapter 1: MicroBlaze Architecture

Chapter 2: MicroBlaze Signal Interface Description

Chapter 3: MicroBlaze Application Binary Interface

Chapter 4: MicroBlaze Instruction Set Architecture

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 Endieness
- 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.

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx solution records http://support.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes
Data Book	Pages from *The Programmable Logic Data Book*, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://support.xilinx.com/xlnx/xweb/xil_publications_index.jsp

Resource	Description/URL
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://support.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment http://www.support.xilinx.com/xlnx/xil_tt_home.jsp
GNU Manuals	The entire set of GNU manuals http://www.gnu.org/manual

Conventions

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

Typographical

The following typographical conventions are used in this document:

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

Convention	Meaning or Use	Example

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		`.`
		`.`
		`.`

Horizontal ellipsis . . .	Repetitive material that has been omitted	**`allow block`** `block_name` `loc1 loc2 ... locn;`

Online Document

The following conventions are used in this document:

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

Chapter 1

MicroBlaze Architecture

Overview

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

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
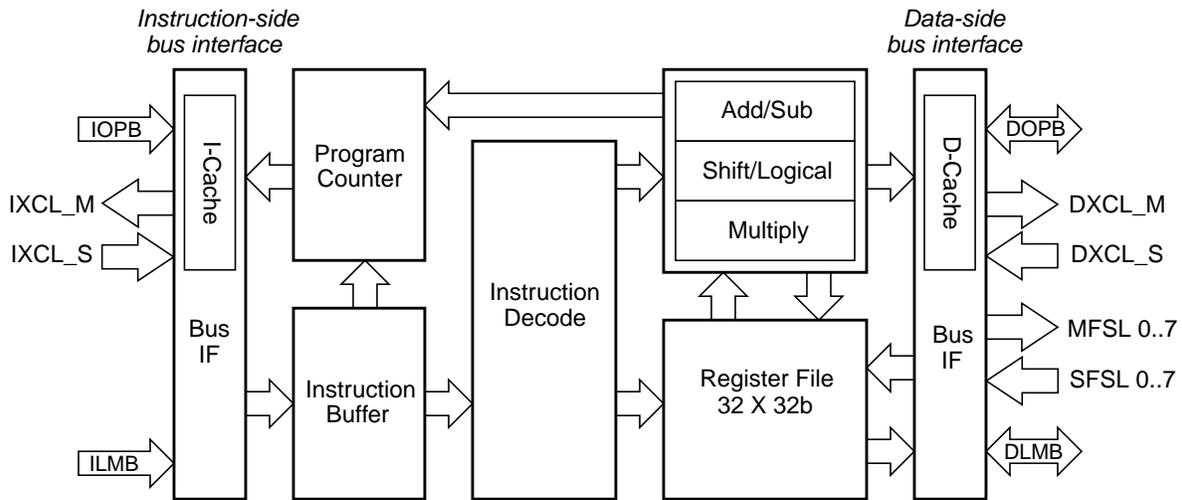
Figure 1-1: **MicroBlaze Core Block Diagram**

Data Types and Endianness

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

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

Byte address	n	n+1	n+2	n+3
Byte label	0	1	2	3
Byte significance	MSByte			LSByte
Bit label	0			31
Bit significance	MSBit			LSBit

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

Byte address	n	n+1
Byte label	0	1
Byte significance	MSByte	LSByte
Bit label	0	15
Bit significance	MSBit	LSBit

Table 1-3: **Byte Data Type**

Byte address	n
Byte label	0
Byte significance	MSByte
Bit label	0 7
Bit significance	MSBit LSBit

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 1-4: **Instruction Set Nomenclature**

Symbol	Description
Ra	R0 - R31, General Purpose Register, source operand a
Rb	R0 - R31, General Purpose Register, source operand b
Rd	R0 - R31, General Purpose Register, destination operand
Imm	16 bit immediate value
Imm*x*	*x* bit immediate value
FSL*x*	3 bit Fast Simplex Link (FSL) port designator where *x* is the port number
C	Carry flag, MSR[29]
Sa	Special Purpose Register, source operand
Sd	Special Purpose Register, destination operand
s(*x*)	Sign extend argument *x* to 32-bit value
*Addr	Memory contents at location Addr (data-size aligned)
&	Concatenate. E.g. "0000100 & Imm7" is the concatenation of the fixed field "0000100" and a 7 bit immediate value.

Table 1-5: **MicroBlaze Instruction Set Summary**

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15		16-31	
ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	00000000000	$Rd := Rb + Ra$
RSUB Rd,Ra,Rb	000001	Rd	Ra	Rb	00000000000	$Rd := Rb + \overline{Ra} + 1$
ADDC Rd,Ra,Rb	000010	Rd	Ra	Rb	00000000000	$Rd := Rb + Ra + C$
RSUBC Rd,Ra,Rb	000011	Rd	Ra	Rb	00000000000	$Rd := Rb + \overline{Ra} + C$
ADDK Rd,Ra,Rb	000100	Rd	Ra	Rb	00000000000	$Rd := Rb + Ra$
RSUBK Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000000	$Rd := Rb + \overline{Ra} + 1$
ADDKC Rd,Ra,Rb	000110	Rd	Ra	Rb	00000000000	$Rd := Rb + Ra + C$
RSUBKC Rd,Ra,Rb	000111	Rd	Ra	Rb	00000000000	$Rd := Rb + \overline{Ra} + C$
CMP Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000001	$Rd := Rb + \overline{Ra} + 1(signed)$
CMPU Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000011	$Rd := Rb + \overline{Ra} + 1(unsigned)$
ADDI Rd,Ra,Imm	001000	Rd	Ra		Imm	$Rd := s(Imm) + Ra$
RSUBI Rd,Ra,Imm	001001	Rd	Ra		Imm	$Rd := s(Imm) + \overline{Ra} + 1$
ADDIC Rd,Ra,Imm	001010	Rd	Ra		Imm	$Rd := s(Imm) + Ra + C$
RSUBIC Rd,Ra,Imm	001011	Rd	Ra		Imm	$Rd := s(Imm) + \overline{Ra} + C$
ADDIK Rd,Ra,Imm	001100	Rd	Ra		Imm	$Rd := s(Imm) + Ra$
RSUBIK Rd,Ra,Imm	001101	Rd	Ra		Imm	$Rd := s(Imm) + \overline{Ra} + 1$
ADDIKC Rd,Ra,Imm	001110	Rd	Ra		Imm	$Rd := s(Imm) + Ra + C$
RSUBIKC Rd,Ra,Imm	001111	Rd	Ra		Imm	$Rd := s(Imm) + \overline{Ra} + C$
MUL Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000000	$Rd := Ra * Rb$
BSRL Rd,Ra,Rb	010001	Rd	Ra	Rb	00000000000	$Rd := Ra >> Rb$
BSRA Rd,Ra,Rb	010001	Rd	Ra	Rb	01000000000	$Rd := Ra[0], (Ra >> Rb)$
BSLL Rd,Ra,Rb	010001	Rd	Ra	Rb	10000000000	$Rd := Ra << Rb$
MULI Rd,Ra,Imm	011000	Rd	Ra		Imm	$Rd := Ra * s(Imm)$
BSRLI Rd,Ra,Imm	011001	Rd	Ra		00000000000 & Imm5	$Rd := Ra >> Imm5$
BSRAI Rd,Ra,Imm	011001	Rd	Ra		00000010000 & Imm5	$Rd := Ra[0], (Ra >> Imm5)$
BSLLI Rd,Ra,Imm	011001	Rd	Ra		00000100000 & Imm5	$Rd := Ra << Imm5$
IDIV Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000000	$Rd := Rb/Ra$, signed
IDIVU Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000001	$Rd := Rb/Ra$, unsigned
GET Rd,FSLx	011011	Rd	00000		0000000000000 & FSLx	$Rd := FSLx$ (blocking data read) $MSR[FSL] := FSLx\_S\_Control$

Table 1-5:   **MicroBlaze Instruction Set Summary** *(Continued)*

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	**0-5**	**6-10**	**11-15**	**16-31**		
PUT Ra,FSLx	011011	00000	Ra	1000000000000 & FSLx		FSLx := Ra (blocking data write)
NGET Rd,FSLx	011011	Rd	00000	0100000000000 & FSLx		Rd := FSLx (non-blocking data read) MSR[FSL] := FSLx_S_Control MSR[C] := not FSLx_S_Exists
NPUT Ra,FSLx	011011	00000	Ra	1100000000000 & FSLx		FSLx := Ra (non-blocking data write) MSR[C] := FSLx_M_Full
CGET Rd,FSLx	011011	Rd	00000	0010000000000 & FSLx		Rd := FSLx (blocking control read) MSR[FSL] := not FSLx_S_Control
CPUT Ra,FSLx	011011	00000	Ra	1010000000000 & FSLx		FSLx := Ra (blocking control write)
NCGET Rd,FSLx	011011	Rd	00000	0110000000000 & FSLx		Rd := FSLx (non-blocking control read) MSR[FSL] := not FSLx_S_Control MSR[C] := not FSLx_S_Exists
NCPUT Ra,FSLx	011011	00000	Ra	1110000000000 & FSLx		FSLx := Ra (non-blocking control write) MSR[C] := FSLx_M_Full
OR Rd,Ra,Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
AND Rd,Ra,Rb	100001	Rd	Ra	Rb	00000000000	Rd := Ra and Rb
XOR Rd,Ra,Rb	100010	Rd	Ra	Rb	00000000000	Rd := Ra xor Rb
ANDN Rd,Ra,Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{Rb}$
SRA Rd,Ra	100100	Rd	Ra	000000000000001		Rd := Ra[0], (Ra >> 1); C := Ra[31]
SRC Rd,Ra	100100	Rd	Ra	000000000100001		Rd := C, (Ra >> 1); C := Ra[31]
SRL Rd,Ra	100100	Rd	Ra	000000001000001		Rd := 0, (Ra >> 1); C := Ra[31]
SEXT8 Rd,Ra	100100	Rd	Ra	000000001100000		Rd[0:23] := Ra[24]; Rd[24:31] := Ra[24:31]
SEXT16 Rd,Ra	100100	Rd	Ra	000000001100001		Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]
WIC Ra,Rb	100100	Ra	Ra	Rb	01101000	ICache_Tag := Ra, ICache_Data := Rb
WDC Ra,Rb	100100	Ra	Ra	Rb	01100100	DCache_Tag := Ra, DCache_Data := Rb
MTS Sd,Ra	100101	00000	Ra	1100000000000 & Sd		Sd := Ra , where Sd=001 is MSR
MFS Rd,Sa	100101	Rd	00000	1000000000000 & Sa		Rd := Sa , where Sa=000 is PC, 001 is MSR, 011 is EAR, and 101 is ESR
MSRCLR Rd,Imm	100101	Rd	00001	00 & Imm14		Rd := MSR; MSR := MSR ^ Imm14
MSRSET Rd,Imm	100101	Rd	00000	00 & Imm14		Rd := MSR; MSR := MSR ^ Imm14
BR Rb	100110	00000	00000	Rb	00000000000	PC := PC + Rb
BRD Rb	100110	00000	10000	Rb	00000000000	PC := PC + Rb

Table 1-5: **MicroBlaze Instruction Set Summary** *(Continued)*

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	**0-5**	**6-10**	**11-15**	**16-31**		
BRLD Rd,Rb	100110	Rd	10100	Rb	00000000000	PC := PC + Rb; Rd := PC
BRA Rb	100110	00000	01000	Rb	00000000000	PC := Rb
BRAD Rb	100110	00000	11000	Rb	00000000000	PC := Rb
BRALD Rd,Rb	100110	Rd	11100	Rb	00000000000	PC := Rb; Rd := PC
BRK Rd,Rb	100110	Rd	01100	Rb	00000000000	PC := Rb; Rd := PC; MSR[BIP] := 1
BEQ Ra,Rb	100111	00000	Ra	Rb	00000000000	if Ra = 0: PC := PC + Rb
BNE Ra,Rb	100111	00001	Ra	Rb	00000000000	if Ra /= 0: PC := PC + Rb
BLT Ra,Rb	100111	00010	Ra	Rb	00000000000	if Ra < 0: PC := PC + Rb
BLE Ra,Rb	100111	00011	Ra	Rb	00000000000	if Ra <= 0: PC := PC + Rb
BGT Ra,Rb	100111	00100	Ra	Rb	00000000000	if Ra > 0: PC := PC + Rb
BGE Ra,Rb	100111	00101	Ra	Rb	00000000000	if Ra >= 0: PC := PC + Rb
BEQD Ra,Rb	100111	10000	Ra	Rb	00000000000	if Ra = 0: PC := PC + Rb
BNED Ra,Rb	100111	10001	Ra	Rb	00000000000	if Ra /= 0: PC := PC + Rb
BLTD Ra,Rb	100111	10010	Ra	Rb	00000000000	if Ra < 0: PC := PC + Rb
BLED Ra,Rb	100111	10011	Ra	Rb	00000000000	if Ra <= 0: PC := PC + Rb
BGTD Ra,Rb	100111	10100	Ra	Rb	00000000000	if Ra > 0: PC := PC + Rb
BGED Ra,Rb	100111	10101	Ra	Rb	00000000000	if Ra >= 0: PC := PC + Rb
ORI Rd,Ra,Imm	101000	Rd	Ra		Imm	Rd := Ra or s(Imm)
ANDI Rd,Ra,Imm	101001	Rd	Ra		Imm	Rd := Ra and s(Imm)
XORI Rd,Ra,Imm	101010	Rd	Ra		Imm	Rd := Ra xor s(Imm)
ANDNI Rd,Ra,Imm	101011	Rd	Ra		Imm	Rd := Ra and $\overline{s(\text{Imm})}$
IMM Imm	101100	00000	00000		Imm	Imm[0:15] := Imm
RTSD Ra,Imm	101101	10000	Ra		Imm	PC := Ra + s(Imm)
RTID Ra,Imm	101101	10001	Ra		Imm	PC := Ra + s(Imm); MSR[IE] := 1
RTED Ra,Imm	101101	10010	Ra		Imm	PC := Ra + s(Imm); MSR[EE] := 1, MSR[EIP]:=0
RTBD Ra,Imm	101101	10010	Ra		Imm	PC := Ra + s(Imm); MSR[BIP] := 0
BRI Imm	101110	00000	00000		Imm	PC := PC + s(Imm)
BRID Imm	101110	00000	10000		Imm	PC := PC + s(Imm)
BRLID Rd,Imm	101110	Rd	10100		Imm	PC := PC + s(Imm); Rd := PC
BRAI Imm	101110	00000	01000		Imm	PC := s(Imm)
BRAID Imm	101110	00000	11000		Imm	PC := s(Imm)
BRALID Rd,Imm	101110	Rd	11100		Imm	PC := s(Imm); Rd := PC

Table 1-5: **MicroBlaze Instruction Set Summary** *(Continued)*

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15		16-31	
BRKI Rd,Imm	101110	Rd	01100		Imm	PC := s(Imm); Rd := PC; MSR[BIP] := 1
BEQI Ra,Imm	101111	00000	Ra		Imm	if Ra = 0: PC := PC + s(Imm)
BNEI Ra,Imm	101111	00001	Ra		Imm	if Ra /= 0: PC := PC + s(Imm)
BLTI Ra,Imm	101111	00010	Ra		Imm	if Ra < 0: PC := PC + s(Imm)
BLEI Ra,Imm	101111	00011	Ra		Imm	if Ra <= 0: PC := PC + s(Imm)
BGTI Ra,Imm	101111	00100	Ra		Imm	if Ra > 0: PC := PC + s(Imm)
BGEI Ra,Imm	101111	00101	Ra		Imm	if Ra >= 0: PC := PC + s(Imm)
BEQID Ra,Imm	101111	10000	Ra		Imm	if Ra = 0: PC := PC + s(Imm)
BNEID Ra,Imm	101111	10001	Ra		Imm	if Ra /= 0: PC := PC + s(Imm)
BLTID Ra,Imm	101111	10010	Ra		Imm	if Ra < 0: PC := PC + s(Imm)
BLEID Ra,Imm	101111	10011	Ra		Imm	if Ra <= 0: PC := PC + s(Imm)
BGTID Ra,Imm	101111	10100	Ra		Imm	if Ra > 0: PC := PC + s(Imm)
BGEID Ra,Imm	101111	10101	Ra		Imm	if Ra >= 0: PC := PC + s(Imm)
LBU Rd,Ra,Rb	110000	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:23] := 0, Rd[24:31] := *Addr
LHU Rd,Ra,Rb	110001	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:15] := 0, Rd[16:31] := *Addr
LW Rd,Ra,Rb	110010	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd := *Addr
SB Rd,Ra,Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[24:31]
SH Rd,Ra,Rb	110101	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[16:31]
SW Rd,Ra,Rb	110110	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd
LBUI Rd,Ra,Imm	111000	Rd	Ra		Imm	Addr := Ra + s(Imm); Rd[0:23] := 0, Rd[24:31] := *Addr
LHUI Rd,Ra,Imm	111001	Rd	Ra		Imm	Addr := Ra + s(Imm); Rd[0:15] := 0, Rd[16:31] := *Addr
LWI Rd,Ra,Imm	111010	Rd	Ra		Imm	Addr := Ra + s(Imm); Rd := *Addr

Table 1-5: **MicroBlaze Instruction Set Summary** *(Continued)*

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
SBI Rd,Ra,Imm	111100	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[24:31]
SHI Rd,Ra,Imm	111101	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[16:31]
SWI Rd,Ra,Imm	111110	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd

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.

0	31

↑
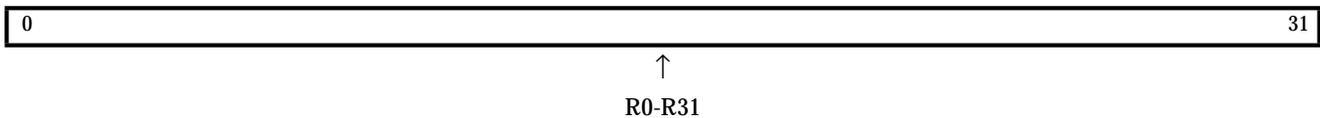
R0-R31

Figure 1-2: **R0-R31**

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

Bits	Name	Description	Reset Value
0:31	R0	R0 is defined to always have the value of zero. Anything written to R0 is discarded.	0x00000000
0:31	R1 through R31	R1 through R31 are 32-bit general purpose registers	0x00000000
0:31	R14	32-bit used to store return addresses for interrupts	0x00000000
0:31	R15	32-bit general purpose register	0x00000000

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

Bits	Name	Description	Reset Value
0:31	R16	32-bit used to store return addresses for breaks	0x00000000
0:31	R17	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	0x00000000
0:31	R18 through R31	R18 through R31 are 32-bit general purpose registers.	0x00000000

Special Purpose Registers

Program Counter (PC)

The Program Counter is the 32-bit address of the execution instruction. It can be read with an MFS instruction. It cannot be written to using an MTS instruction. When used with the MFS instruction the PC register is specified by setting Sa = 00000, or Sa = rpc.

0	31

↑
PC
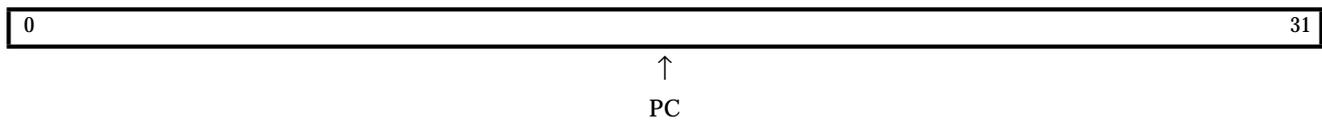
Figure 1-3: **PC**

Table 1-7: **Program Counter (PC)**

Bits	Name	Description	Reset Value
0:31	PC	**Program Counter** Address of executing instruction, i.e. "mfs r2 rpc" will store the address of the mfs instruction itself in R2	0x00000000

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 Sx = 00001, or Sx = rmsr.

0		22	23	24	25	26	27	28	29	30	31

CC   RESERVED   EIP EE DCE DZ ICE FSL BIP C IE BE
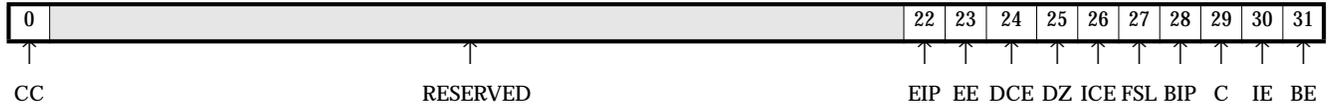
Figure 1-4: **MSR**

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

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

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

Bits	Name	Description	Reset Value
28	BIP	**Break in Progress** 0 No Break in Progress 1 Break in Progress Source of break can be software break instruction or hardware break from Ext_Brk or Ext_NM_Brk pin. Read-only	0
29	C	**Arithmetic Carry** 0 No Carry (Borrow) 1 Carry (No Borrow) Read-only	0
30	IE	**Interrupt Enable** 0 Interrupts disabled 1 Interrupts enabled Read/Write	0
31	BE	**Buslock Enable**[2] 0 Buslock disabled on data-side OPB 1 Buslock enabled on data-side OPB Buslock Enable does not affect operation of IXCL, DXCL, ILMB, DLMB, or IOPB. Read/Write	0

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.
2. For a details on the OPB protocol, please refer to the IBM CoreConnect specification: *64-Bit On-Chip Peripheral Bus, Architectural Specifications, Version 2.0.*

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

0	31

↑
EAR
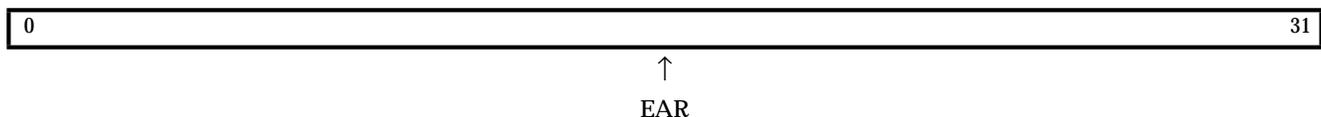
Figure 1-5: **EAR**

Table 1-9: **Machine Status Register (EAR)**

Bits	Name	Description	Reset Value
0:31	EAR	**Exception Address Register**	0x00000000

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
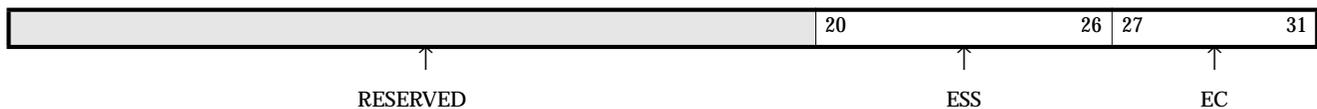
Figure 1-6: **ESR**

Table 1-10: **Exception Status Register (ESR)**

Bits	Name	Description	Reset Value
0:19	Reserved		
20:26	ESS	**Exception Specific Status** For details refer to Table 1-11. Read-only	
27:31	EC	**Exception Cause** 00001 Unaligned data access exception 00010 Illegal op-code exception 00011 Instruction bus error exception 00100 Data bus error exception 00101 Divide by zero exception Read-only	0

Table 1-11: **Exception Specific Status (ESS)**

Exception Cause	Bits	Name	Description	Reset Value
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

Pipeline

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.

	cycle 1	cycle 2	cycle 3	cycle4	cycle5	cycle6	cycle7
instruction 1	Fetch	Decode	Execute				
instruction 2		Fetch	Decode	Execute	Execute	Execute	
instruction 3			Fetch	Decode	Stall	Stall	Execute

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 bounders), 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".

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 1-12: **Vectors and Return Address Register File Location**

Event	Vector Address	Register File Return Address
Reset	0x00000000 - 0x00000004	-
User Vector (Exception)	0x00000008 - 0x0000000C	-
Interrupt	0x00000010 - 0x00000014	R14
Break: Non-maskable hardware	0x00000018 - 0x0000001C	R16
Break: Hardware Break		
Break: Software		
Hardware Exception	0x00000020 - 0x00000024	R17

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

```
PC ← 0x00000000
```

1. Reset input controlled by the XMD debugger via MDM

```
MSR ← 0
EAR ← 0
ESR ← 0
```

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**

Scenario	LMB Memory Vector	OPB Memory Vector
Normally	4 cycles	6 cycles
Worst case without hardware divider	6 cycles	8 cycles
Worst case with hardware divider[1]	37 cycles	39 cycles

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

Equivalent Pseudocode

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

User Vector (Exception)

The user exception vector is located at address 0x8. A user exception is 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

```
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

```
r17 ← PC
PC ← 0x00000020
MSR[EE] ← 0
MSR[EIP] ← 1
```

Breaks

There are two kinds of breaks:

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

Software Breaks

To perform a software break, use the `brk` and `brki` instructions. Refer to 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 `Ext_BRK` and `Ext_NM_BRK` input ports). On a break the instruction in the execution stage will complete, while the instruction in the decode stage is replaced by a branch to the break vector (address 0x18). The break return address (the PC associated with the instruction in the decode stage at the time of the break) is automatically loaded into general purpose register R16.MicroBlaze also sets the Break In Progress (BIP) flag in the Machine Status Register (MSR).

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

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

```
r16 ← PC
PC ← 0x00000018
MSR[BIP] ← 1
```

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 in split into two segments - a cacheable segment and a non-cacheable segment. The cacheable segment is determined by two parameters, C_ICACHE_BASEADDR and C_ICACHE_HIGHADDR. All addresses within this range correspond to the cacheable address space segment. All other addresses are non-cacheable.

Instruction Address Bits

0   30 31

| | Tag Address | Cache Word Address | - | - |

Figure 1-7: **OPB Instruction Cache Organization
(CacheLink is similar but uses 4 word cache lines)**

Cacheable instruction addresses are further split into two segments - a cache word address segment and a tag address segment. The size of the two segments 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 <u>byte</u> address range is 12 bits (i.e. a 10 bit cache <u>word</u> 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.
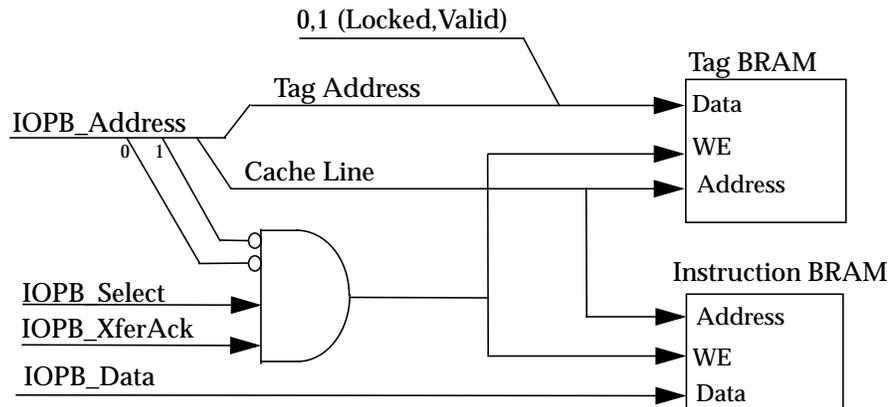
Figure 1-8: **Cache Operation**

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

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.
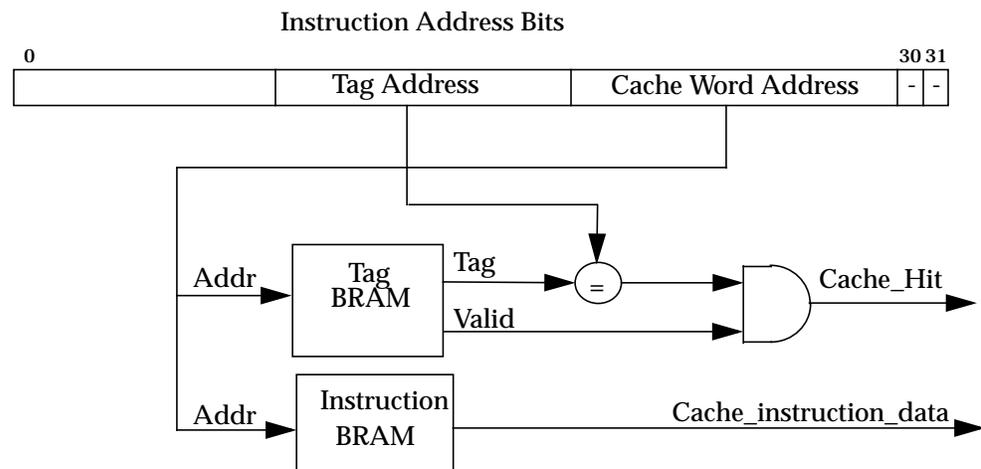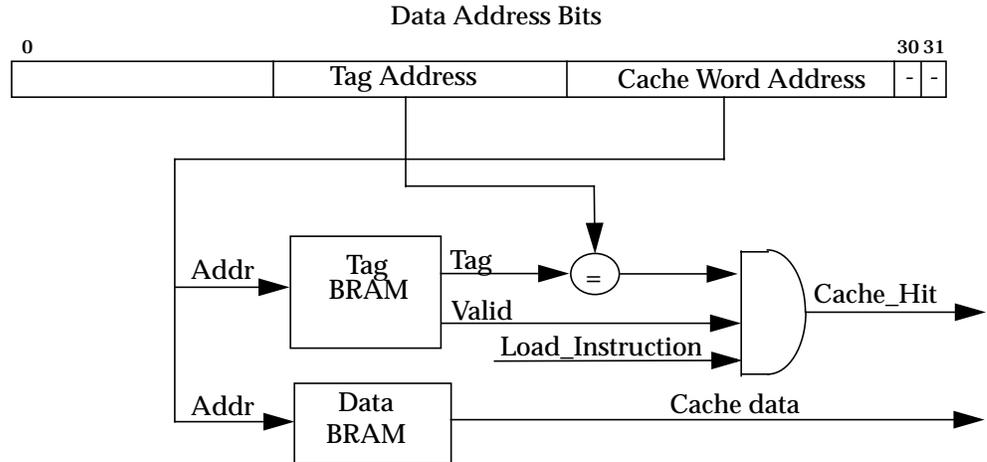
Data Address Bits

Figure 1-9: **OPB Data Cache Organization**
(CacheLink is similar but uses 4 word cache lines)

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 C_DCACHE_BASEADDR= 0x00400000, C_DCACHE_HIGHADDR=0x00403fff, and C_DCACHE_BYTE_SIZE=2048; the cacheable byte address range is 14 bits, and the cache <u>byte</u> address range is 11 bits (i.e. a 9 bit cache <u>word</u> 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.
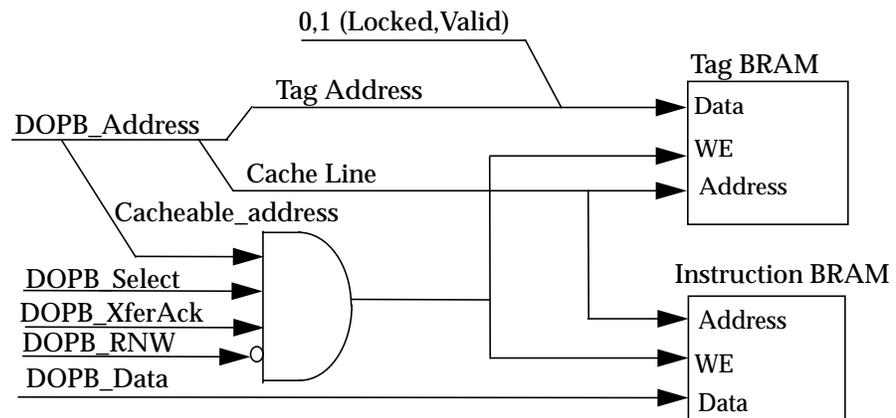
Figure 1-10: **Data Cache Operation**

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:

```
// Configure fx
cput FSLx, Rc
// Store operands
put FSLx, Ra // op 1
put FSLx, Rb // op 2
// Load result
get FSLx, Rt
```
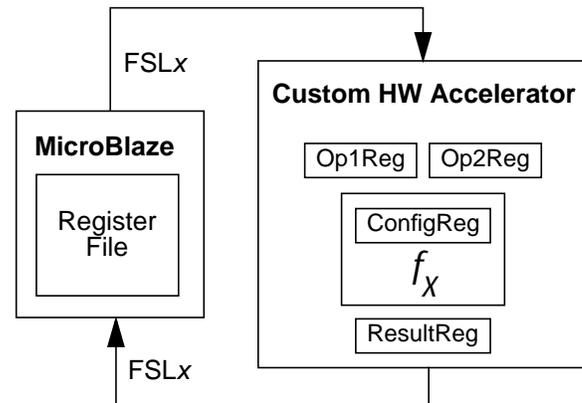
Figure 1-11: **FSL used with HW accelerated function** f_x

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.

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
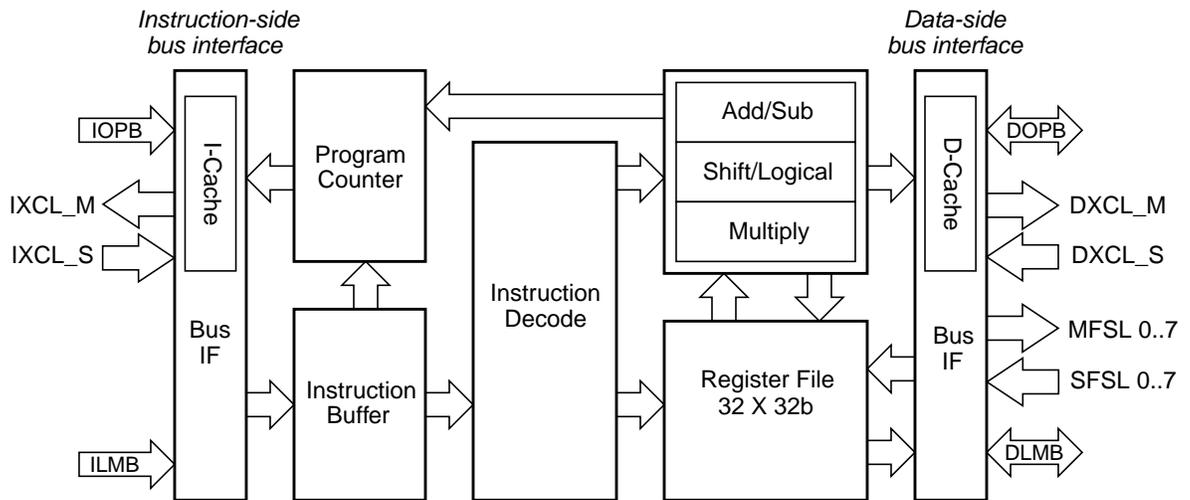
Figure 2-1: **MicroBlaze Core Block Diagram**

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

Signal	Interface	I/O	Description
DM_ABus[0:31]	DOPB	O	Data interface OPB address bus
DM_BE[0:3]	DOPB	O	Data interface OPB byte enables
DM_busLock	DOPB	O	Data interface OPB buslock
DM_DBus[0:31]	DOPB	O	Data interface OPB write data bus
DM_request	DOPB	O	Data interface OPB bus request
DM_RNW	DOPB	O	Data interface OPB read, not write
DM_select	DOPB	O	Data interface OPB select
DM_seqAddr	DOPB	O	Data interface OPB sequential address
DOPB_DBus[0:31]	DOPB	I	Data interface OPB read data bus
DOPB_errAck	DOPB	I	Data interface OPB error acknowledge
DOPB_MGrant	DOPB	I	Data interface OPB bus grant
DOPB_retry	DOPB	I	Data interface OPB bus cycle retry
DOPB_timeout	DOPB	I	Data interface OPB timeout error
DOPB_xferAck	DOPB	I	Data interface OPB transfer acknowledge
IM_ABus[0:31]	IOPB	O	Instruction interface OPB address bus
IM_BE[0:3]	IOPB	O	Instruction interface OPB byte enables
IM_busLock	IOPB	O	Instruction interface OPB buslock
IM_DBus[0:31]	IOPB	O	Instruction interface OPB write data bus (always 0x00000000)
IM_request	IOPB	O	Instruction interface OPB bus request

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

Signal	Interface	I/O	Description
IM_RNW	IOPB	O	Instruction interface OPB read, not write (tied to '0')
IM_select	IOPB	O	Instruction interface OPB select
IM_seqAddr	IOPB	O	Instruction interface OPB sequential address
IOPB_DBus[0:31]	IOPB	I	Instruction interface OPB read data bus
IOPB_errAck	IOPB	I	Instruction interface OPB error acknowledge
IOPB_MGrant	IOPB	I	Instruction interface OPB bus grant
IOPB_retry	IOPB	I	Instruction interface OPB bus cycle retry
IOPB_timeout	IOPB	I	Instruction interface OPB timeout error
IOPB_xferAck	IOPB	I	Instruction interface OPB transfer acknowledge
Data_Addr[0:31]	DLMB	O	Data interface LB address bus
Byte_Enable[0:3]	DLMB	O	Data interface LB byte enables
Data_Write[0:31]	DLMB	O	Data interface LB write data bus
D_AS	DLMB	O	Data interface LB address strobe
Read_Strobe	DLMB	O	Data interface LB read strobe
Write_Strobe	DLMB	O	Data interface LB write strobe
Data_Read[0:31]	DLMB	I	Data interface LB read data bus
DReady	DLMB	I	Data interface LB data ready
Instr_Addr[0:31]	ILMB	O	Instruction interface LB address bus
I_AS	ILMB	O	Instruction interface LB address strobe
IFetch	ILMB	O	Instruction interface LB instruction fetch
Instr[0:31]	ILMB	I	Instruction interface LB read data bus
IReady	ILMB	I	Instruction interface LB data ready
FSL0_M .. FSL7_M	MFSL	O	Master interface to Output FSL channels
FSL0_S .. FSL7_S	SFSL	I	Slave interface to Input FSL channels
ICache_FSL_in...	IXCL_S	IO	Instruction side CacheLink FSL slave interface
ICache_FSL_out...	IXCL_M	IO	Instruction side CacheLink FSL master interface
DCache_FSL_in...	DXCL_S	IO	Data side CacheLink FSL slave interface
DCache_FSL_out...	DXCL_M	IO	Data side CacheLink FSL master interface
Interrupt	Core	I	Interrupt

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

Signal	Interface	I/O	Description
Reset	Core	I	Core reset
Clk	Core	I	Clock
Debug_Rst	Core	I	Reset signal from OPB JTAG UART
Ext_BRK	Core	I	Break signal from OPB JTAG UART
Ext_NM_BRK	Core	I	Non-maskable break signal from OPB JTAG UART
Dbg_...	Core	IO	Debug signals from OPB MDM
Valid_Instr	Core	O	Trace signals for real time HW analysis
PC_Ex	Core	O	Trace signals for real time HW analysis
Reg_Write	Core	O	Trace signals for real time HW analysis
Reg_Addr	Core	O	Trace signals for real time HW analysis
MSR_Reg	Core	O	Trace signals for real time HW analysis
New_Reg_Value	Core	O	Trace signals for real time HW analysis
Pipe_Running	Core	O	Trace signals for real time HW analysis
Interrup_Taken	Core	O	Trace signals for real time HW analysis
Jump_Taken	Core	O	Trace signals for real time HW analysis
Prefetch_Addr	Core	O	Trace signals for real time HW analysis
MB_Halted	Core	O	Trace signals for real time HW analysis
Trace_...	Core	O	Trace signals for real time HW analysis

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.
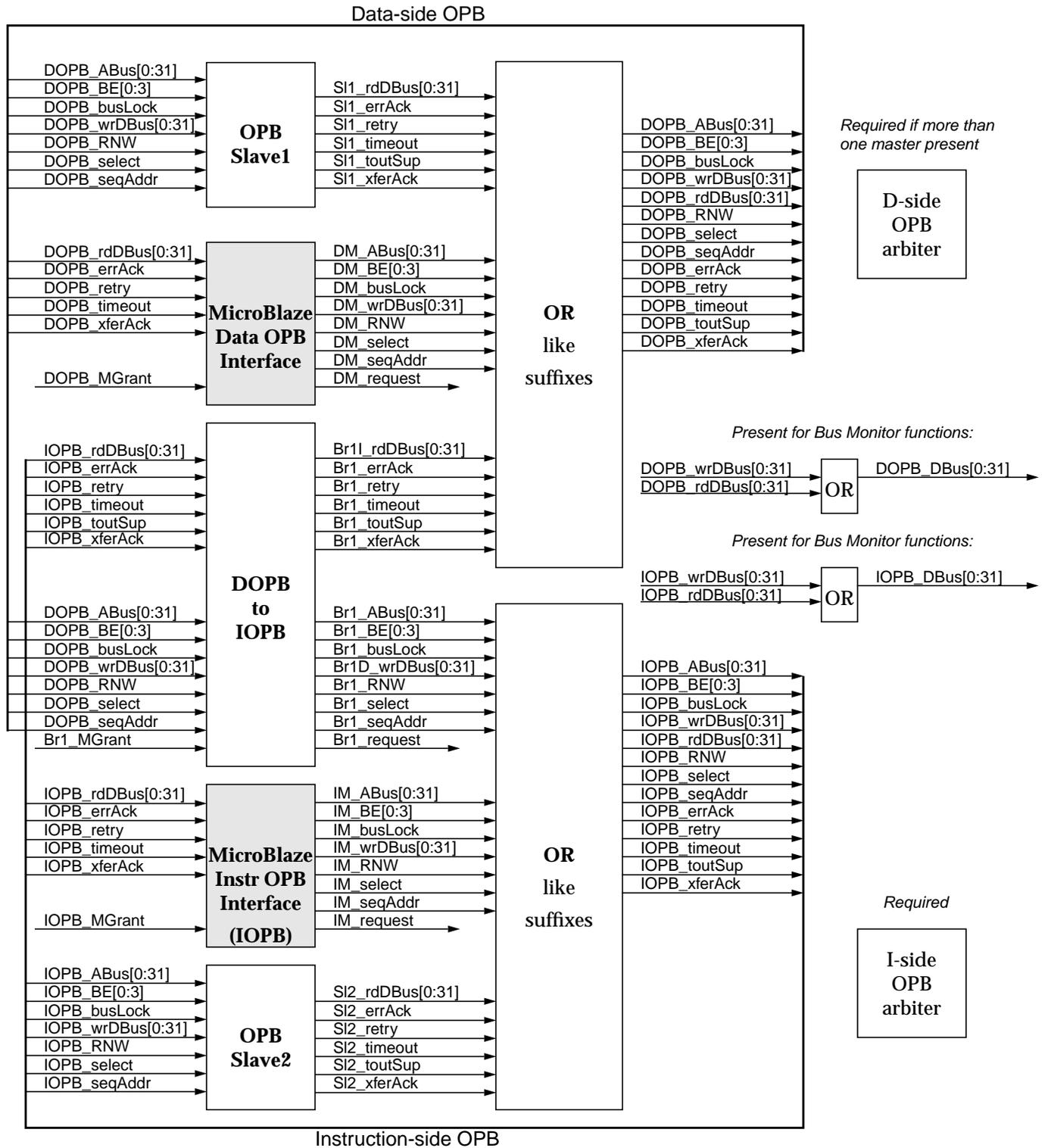
Figure 2-2: **OPB Interconnection (breaking up read and write buses)**

Data-side OPB

OPB
Slave1

DOPB_ABus[0:31]
DOPB_BE[0:3]
DOPB_busLock
DOPB_wrDBus[0:31]
DOPB_RNW
DOPB_select
DOPB_seqAddr

Sl1_rdDBus[0:31]
Sl1_errAck
Sl1_retry
Sl1_timeout
Sl1_toutSup
Sl1_xferAck

**MicroBlaze
Data OPB
Interface**

DOPB_rdDBus[0:31]
DOPB_errAck
DOPB_retry
DOPB_timeout
DOPB_xferAck

DOPB_MGrant

DM_ABus[0:31]
DM_BE[0:3]
DM_busLock
DM_wrDBus[0:31]
DM_RNW
DM_select
DM_seqAddr
DM_request

OR

like

suffixes

DOPB_ABus[0:31]
DOPB_BE[0:3]
DOPB_busLock
DOPB_wrDBus[0:31]
DOPB_rdDBus[0:31]
DOPB_RNW
DOPB_select
DOPB_seqAddr
DOPB_errAck
DOPB_retry
DOPB_timeout
DOPB_toutSup
DOPB_xferAck

*Required if more than
one master present*

D-side
OPB
arbiter

OPB
Slave2
(multi-

DOPB_ABus[0:31]
DOPB_BE[0:3]
DOPB_busLock
DOPB_wrDBus[0:31]
DOPB_RNW
DOPB_select
DOPB_seqAddr

Sl2_rdDBus[0:31]
Sl2_errAck
Sl2_retry
Sl2_timeout
Sl2_toutSup
Sl2_xferAck

Present for Bus Monitor functions:

DOPB_wrDBus[0:31]
DOPB_rdDBus[0:31]

OR

DOPB_DBus[0:31]

Present for Bus Monitor functions:

IOPB_wrDBus[0:31]
IOPB_rdDBus[0:31]

OR

IOPB_DBus[0:31]

IOPB_ABus[0:31]
IOPB_BE[0:3]
IOPB_busLock
IOPB_wrDBus[0:31]
IOPB_RNW
IOPB_select
IOPB_seqAddr

Sl2_rdDBus[0:31]
Sl2_errAck
Sl2_retry
Sl2_timeout
Sl2_toutSup
Sl2_xferAck

IOPB_ABus[0:31]
IOPB_BE[0:3]
IOPB_busLock
IOPB_wrDBus[0:31]
IOPB_rdDBus[0:31]
IOPB_RNW
IOPB_select
IOPB_seqAddr
IOPB_errAck
IOPB_retry
IOPB_timeout
IOPB_toutSup
IOPB_xferAck

IOPB_rdDBus[0:31]
IOPB_errAck
IOPB_retry
IOPB_timeout
IOPB_xferAck

**MicroBlaze
Instr OPB
Interface**

IOPB_MGrant

IM_ABus[0:31]
IM_BE[0:3]
IM_busLock
IM_RNW
IM_select
IM_seqAddr
IM_request

OR

like

suffixes

Instruction-side OPB

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**

Signal	Data Interface	Instruction Interface	Type	Description
Addr[0:31]	Data_Addr[0:31]	Instr_Addr[0:31]	O	Address bus
Byte_Enable[0:3]	Byte_Enable[0:3]	*not used*	O	Byte enables
Data_Write[0:31]	Data_Write[0:31]	*not used*	O	Write data bus
AS	D_AS	I_AS	O	Address strobe
Read_Strobe	Read_Strobe	IFetch	O	Read in progress
Write_Strobe	Write_Strobe	*not used*	O	Write in progress
Data_Read[0:31]	Data_Read[0:31]	Instr[0:31]	I	Read data bus
Ready	DReady	IReady	I	Ready for next transfer
Clk	Clk	Clk	I	Bus clock

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]**

Byte_Enable[0:3]	Byte Lanes Used			
	Data[0:7]	Data[8:15]	Data[16:23]	Data[24:31]
0000				
0001				x
0010			x	
0100		x		

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

Byte_Enable[0:3]	Byte Lanes Used			
	Data[0:7]	Data[8:15]	Data[16:23]	Data[24:31]
1000	x			
0011			x	x
1100	x	x		
1111	x	x	x	x

Data_Write[0:31]

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

AS

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

Read_Strobe

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

Write_Strobe

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

Data_Read[0:31]

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

Ready

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

Clk

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

LMB Transactions

The following diagrams provide examples of LMB bus operations.

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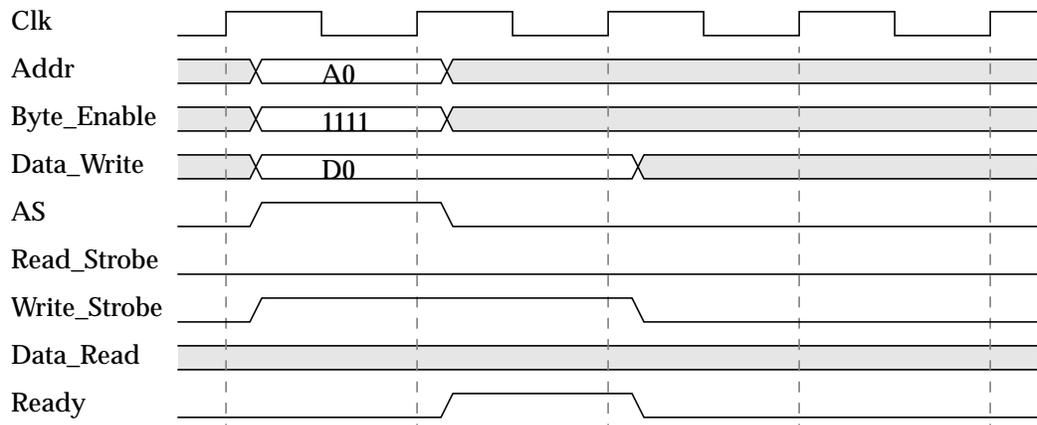
Figure 2-4: **LMB Generic Write Operation**

Generic Read Operation
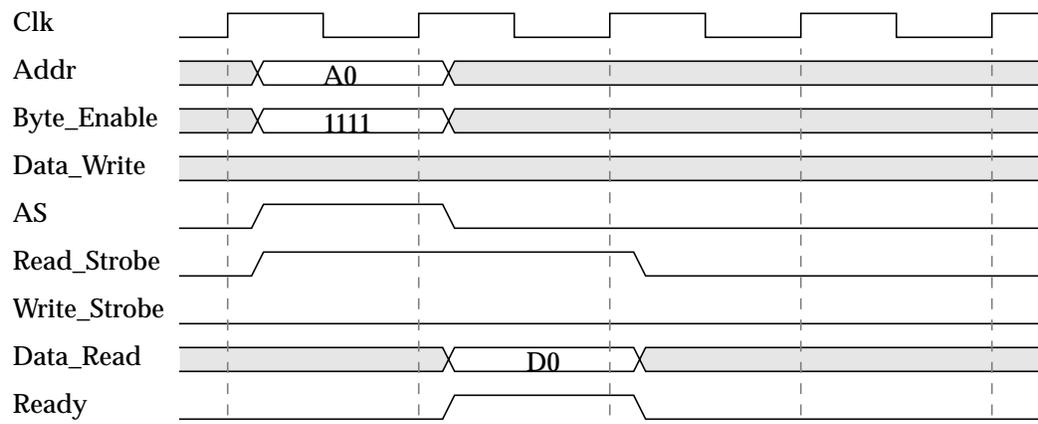
Figure 2-5: **LMB Generic Read Operation**

Back-to-Back Write Operation (Typical LMB access - 2 clocks per write)
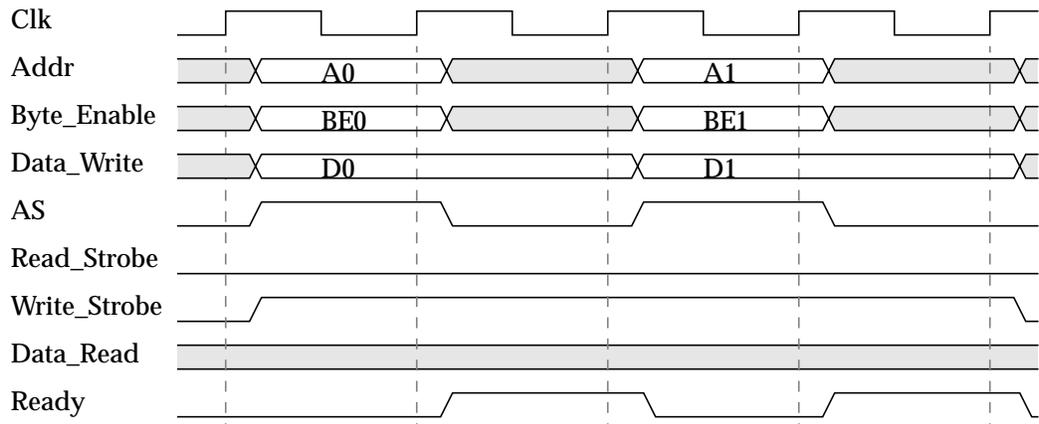
Figure 2-6: **LMB Back-to-Back Write Operation**

Single Cycle Back-to-Back Read Operation (Typical I-side access - 1 clock per read)
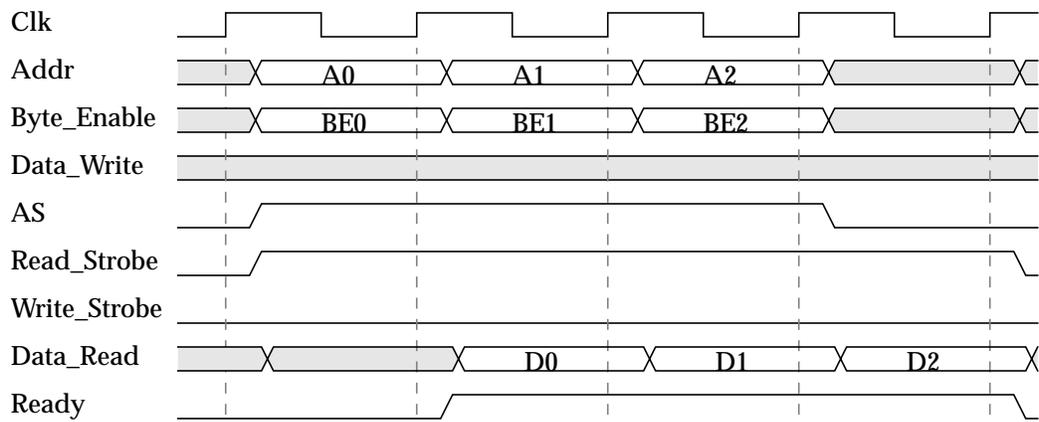
Figure 2-7: **LMB Single Cycle Back-to-Back Read Operation**

Back-to-Back Mixed Read/Write Operation (Typical D-side timing)
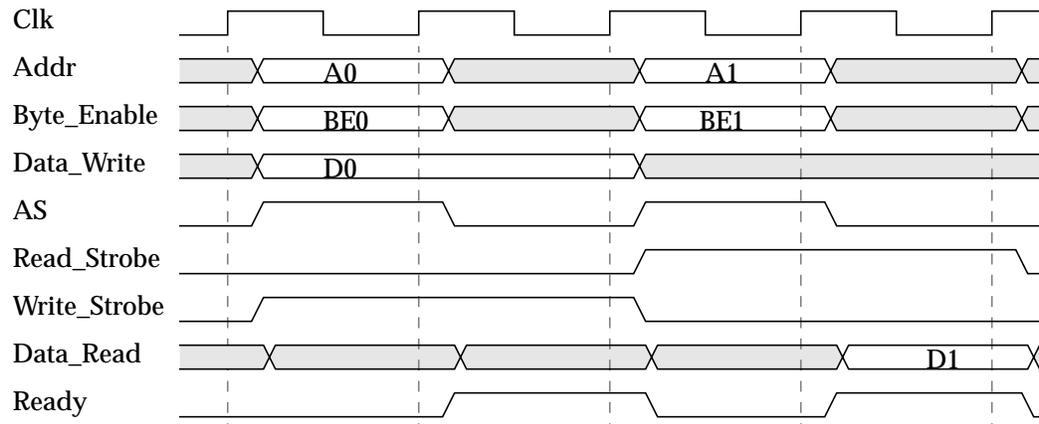
Figure 2-8: **Back-to-Back Mixed Read/Write Operation**

Read and Write Data Steering

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

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

MicroBlaze does not support transfers that are larger than the addressed device. These types of transfers require dynamic bus sizing and conversion cycles that are not supported by the MicroBlaze bus interface. Data steering for read cycles is shown in Table 2-4, and data steering for write cycles is shown in Table 2-5

Table 2-4: **Read Data Steering (load to Register rD)**

Address [30:31]	Byte_Enable [0:3]	Transfer Size	Register rD Data			
			rD[0:7]	rD[8:15]	rD[16:23]	rD[24:31]
11	0001	byte				Byte3
10	0010	byte				Byte2
01	0100	byte				Byte1
00	1000	byte				Byte0
10	0011	halfword			Byte2	Byte3
00	1100	halfword			Byte0	Byte1
00	1111	word	Byte0	Byte1	Byte2	Byte3

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

Address [30:31]	Byte_Enable [0:3]	Transfer Size	Write Data Bus Bytes			
			Byte0	Byte1	Byte2	Byte3
11	0001	byte				**rD[24:31]**
10	0010	byte			**rD[24:31]**	
01	0100	byte		**rD[24:31]**		
00	1000	byte	**rD[24:31]**			
10	0011	halfword			**rD[16:23]**	**rD[24:31]**
00	1100	halfword	**rD[16:23]**	**rD[24:31]**		
00	1111	word	**rD[0:7]**	**rD[8:15]**	**rD[16:23]**	**rD[24:31]**

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 2-6: **Master FSL signals**

Signal Name	Description	VHDL Type	Direction
FSLn_M_Clk	Clock	std_logic	input
FSLn_M_Write	Write enable signal that indicating data is ready to be written to the output FSL when set	std_logic	output
FSLn_M_Data	Data value written to the output FSL	std_logic_vector	output
FSLn_M_Control	Control bit value written to the output FSL	std_logic	output
FSLn_M_Full	Full Bit indicating output FSL FIFO is full when set	std_logic	input

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**

Signal Name	Description	VHDL Type	Direction
FSLn_S_Clk	Clock	std_logic	input
FSLn_S_Read	Read acknowledge signal indicating that data has been read from the input FSL	std_logic	output
FSLn_S_Data	Data value currently available at the top of the input FSL	std_logic_vector	input
FSLn_S_Control	Control Bit value currently available at the top of the input FSL	std_logic	input
FSLn_S_Exists	Flag indicating that data exists in the input FSL	std_logic	input

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**

Signal Name	Description	VHDL Type	Direction
ICACHE_FSL_IN_Clk	Clock output to I-side return read data FSL	std_logic	output
ICACHE_FSL_IN_Read	Read signal to I-side return read data FSL.	std_logic	output
ICACHE_FSL_IN_Data	Read data from I-side return read data FSL	std_logic_vector (0 to 31)	input
ICACHE_FSL_IN_Control	FSL control-bit from I-side return read data FSL. Reserved for future use	std_logic	input
ICACHE_FSL_IN_Exists	More read data exists in I-side return FSL	std_logic	input
ICACHE_FSL_OUT_Clk	Clock output to I-side read access FSL	std_logic	output
ICACHE_FSL_OUT_Write	Write new cache miss access request to I-side read access FSL	std_logic	output
ICACHE_FSL_OUT_Data	Cache miss access (=address) to I-side read access FSL	std_logic_vector (0 to 31)	output
ICACHE_FSL_OUT_Control	FSL control-bit to I-side read access FSL. Reserved for future use	std_logic	output
ICACHE_FSL_OUT_Full	FSL access buffer for I-side read accesses is full	std_logic	input
DCACHE_FSL_IN_Clk	Clock output to D-side return read data FSL	std_logic	output

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

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

CacheLink Transactions

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

- Access information is encoded over the FSL data and control signals (e.g. DCACHE_FSL_OUT_Data, DCACHE_FSL_OUT_Control, ICACHE_FSL_IN_Data, and ICACHE_FSL_IN_Control)

- Information is sent (stored) by raising the write enable signal (e.g. DCACHE_FSL_OUT_Write).

- The sender is only allowed to write if the full signal from the receiver is inactive (e.g. DCACHE_FSL_OUT_Full = 0).

- Information is received (loaded) by raising the read signal (e.g. ICACHE_FSL_IN_Read)

- The receiver is only allowed to read as long as the sender signals that new data exists (e.g. ICACHE_FSL_IN_Exists = 1).

For details on the generic FSL protocol please refer to the "Fast Simplex Link (FSL) bus" data sheet (DS449).

The CacheLink solution uses one incoming (slave) and one outgoing (master) FSL per cache controller. The outgoing FSL is used to send access requests, while the incoming FSL is used for receiving the requested cache lines. CacheLink also uses a specific encoding of the transaction information over the FSL data and control signals.

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

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

Instruction and Data Cache Read Miss

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

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

Data Cache Write

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

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

Debug Interface Description

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

---

1. Byte and halfword read misses are naturally expected to return complete words, the cache controller then provides the execution unit with the correct bytes.

MicroBlaze processors at the same time. The debug signals on MicroBlaze are listed in Table 2-9.

Table 2-9:   **MicroBlaze Debug signals**

Signal Name	Description	VHDL Type	Direction
Dbg_Clk	JTAG Clock from MDM	std_logic	input
Dbg_TDI	JTAG TDI from MDM	std_logic	input
Dbg_TDO	JTAG TDO to MDM	std_logic	output
Dbg_Reg_En	Debug Register Enable from MDM	std_logic	input
Dbg_Capture	JTAG BSCAN Capture signal from MDM	std_logic	input
Dbg_Update	JTAG BSCAN Update signal from MDM	std_logic	input

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**

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

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

Signal Name	Description	VHDL Type	Direction
Jump_Taken	Branch instruction evaluated true	std_logic	output
Prefetch_Addr	Which position in the prefetch buffer should be used for the decode stage in the next pipeline shift	std_logic_vector (0 to 3)	output
MB_Halted	Processor pipeline execution is halted	std_logic	output
Trace_Branch_Instr	Instruction to be executed is a branch instruction	std_logic	output
Trace_Delay_Slot	Current cycle is a part of multi-cycle instruction execution	std_logic	output
Trace_Data_Address	Address for D-side memory access	std_logic_vector (0 to 31)	output
Trace_AS	Trace_Data_Address is valid	std_logic	output
Trace_Data_Read	D-side memory access is a read	std_logic	output
Trace_Data_Write	D-side memory access is a write	std_logic	output
Trace_DCache_Req	Data memory address is in D-Cache range	std_logic	output
Trace_DCache_Hit	Data memory address is present in D-Cache	std_logic	output
Trace_ICache_Req	Instruction memory address is in I-Cache range	std_logic	output
Trace_ICache_Hit	Instruction memory address is present in I-Cache	std_logic	output
Trace_Instr_EX	Execute stage instruction code	std_logic_vector (0 to 31)	output

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 respectivly.

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 2-11: **MPD Parameters**

Parameter Name	Feature/Description	Allowable Values	Default Value	EDK Tool Assigned	VHDL Type
C_FAMILY	Target Family	qrvirtex2 qvirtex2 spartan2 spartan2e spartan3 virtex virtex2 virtex2p virtex4 virtexe	virtex2	yes	string
C_DATA_SIZE	Data Size	32	32	NA	integer
C_INSTANCE	Instance Name	Any instance name	microb laze	yes	string
C_D_OPB	Data side OPB interface	0, 1	1	yes	integer
C_D_LMB	Data side LMB interface	0, 1	1	yes	integer
C_I_OPB	Instruction side OPB interface	0, 1	1	yes	integer
C_I_LMB	Instruction side LMB interface	0, 1	1	yes	integer
C_USE_BARREL	Barrel Shifter	0, 1	0		integer
C_USE_DIV	Divide Unit	0, 1	0		integer
C_USE_MSR_INSTR	Enable use of instructions: MSRSET and MSRCLR	0, 1	0		integer
C_UNALIGNED_EXCEPTION	Enable exception handling for unaligned data accesses	0, 1	0		integer
C_ILL_OPCODE_EXCEPTION	Enable exception handling for illegal op-code	0, 1	0		integer
C_IOPB_BUS_EXCEPTION	Enable exception handling for IOPB bus error	0, 1	0		integer
C_DOPB_BUS_EXCEPTION	Enable exception handling for DOPB bus error	0, 1	0		integer
C_DIV_ZERO_EXCEPTION	Enable exception handling for division by zero	0, 1	0		integer
C_DEBUG_ENABLED	MDM Debug interface	0,1	0		integer

Table 2-11: **MPD Parameters**

Parameter Name	Feature/Description	Allowable Values	Default Value	EDK Tool Assigned	VHDL Type
C_NUMBER_OF_PC_BRK	Number of hardware breakpoints	0-8	1		integer
C_NUMBER_OF_RD_ADDR_BRK	Number of read address watchpoints	0-4	0		integer
C_NUMBER_OF_WR_ADDR_BRK	Number of write address watchpoints	0-4	0		integer
C_INTERRUPT_IS_EDGE	Level/Edge Interrupt	0, 1	0		integer
C_EDGE_IS_POSITIVE	Negative/Positive Edge Interrupt	0, 1	1		integer
C_FSL_LINKS	Number of FSL interfaces	0-8	0	yes	integer
C_FSL_DATA_SIZE	FSL data bus size	32	32	NA	integer
C_ICACHE_BASEADDR	Instruction cache base address	0x00000000 - 0xFFFFFFFF	0x0000 0000		std_logic _vector
C_ICACHE_HIGHADDR	Instruction cache high address	0x00000000 - 0xFFFFFFFF	0x3FFF FFFF		std_logic _vector
C_USE_ICACHE	Instruction cache	0,1	0		integer
C_ALLOW_ICACHE_WR	Instruction cache write enable	0,1	1		integer
C_ADDR_TAG_BITS	Instruction cache address tags	0-24	7	yes	integer
C_CACHE_BYTE_SIZE	Instruction cache size	512, 1024, 2048, 4096, 8192, 16384, 32768, 65536[1]	8192		integer
C_ICACHE_USE_FSL	Cache over CacheLink instead of OPB for instructions	0,1	0		integer
C_DCACHE_BASEADDR	Data cache base address	0x00000000 - 0xFFFFFFFF	0x0000 0000		std_logic _vector
C_DCACHE_HIGHADDR	Data cache high address	0x00000000 - 0xFFFFFFFF	0x3FFF FFFF		std_logic _vector
C_USE_DCACHE	Data cache	0,1	0		integer
C_ALLOW_DCACHE_WR	Data cache write enable	0,1	1		integer
C_DCACHE_ADDR_TAG	Data cache address tags	0-24	7	yes	integer

Table 2-11: **MPD Parameters**

Parameter Name	Feature/Description	Allowable Values	Default Value	EDK Tool Assigned	VHDL Type
C_DCACHE_BYTE_SIZE	Data cache size	2048, 4096, 8192, 16384, 32768, 65536[2]	8192		integer
C_DCACHE_USE_FSL	Cache over CacheLink instead of OPB for data	0,1	0		integer

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**

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

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.

Table 3-2: **Register usage conventions**

Register	Type	Enforcement	Purpose
R0	Dedicated	HW	Value 0
R1	Dedicated	SW	Stack Pointer
R2	Dedicated	SW	Read-only small data area anchor
R3-R4	Volatile	SW	Return Values/Temporaries
R5-R10	Volatile	SW	Passing parameters/Temporaries
R11-R12	Volatile	SW	Temporaries
R13	Dedicated	SW	Read-write small data area anchor
R14	Dedicated	HW	Return address for Interrupt
R15	Dedicated	SW	Return address for Sub-routine
R16	Dedicated	HW	Return address for Trap (Debugger)
R17	Dedicated	HW, if configured to support HW exceptions, else SW	Return Address for Exceptions
R18	Dedicated	SW	Reserved for Assembler
R19-R31	Non-volatile	SW	Must be saved across function calls. Callee-save
RPC	Special	HW	Program counter
RMSR	Special	HW	Machine Status Register

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

- The volatile registers are used as temporaries and do not retain values across the function calls. Registers R3 through R12 are volatile, of which R3 and R4 are used for returning values to the caller function, if any. Registers R5 through R10 are used for passing parameters between sub-routines.

- Registers R19 through R31 retain their contents across function calls and are hence termed as non-volatile registers. The callee function is expected to save those non-volatile registers, which are being used. These are typically saved to the stack during the prologue and then reloaded during the epilogue.

- Certain registers are used as dedicated registers and programmers are not expected to use them for any other purpose.

    ♦ Registers R14 through R17 are used for storing the return address from interrupts, sub-routines, traps and exceptions in that order. Sub-routines are called using the branch and link instruction, which saves the current Program Counter (PC) onto register R15.

    ♦ Small data area pointers are used for accessing certain memory locations with 16 bit immediate value. These areas are discussed in the memory model section of this document. The read only small data area (SDA) anchor R2 (Read-Only) is

used to access the constants such as literals. The other SDA anchor R13 (Read-Write) is used for accessing the values in the small data read-write section.

 ♦ Register R1 stores the value of the stack pointer and is updated on entry and exit from functions.

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

• MicroBlaze has certain special registers such as a program counter (rpc) and machine status register (rmsr). These registers are not mapped directly to the register file and hence the usage of these registers is different from the general purpose registers. The value from rmsr and rpc can be transferred to general purpose registers by using **mts** and **mfs** instructions (For more details refer to the "MicroBlaze Application Binary Interface" chapter).

Stack Convention

The stack conventions used by MicroBlaze are detailed in Figure 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**

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

Figure 3-1: **Stack Convention**

New Stack Pointer	Link Register
Low Address	

Consider an example where Func1 calls Func2, which in turn calls Func3. The stack representation at different instances is depicted in Figure 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.
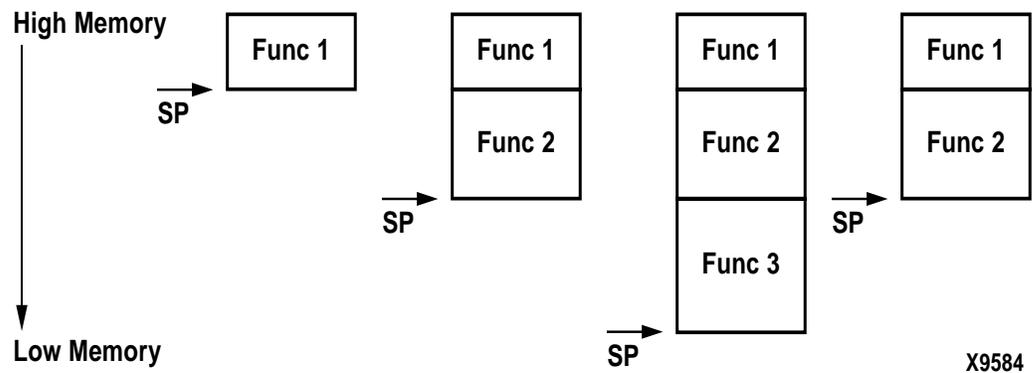
Figure 3-2: **Stack Frame**

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 3-3: **Interrupt and Exception Handling**

On	Hardware jumps to	Software Labels
Start / Reset	0x0	_start
User exception	0x8	_exception_handler
Interrupt	0x10	_interrupt_handler
Hardware exception	0x20	_hw_exception_handler

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.

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

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

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

In the current MicroBlaze system, there are dummy routines for interrupt and exception handling, which you can change. In order to override these routines and link your interrupt and exception handlers, you must define the interrupt handler code with an attribute **interrupt_handler**. For more details about the use and syntax of the interrupt handler attribute, please refer to the *GNU Compiler Tools* chapter in the document: UG111 *Embedded System Tools Reference Manual.*

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

Symbol	Meaning
+	Add
-	Subtract
×	Multiply
∧	Bitwise logical AND
∨	Bitwise logical OR
⊕	Bitwise logical XOR
$\bar{x}$	Bitwise logical complement of x
←	Assignment
>>	Right shift
<<	Left shift
rx	Register x
$x[i]$	Bit i in register x
$x[i:j]$	Bits i through j in register x
=	Equal comparison
≠	Not equal comparison
>	Greater than comparison
>=	Greater than or equal comparison
<	Less than comparison
<=	Less than or equal comparison
sext(x)	Sign-extend x

Table 1: Symbol notation

Symbol	Meaning
Mem(*x*)	Memory location at address *x*
FSLx	FSL interface x
LSW(*x*)	Least Significant Word of *x*

Formats

MicroBlaze uses two instruction formats: Type A and Type B.

Type A

Type A is used for register-register instructions. It contains the opcode, one destination and two source registers.

Opcode	Destination Reg	Source Reg A	Source Reg B	0	0	0	0	0	0	0	0	0	0	0
0	6	11	16	21										31

Type B

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

Opcode	Destination Reg	Source Reg A	Immediate Value
0	6	11	16 31

Instructions

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

add

Arithmetic Add

add	rD, rA, rB	Add
addc	rD, rA, rB	Add with Carry
addk	rD, rA, rB	Add and Keep Carry
addkc	rD, rA, rB	Add with Carry and Keep Carry

0 0 0 K C 0	rD	rA	rB	0 0 0 0 0 0 0 0 0 0 0
0	6	11	16	21 31
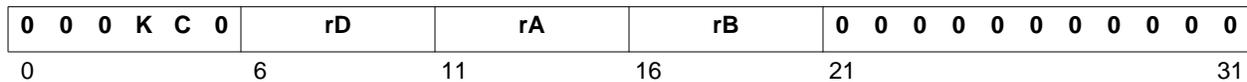
Description

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

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

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Note

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

addi

Arithmetic Add Immediate

addi	rD, rA, IMM	Add Immediate
addic	rD, rA, IMM	Add Immediate with Carry
addik	rD, rA, IMM	Add Immediate and Keep Carry
addikc	rD, rA, IMM	Add Immediate with Carry and Keep Carry

0 0 1 K C 0	rD	rA	IMM
0	6	11	16 31
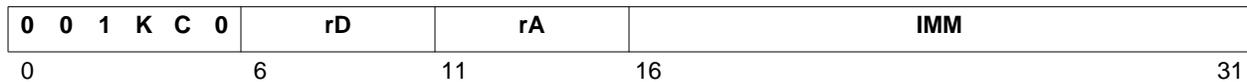
Description

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

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

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

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

and

Logical AND

and   rD, rA, rB

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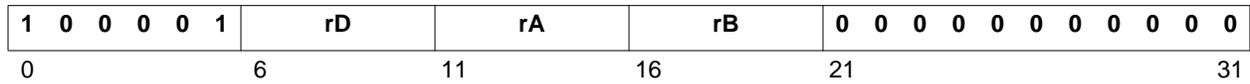
Description

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

Pseudocode

(rD) ← (rA) ∧ (rB)

Registers Altered

- rD

Latency

1 cycle

andi

Logial AND with Immediate

      andi   rD, rA, IMM

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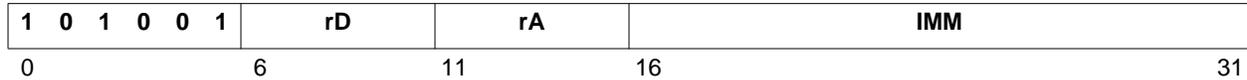
Description

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

Pseudocode

```
(rD) ← (rA) ∧ sext(IMM)
```

Registers Altered

- rD

Latency

1 cycle

Note

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

andn

Logical AND NOT

andn			rD, rA, rB					

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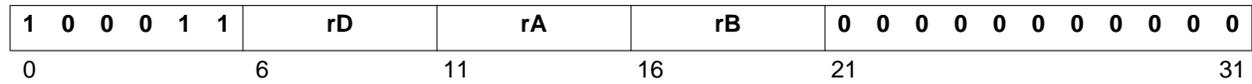
Description

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

Pseudocode

$(rD) \leftarrow (rA) \wedge (\overline{rB})$

Registers Altered

- rD

Latency

1 cycle

andni

Logical AND NOT with Immediate

andni		rD, rA, IMM		

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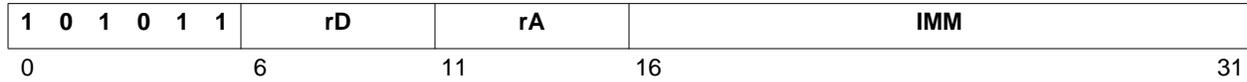
Description

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

Pseudocode

$(rD) \leftarrow (rA) \wedge (\overline{sext(IMM)})$

Registers Altered

- rD

Latency

1 cycle

Note

By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See 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
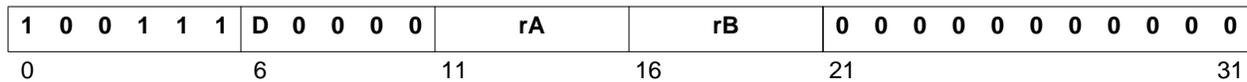
1	0	0	1	1	1	D	0	0	0	0	rA	rB	0	0	0	0	0	0	0	0	0	0	0

0　　　　　　　　6　　　　　　11　　　　　16　　　　21　　　　　　　　　　　　　　　31

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

beqi

Branch Immediate if Equal

beqi	rA, IMM	Branch Immediate if Equal
beqid	rA, IMM	Branch Immediate if Equal with Delay
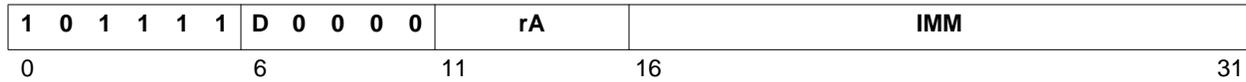
1	0	1	1	1	1	D	0	0	0	0	rA	IMM
0						6					11	16 31

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

Note

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

bge

Branch if Greater or Equal

bge	rA, rB	Branch if Greater or Equal
bged	rA, rB	Branch if Greater or Equal with Delay
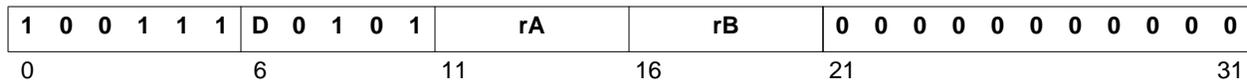
1	0	0	1	1	1	D	0	1	0	1	rA	rB	0	0	0	0	0	0	0	0	0	0	0

0   6   11   16   21   31

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

bgei

Branch Immediate if Greater or Equal

bgei	rA, IMM	Branch Immediate if Greater or Equal
bgeid	rA, IMM	Branch Immediate if Greater or Equal with Delay
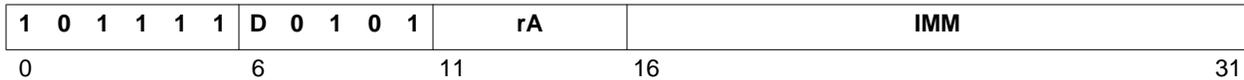
1	0	1	1	1	1	D	0	1	0	1	rA	IMM

0	6	11	16	31

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.

bgt

Branch if Greater Than

bgt	rA, rB	Branch if Greater Than
bgtd	rA, rB	Branch if Greater Than with Delay
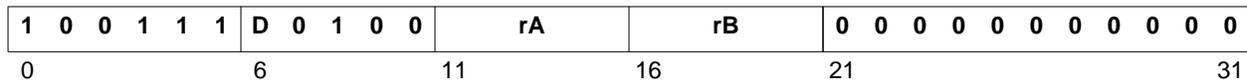
1	0	0	1	1	1	D	0	1	0	0	rA	rB	0	0	0	0	0	0	0	0	0	0	0
0						6					11	16	21										31

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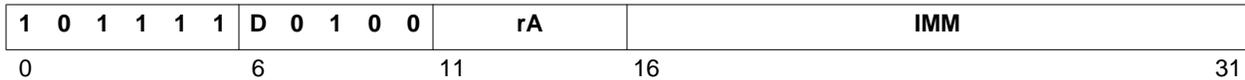
1 0 1 1 1 1	D 0 1 0 0	rA	IMM
0	6	11	16 31

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	
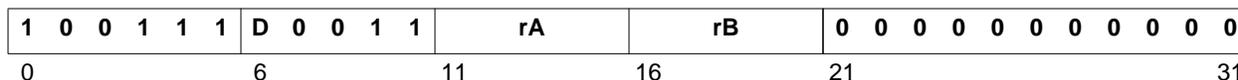
1	0	0	1	1	1	D	0	0	1	1	rA	rB	0	0	0	0	0	0	0	0	0	0	0

0   6   11   16   21   31

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

blei

Branch Immediate if Less or Equal

blei	rA, IMM	Branch Immediate if Less or Equal
bleid	rA, IMM	Branch Immediate if Less or Equal with Delay
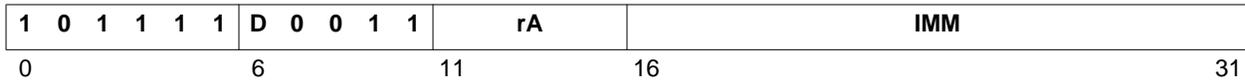
1	0	1	1	1	1	D	0	0	1	1	rA	IMM
0						6					11	16 31

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

Note

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

blt

Branch if Less Than

blt	rA, rB	Branch if Less Than
bltd	rA, rB	Branch if Less Than with Delay

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

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

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

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

bne	rA, rB	Branch if Not Equal
bned	rA, rB	Branch if Not Equal with Delay
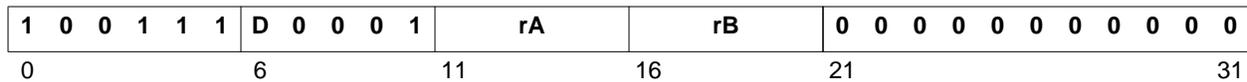
1	0	0	1	1	1	D	0	0	0	1	rA	rB	0	0	0	0	0	0	0	0	0	0	0
0						6					11	16	21										31

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

bnei

Branch Immediate if Not Equal

| **bnei** | rA, IMM | Branch Immediate if Not Equal |
| **bneid** | rA, IMM | Branch Immediate if Not Equal with Delay |

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

Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

2 cycles (if branch is taken and the D bit is set)

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

Note

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

br

Unconditional Branch

br	rB	Branch
bra	rB	Branch Absolute
brd	rB	Branch with Delay
brad	rB	Branch Absolute with Delay
brld	rD, rB	Branch and Link with Delay
brald	rD, rB	Branch Absolute and Link with Delay

1 0 0 1 1 0	rD	D A L 0 0	rB	0 0 0 0 0 0 0 0 0 0 0
0	6	11	16	21 31
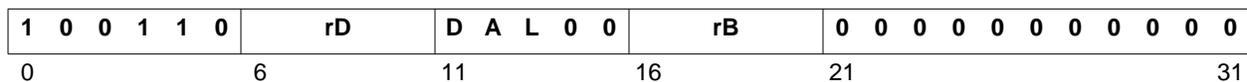
Description

Branch to the instruction located at address determined by rB.

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

The mnemonics bra, brad and brald will set the A bit. If the A bit is set, it means that the branch is to an absolute value and the target is the value in rB, otherwise, it is a relative branch and the target will be PC + rB.

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

Pseudocode

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

Registers Altered

- rD
- PC

Latency

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

Note

The instructions brl and bral are not available.

bri

Unconditional Branch Immediate

bri	IMM	Branch Immediate
brai	IMM	Branch Absolute Immediate
brid	IMM	Branch Immediate with Delay
braid	IMM	Branch Absolute Immediate with Delay
brlid	rD, IMM	Branch and Link Immediate with Delay
bralid	rD, IMM	Branch Absolute and Link Immediate with Delay

1 0 1 1 1 0	rD	D A L 0 0	IMM
0	6	11	16 31
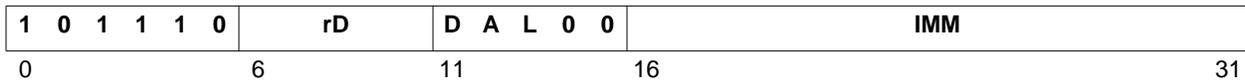
Description

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

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

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

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

Pseudocode

```
if L = 1 then
  (rD) ← PC
if A = 1 then
  PC ← (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)
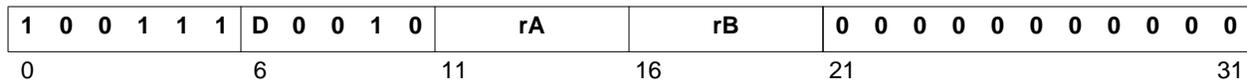
Notes

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

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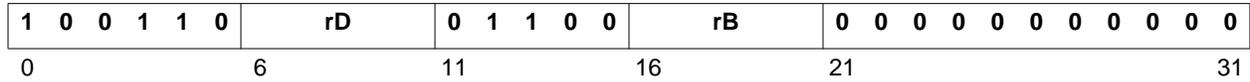
Description

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

Pseudocode

```
(rD) ← PC
PC ← (rB)
MSR[BIP] ← 1
```

Registers Altered

- rD
- PC
- MSR[BIP]

Latency

3 cycles

brki

Break Immediate

brki rD, IMM

1 0 1 1 1 0	rD	0 1 1 0 0	IMM
0	6	11 16	31
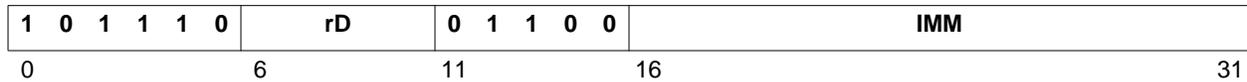
Description

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.

bs

Barrel Shift

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

0 1 0 0 0 1	rD	rA	rB	S T 0 0 0 0 0 0 0 0 0
0	6	11	16	21 31

Description

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

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

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

Note

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

bsi

Barrel Shift Immediate

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

0 1 1 0 0 1	rD	rA	0 0 0 0 0	S T 0 0 0 0	IMM
0	6	11	16	21	27 31
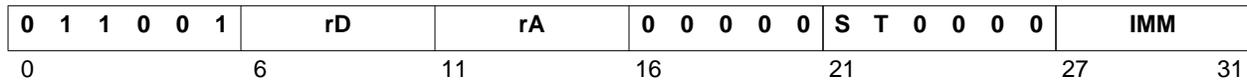
Description

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

| cmp | rD, rA, rB | compare rB with rA (signed) |
| cmpu | rD, rA, rB | compare rB with rA (unsigned) |
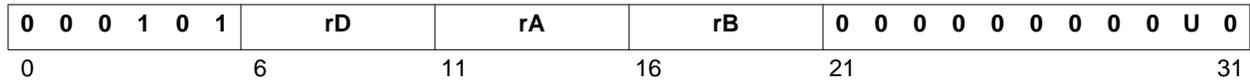
0	0	0	1	0	1	rD	rA	rB	0	0	0	0	0	0	0	0	0	U	0
0						6	11	16	21										31

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

```
(rD) ← (rB) + (r̄A) + 1
(rD)(MSB) ← (rA) > (rB)
```

Registers Altered

- rD

Latency

1 cycle

.

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)

0 1 1 0 1 1	rD	0 0 0 0 0	0 n c 0 0 0 0 0 0 0 0 0 0 0	FSLx
0	6	11	16	29 31

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

Registers Altered

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

Latency

2 cycles 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 rsubc instruction can be used for counting down a index variable

idiv

Integer Divide

idiv	rD, rA, rB	divide rB by rA (signed)
idivu	rD, rA, rB	divide rB by rA (unsigned)

0	1	0	0	1	0	rD	rA	rB	0	0	0	0	0	0	0	0	0	U	0

0   6   11   16   21   31
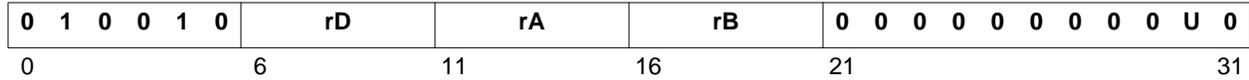
Description

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

If the U bit is set, rA and rB is considered unsigned values. If the U bit is clear, rA and rB is considered signed values

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

Pseudocode

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

Registers Altered

- rD
- MSR[Divide_By_Zero]

Latency

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

Note

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

.imm

Immediate

imm   IMM

1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	IMM
0						6					11					16 31
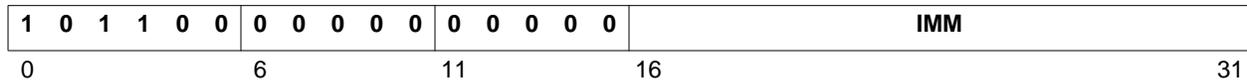
Description

The instruction imm loads the IMM value into a temporary register. It also locks this value so it can be used by the following instruction and form a 32-bit immediate value.

The instruction imm is used in conjunction with Type B instructions. Since Type B instructions have only a 16-bit immediate value field, a 32-bit immediate value cannot be used directly. However, 32-bit immediate values can be used in MicroBlaze. By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. The imm instruction locks the 16-bit IMM value temporarily for the next instruction. A Type B instruction that immediately follows the imm instruction will then form a 32-bit immediate value from the 16-bit IMM value of the imm instruction (upper 16 bits) and its own 16-bit immediate value field (lower 16 bits). If no Type B instruction follows the IMM instruction, the locked value gets unlocked and becomes useless.

Latency

1 cycle

Notes

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

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

lbu

Load Byte Unsigned

lbu   rD, rA, rB

1	1	0	0	0	0	rD	rA	rB	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

```
Addr ← (rA) + (rB)
(rD)[24:31] ← Mem(Addr)
(rD)[0:23]  ← 0
```

Registers Altered

- rD

Latency

2 cycles

lbui

Load Byte Unsigned Immediate

lbui			rD, rA, IMM						
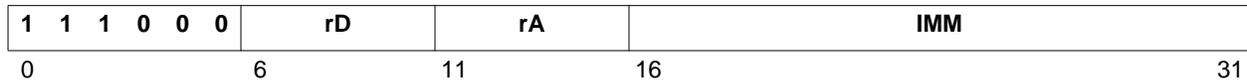
1	1	1	0	0	0	rD	rA	IMM
0						6	11	16 31

Description

Loads a byte (8 bits) from the memory location that results from adding the contents of register rA with the value in IMM, sign-extended to 32 bits. The data is placed in the least significant byte of register rD and the other three bytes in rD are cleared.

Pseudocode

```
Addr ← (rA) + sext(IMM)
(rD)[24:31] ← Mem(Addr)
(rD)[0:23] ← 0
```

Registers Altered

- rD

Latency

2 cycles

Note

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

lhu

Load Halfword Unsigned

lhu   rD, rA, rB

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

Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

lhui

Load Halfword Unsigned Immediate

lhui			rD, rA, IMM					
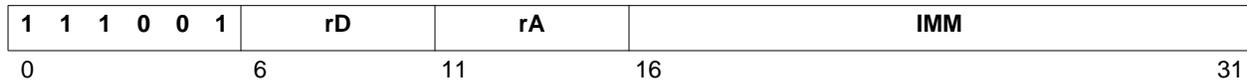
1	1	1	0	0	1	rD	rA	IMM
0						6	11	16 31

Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

Note

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

lw

Load Word

lw   rD, rA, rB

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

Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

lwi

Load Word Immediate

lwi rD, rA, IMM

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

Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

Note

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

mfs

Move From Special Purpose Register

        mfs   rD, rS

1 0 0 1 0 1	rD	0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	rS
0	6	11	16	31
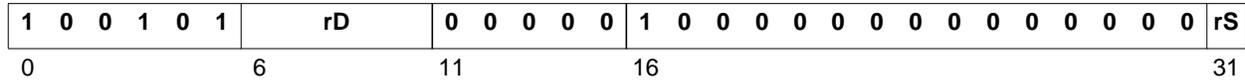
Description

Copies the contents of the special purpose register rS into register rD.

Pseudocode

(rD) ← (rS)

Registers Altered

- rD

Latency

1 cycle

Note

To refer to special purpose registers in assembly language, use rpc for PC, 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

msrclr   rD, Imm
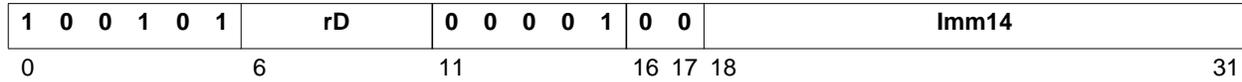
1	0	0	1	0	1	rD	0	0	0	0	1	0	0	Imm14
0						6	11					16	17 18	31

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)   ← (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 cleared.

This instruction only exists in version 2.10.a and above.

msrset

Read MSR and set bits in MSR

msrset   rD, Imm
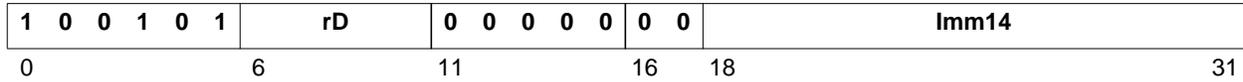
1	0	0	1	0	1	rD	0	0	0	0	0	0	0	Imm14

0   6   11   16 18   31

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

1	0	0	1	0	1	0	0	0	0	0	rA	1	1	0	0	0	0	0	0	0	0	0	0	0	0	rS

0   6   11   16   31
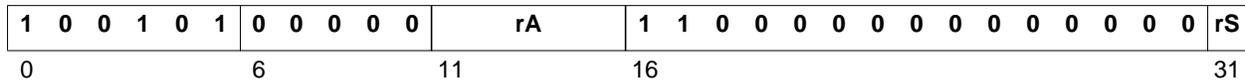
Description

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

mul   rD, rA, rB

0	1	0	0	0	0	rD	rA	rB	0	0	0	0	0	0	0	0	0	0	0	0

0   6   11   16   21   31
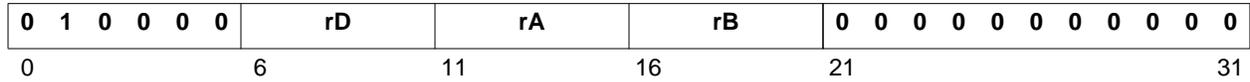
Description

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) ← LSW((rA) × (rB))

Registers Altered

- rD

Latency

3 cycles

Note

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

muli

Multiply Immediate

muli   rD, rA, IMM
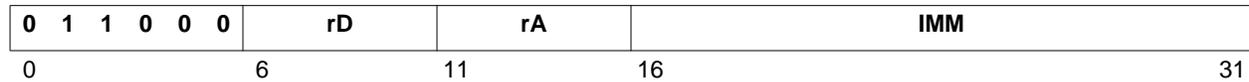
0	1	1	0	0	0	rD	rA	IMM
0						6	11	16 31

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

$(rD) \leftarrow LSW((rA) \times sext(IMM))$

Registers Altered
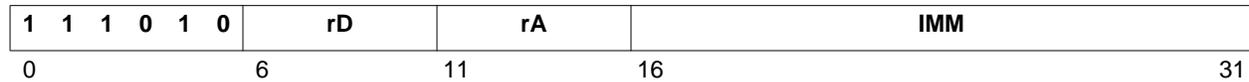
- rD

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.

or

Logical OR

or	rD, rA, rB

1	0	0	0	0	0	rD	rA	rB	0	0	0	0	0	0	0	0	0	0	0

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

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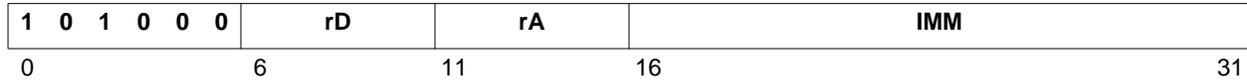
Description

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

$(rD) \leftarrow (rA) \vee (IMM)$

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

put	rA, FSLx	put data to FSL x (blocking)
nput	rA, FSLx	put data to FSL x (non-blocking)
cput	rA, FSLx	put control to FSL x (blocking)
ncput	rA, FSLx	put control to FSL x (non-blocking)

0 1 1 0 1 1	0 0 0 0 0	rA	1 n c 0 0 0 0 0 0 0 0 0 0	FSLx
0	6	11	16	29 31
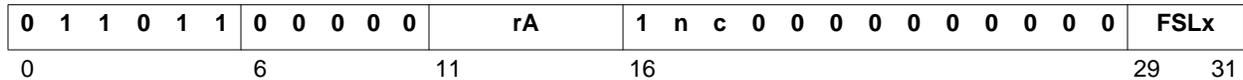
Description

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) ← (rA)
if (n = 1) then
  MSR[Carry] ← (FSLx Full bit)
(FSLx Control bit) ← 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.

rsub

Arithmetic Reverse Subtract

rsub	rD, rA, rB	Subtract
rsubc	rD, rA, rB	Subtract with Carry
rsubk	rD, rA, rB	Subtract and Keep Carry
rsubkc	rD, rA, rB	Subtract with Carry and Keep Carry
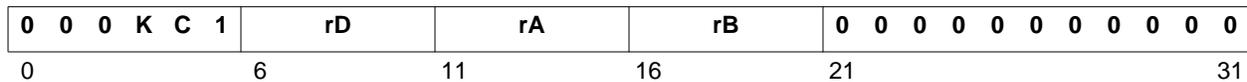
0 0 0 K C 1	rD	rA	rB	0 0 0 0 0 0 0 0 0 0 0
0	6	11	16	21 31

Description

The contents of register rA is subtracted from the contents of register rB and the result is placed into register rD. Bit 3 of the instruction (labeled as K in the figure) is set to a one for the mnemonic rsubk. Bit 4 of the instruction (labeled as C in the figure) is set to a one for the mnemonic rsubc. Both bits are set to a one for the mnemonic rsubkc.

When an rsub instruction has bit 3 set (rsubk, rsubkc), the carry flag will Keep its previous value regardless of the outcome of the execution of the instruction. If bit 3 is cleared (rsub, rsubc), then the carry flag will be affected by the execution of the instruction.

When bit 4 of the instruction is set to a one (rsubc, rsubkc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (rsub, rsubk), the content of the carry flag does not affect the execution of the instruction (providing a normal subtraction).

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

In subtractions, Carry = ($\overline{\text{Borrow}}$). When the Carry is set by a subtraction, it means that there is no Borrow, and when the Carry is cleared, it means that there is a Borrow.

rsubi

Arithmetic Reverse Subtract Immediate

rsubi	rD, rA, IMM	Subtract Immediate
rsubic	rD, rA, IMM	Subtract Immediate with Carry
rsubik	rD, rA, IMM	Subtract Immediate and Keep Carry
rsubikc	rD, rA, IMM	Subtract Immediate with Carry and Keep Carry

0	0	1	K	C	1	rD	rA	IMM
0						6	11	16 31

Description

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

In subtractions, Carry = $(\overline{Borrow})$. 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.
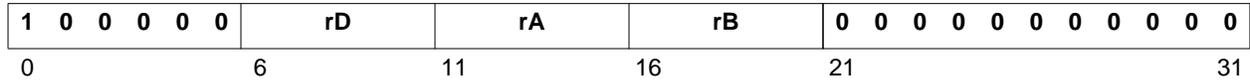
By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.

rtbd

Return from Break

rtbd   rA, IMM

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

Description

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

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

Pseudocode

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

Registers Altered

- PC
- MSR[BIP]

Latency

2 cycles

rtid

Return from Interrupt

    rtid   rA, IMM

1	0	1	1	0	1	1	0	0	0	1	rA	IMM
0						6					11 16	31
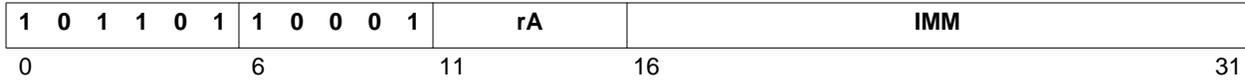
Description

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

1	0	1	1	0	1	1	0	0	1	0	rA	IMM
0						6					11 16	31
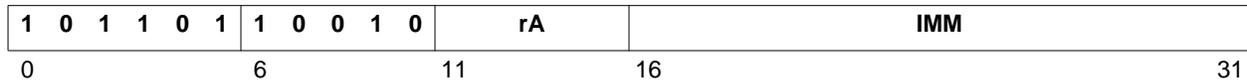
Description

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

```
PC ← (rA) + sext(IMM)
allow following instruction to complete execution
MSR[EE] ← 1
MSR[EIP] ← 0
ESR ← 0
```

Registers Altered

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

Latency

2 cycles

Notes

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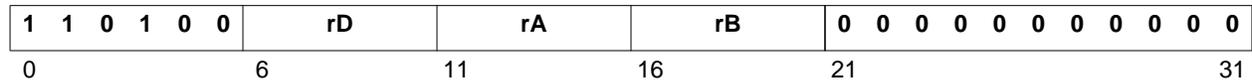
rtsd

Return from Subroutine

rtsd   rA, IMM

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

Description

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

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

Pseudocode

```
PC ← (rA) + sext(IMM)
allow following instruction to complete execution
```

Registers Altered

- PC

Latency

2 cycles

sb

Store Byte

sb   rD, rA, rB

1	1	0	1	0	0	rD	rA	rB	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

```
Addr ← (rA) + (rB)
Mem(Addr) ← (rD)[24:31]
```

Registers Altered

- None

Latency

2 cycles

sbi

Store Byte Immediate

sbi   rD, rA, IMM

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

Description

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

Pseudocode

```
Addr ← (rA)+ sext(IMM)
Mem(Addr) ←(rD)[24:31]
```

Registers Altered

- None

Latency

2 cycles

Note

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

sext16   **Sign Extend Halfword**

        sext16   rD, rA

1	0	0	1	0	0	rD	rA	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1

0   6   11   16   31
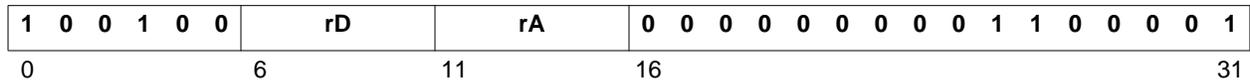
Description

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

Pseudocode

```
(rD)[0:15] ← (rA)[16]
(rD)[16:31] ← (rA)[16:31]
```

Registers Altered

- rD

Latency

1 cycle

sext8

Sign Extend Byte

sext8   rD, rA

1	0	0	1	0	0	rD	rA	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

0   6   11   16   31
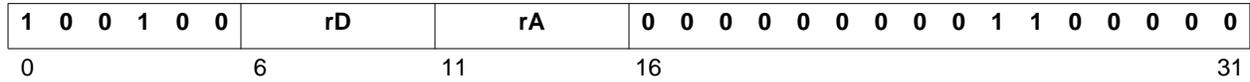
Description

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

Pseudocode

```
(rD)[0:23] ← (rA)[24]
(rD)[24:31] ← (rA)[24:31]
```

Registers Altered

- rD

Latency

1 cycle

sh

Store Halfword

sh   rD, rA, rB
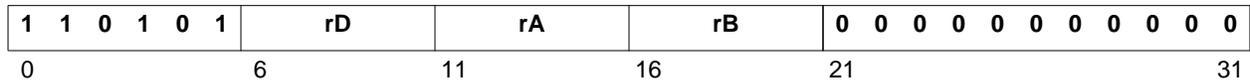
1 1 0 1 0 1	rD	rA	rB	0 0 0 0 0 0 0 0 0 0 0
0	6	11	16	21 31

Description

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

Pseudocode

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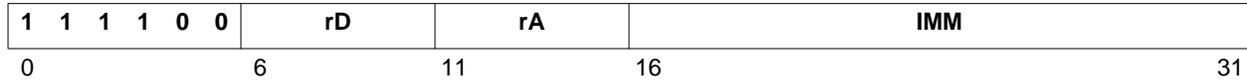
Registers Altered

- None

Latency

2 cycles

shi

Store Halfword Immediate

shi   rD, rA, IMM

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

Description

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

Pseudocode

```
Addr ← (rA) + sext(IMM)
Addr[31] ← 0
Mem(Addr) ← (rD)[16:31]
```

Registers Altered

- None

Latency

2 cycles

Note

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

sra

Shift Right Arithmetic

sra   rD, rA

1	0	0	1	0	0	rD	rA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

0   6   11   16   31

Description

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

src

Shift Right with Carry

src   rD, rA

1	0	0	1	0	0	rD	rA	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1

0   6   11   16   31

Description

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

srl

Shift Right Logical

srl   rD, rA

1 0 0 1 0 0	rD	rA	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
0	6	11	16 31
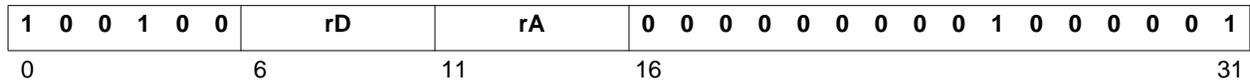
Description

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	rD, rA, rB			

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

Description

Stores the contents of register rD, into the word aligned memory location that results from adding the contents of registers rA and rB.

Pseudocode

```
Addr ← (rA) + (rB)
Addr[30:31] ← 00
Mem(Addr) ← (rD)[0:31]
```
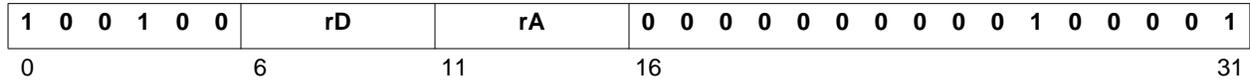
Registers Altered

- None

Latency

2 cycles

swi

Store Word Immediate

swi			rD, rA, IMM				
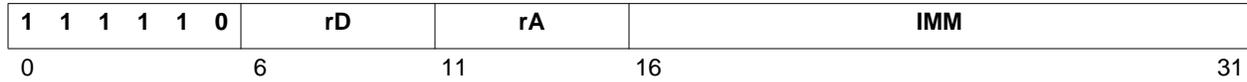
1	1	1	1	1	0	rD	rA	IMM
0						6	11	16 31

Description

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

Pseudocode

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

Register Altered

- None

Latency

2 cycles

Note

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

wdc

Write to Data Cache

wdc   rA,rB

1	0	0	1	0	0	rA	rA	rB	0	0	0	0	1	1	0	0	1	0	0

0   6   11   16   31
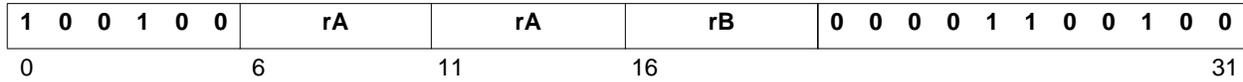
Description

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   rA,rB

1 0 0 1 0 0	rA	rA	rB	0 0 0 0 1 1 0 1 0 0 0
0	6	11	16	31

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

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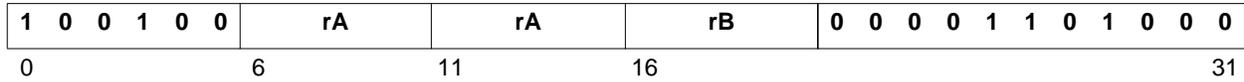
Registers Altered

- None

Latency

1 cycle

---

xor

Logical Exclusive OR

xor	rD, rA, rB		

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

Description

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

Pseudocode

$(rD) \leftarrow (rA) \oplus (rB)$

Registers Altered

- rD

Latency

1 cycle

xori

Logical Exclusive OR with Immediate

xori   rA, rD, IMM

1	0	1	0	1	0	rD	rA	IMM

0   6   11   16   31

Description

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

Pseudocode

(rD) ← (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.