MicroBlaze Processor Reference Guide

Embedded Development Kit

EDK (v3.1 EA) September 16, 2002

XILINX®

MicroBlaze Processor Reference Guide
EDK (v3.1 EA) September 16, 2002

The following table shows the revision history for this document.

	Version	Revision
09/16/02	1.0	Xilinx EDK (Embedded Processor Development Kit) release.

Table of Contents

Chapter 4: MicroBlaze Application Binary Interface

Chapter 5: 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/apps/appsweb.htm
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/partinfo/databook.htm
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://support.xilinx.com/support/troubleshoot/psolvers.htm

Resource	Description/URL
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

Summary

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

Overview

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

Features

The MicroBlaze embedded soft core includes the following features:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- Separate 32-bit instruction and data buses that conform to IBM's OPB (On-chip Peripheral Bus) specification
- Separate 32-bit instruction and data buses with direct connection to on-chip block RAM through a LMB (Local Memory Bus)
- 32-bit address bus
- Single issue pipeline

- Hardware multiplier (in Virtex-II and subsequent devices)

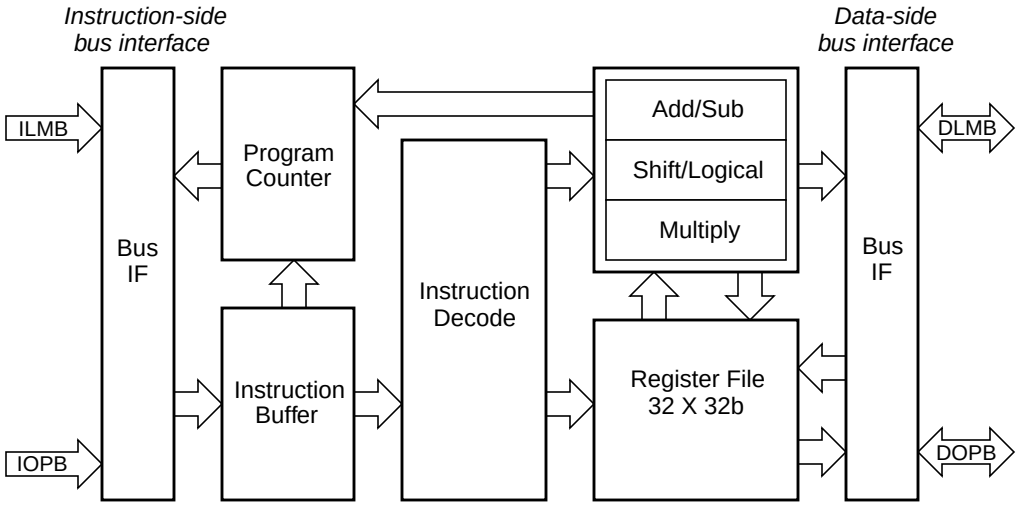


Figure 1-1: **MicroBlaze Core Block Diagram**

Instructions

All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit immediate operand (which can be extended to 32 bits by preceding the Type B instruction with an IMM instruction). Type B instructions have a single destination register operand. Instructions are provided in the following functional categories: arithmetic, logical, branch, load / store, and special. Table 1-2 lists the MicroBlaze instruction set. Refer to the MicroBlaze Instruction Set Architecture document for more information on these instructions. Table 1-1 describes the instruction set nomenclature used in the semantics of each instruction.

Table 1-1: **Instruction Set Nomenclature**

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,
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)

Table 1-2: **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
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	0000	0000.. Imm	Rd : = Ra >> Imm
BSRAI Rd,Ra,Imm	011001	Rd	Ra	0000	0100.. Imm	Rd := Ra[0], (Ra >> Imm)
BSLLI Rd,Ra,Imm	011001	Rd	Ra	0000	1000.. Imm	Rd := Ra << Imm
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	0000000000000001		Rd := Ra[0], (Ra >> 1); C := Ra[31]
SRC Rd,Ra	100100	Rd	Ra	0000000000100001		Rd := C, (Ra >> 1); C := Ra[31]
SRL Rd,Ra	100100	Rd	Ra	0000000001000001		Rd := 0, (Ra >> 1); C := Ra[31]

Table 1-2: **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**		
SEXT8 Rd,Ra	100100	Rd	Ra	0000000001100000		Rd[0:23] := Ra[24]; Rd[24:31] := Ra[24:31]
SEXT16 Rd,Ra	100100	Rd	Ra	0000000001100001		Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]
MTS Sd,Ra	100101	00000	Ra	110000000000000d		Sd := Ra , where S1 is MSR
MFS Rd,Sa	100101	Rd	00000	100000000000000a		Rd := Sa , where S0 is PC and S1 is MSR
BR Rb	100110	00000	00000	Rb	00000000000	PC := PC + Rb
BRD Rb	100110	00000	10000	Rb	00000000000	PC := PC + Rb
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 $\overline{s(Imm)}$
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
RTBD Ra,Imm	101101	10010	Ra	Imm		PC := Ra + s(Imm); MSR[BIP] := 0

Table 1-2: **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		
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
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

Table 1-2: **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		
LWI Rd,Ra,Imm	111010	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd := *Addr
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. R0 is defined to always have the value of zero. Anything written to R0 is discarded, and zero is always read.

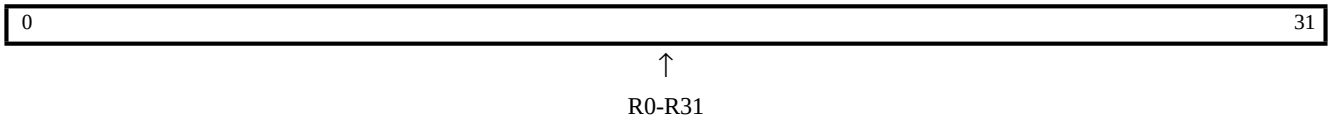


Figure 1-2: **R0-R31**

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

Bits	Name	Description	Reset Value
0:31	R0 through R31	**General Purpose Register** R0 through R31 are 32-bit general purpose registers. R0 is always zero.	0x00000000

Special Purpose Registers

Program Counter (PC)

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

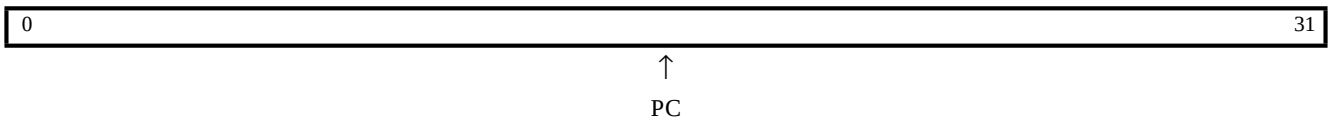


Figure 1-3: **PC**

Table 1-4: **Program Counter (PC)**

Bits	Name	Description	Reset Value
0:31	PC	**Program Counter** Address of next instruction to fetch	0x00000000

Machine Status Register (MSR)

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

Figure 1-4: **MSR**

Table 1-5: **Machine Status Register (MSR)**

Bits	Name	Description	Reset Value
0	CC	**Arithmetic Carry Copy** Copy of the Arithmetic Carry (bit 29). Read only.	0
1:27	Reserved		
28	BIP	**Break in Progress** 0 No Break in Progress 1 Break in Progress Source of break can be software break instruction or hardware break from Ext_Brk or Ext_NM_Brk pin.	0
29	C	**Arithmetic Carry** 0 No Carry (Borrow) 1 Carry (No Borrow)	0
30	IE	**Interrupt Enable** 0 Interrupts disabled 1 Interrupts enabled	0
31	BE	**Buslock Enable** 0 Buslock disabled on data-side OPB 1 Buslock enabled on data-side OPB Buslock Enable does not affect operation of ILMB, DLMB, or IOPB.	0

Pipeline

This section describes the MicroBlaze pipeline architecture.

Pipeline Architecture

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

- Fetch
- Decode
- Execute

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

cycle 1	cycle 2	cycle 3
Fetch	Decode	Execute

In the MicroBlaze parallel pipeline, each stage is active on each clock cycle. Three instructions can be executed simultaneously, one at each of the three pipeline stages. Even

though it takes three clock cycles for each instruction to complete, each pipeline stage can work on other instructions in parallel with and in advance of the instruction that is completing. Within one clock cycle, one new instruction is fetched, another is decoded, and a third is completed. The pipeline effectively completes one instruction per clock cycle.

	cycle 1	cycle 2	cycle 3	cycle4	cycle5
instruction 1	Fetch	Decode	Execute		
instruction 2		Fetch	Decode	Execute	
instruction 3			Fetch	Decode	Execute

Branches

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

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

Delay Slots

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

Load/Store Architecture

MicroBlaze can access memory in the following three data sizes:

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

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

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

- MSByte: Most Significant Byte
- LSByte: Least Significant Byte
- MSBit: Most Significant Bit
- LSBit: Least Significant Bit

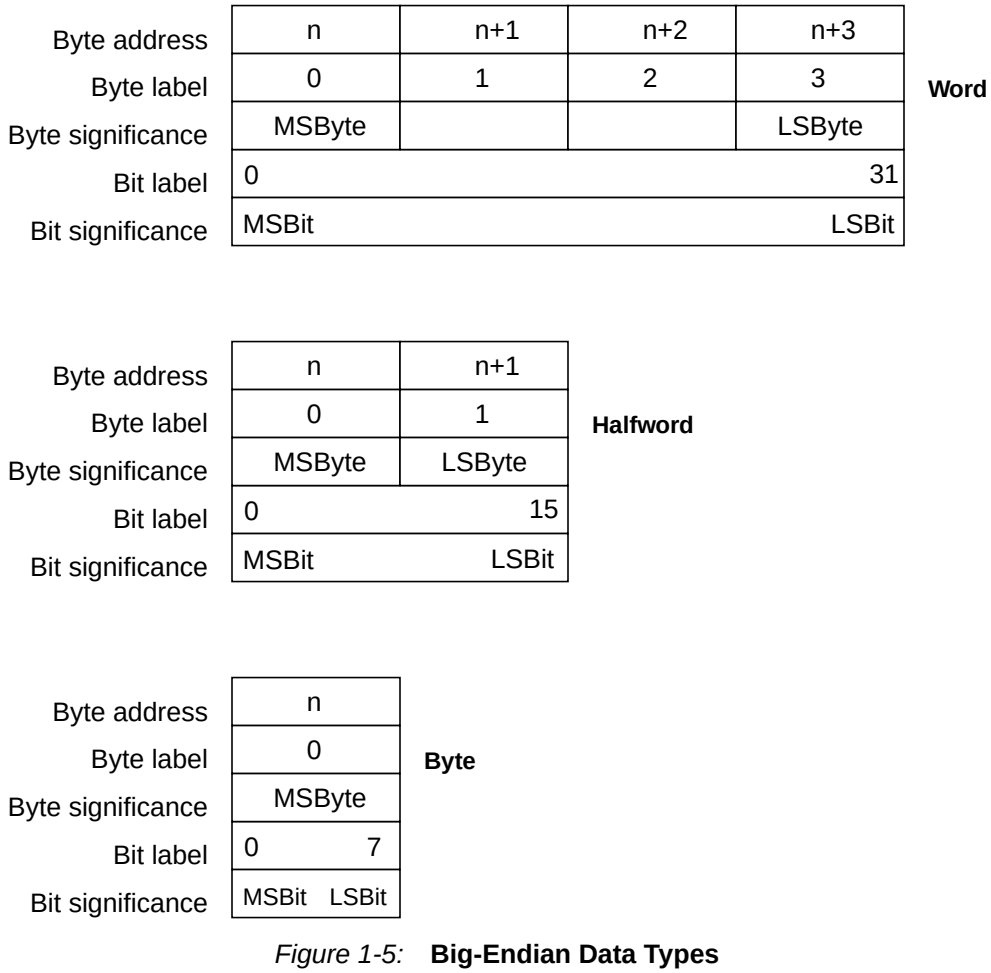


Figure 1-5: **Big-Endian Data Types**

Interrupts, Exceptions and Breaks

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

Interrupts

When an interrupt occurs, MicroBlaze stops the current execution to handle the interrupt request. MicroBlaze branches to address 0x00000010 and uses the General Purpose

Register 14 to store the address of the instruction that was to be executed when the interrupt occurred. It also disables future interrupts by clearing the Interrupt Enable flag in the Machine Status Register (setting bit 30 to 0 in MSR). The instruction located at the address where the current PC points to is not executed. Interrupts do not occur if the BIP bit in the MSR register is active (equal to 1).

Equivalent Pseudocode

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

Exceptions

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

Equivalent Pseudocode

```
r17 ← PC
PC ← 0x00000008
```

Breaks

There are two kinds of breaks:

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

Software Breaks

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

Hardware Breaks

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

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

Equivalent Pseudocode1

```
r16 ← PC
PC ← 0x00000018
```

```
MSR[IE] ← 1
```

Chapter 2

MicroBlaze Bus Interfaces

Summary

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

Overview

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses. Each bus interface unit is further split into a Local Memory Bus (LMB) and IBM's On-chip Peripheral Bus (OPB). The LMB provides single-cycle access to on-chip dual-port block RAM. The OPB interface provides a connection to both on-and off-chip peripherals and memory. .

Features

The MicroBlaze bus interfaces include the following features:

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

Bus Configurations

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

DOPB: Data interface, On-chip Peripheral Bus
DLMB: Data interface, Local Memory Bus (BRAM only)
IOPB: Instruction interface, On-chip Peripheral Bus
ILMB: Instruction interface, Local Memory Bus (BRAM only)
Core: Miscellaneous signals (Clock, Reset, Interrupt)

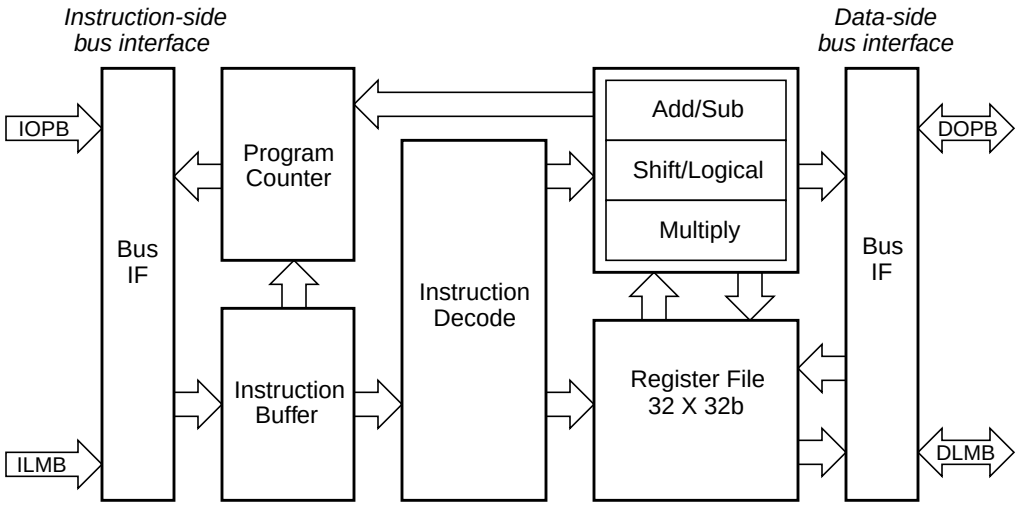


Figure 2-1: **MicroBlaze Core Block Diagram**

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

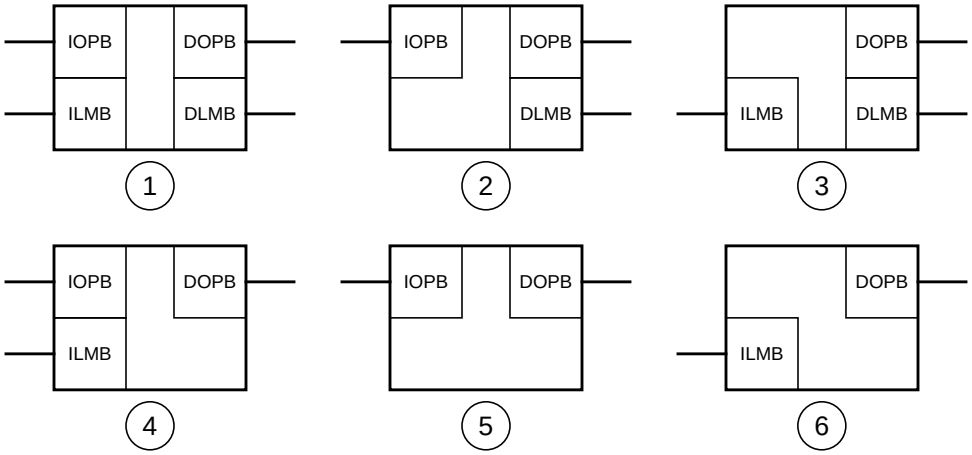


Figure 2-2: **MicroBlaze Bus Configurations**

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

Table 2-1: **MicroBlaze Bus Configurations**

	Configuration	Core Fmax	Debug available	Memory Models Supported
1	IOPB+ILMB+DOPB+DLMB	110	SW / JTAG	Large external instruction memory, Fast internal instruction memory (BRAM), Large external data memory, Fast internal data memory (BRAM)
2	IOPB+DOPB+DLMB	125	SW / JTAG	Large external instruction memory, Large external data memory, Fast internal data memory (BRAM)
3	ILMB+DOPB+DLMB	125	SW / JTAG	Fast internal instruction memory (BRAM), Large external data memory, Fast internal data memory (BRAM)
4	IOPB+ILMB+DOPB	110	JTAG for ILMB memory[1] SW / for IOPB memory	Large external instruction memory, Fast internal instruction memory (BRAM), Large external data memory,
5	IOPB+DOPB	125	SW / JTAG	Large external instruction memory, Large external data memory,
6	ILMB+DOPB	125	JTAG[1]	Fast internal instruction memory (BRAM), Large external data memory,

Note: ILMB memory can be debugged via a software resident monitor if the second port of the dual-ported ILMB BRAM is connected to an OPB BRAM memory controller. See Figure 2-6 and Figure 2-8.

Typical Peripheral Placement

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

Configuration 1

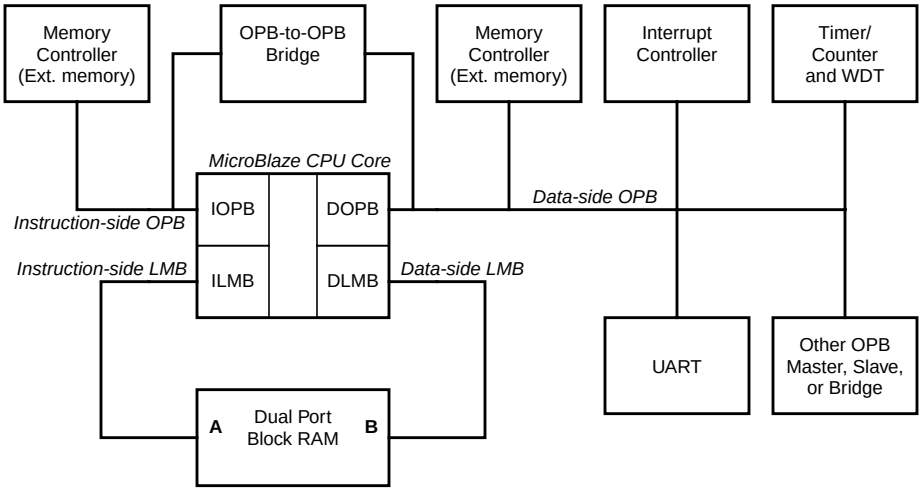


Figure 2-3: **Configuration 1: IOPB+ILMB+DOPB+DLMB**

Purpose

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

Typical Applications

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

Characteristics

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

Configuration 2

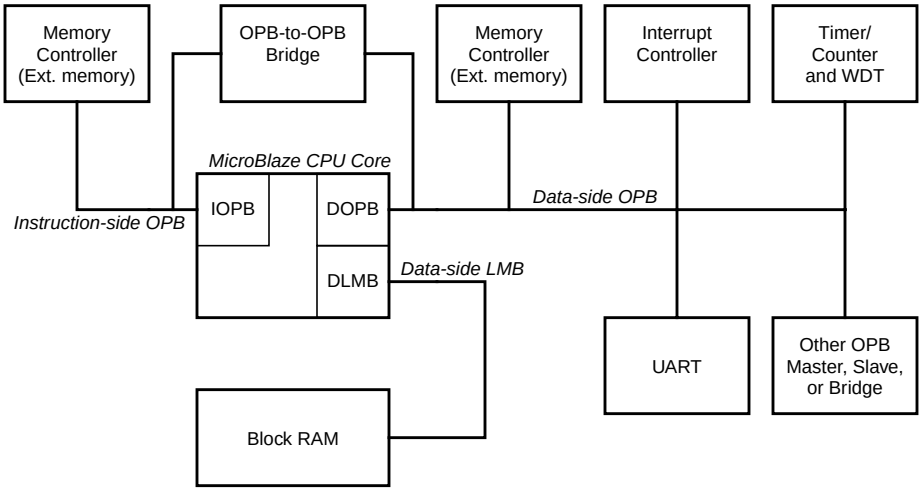


Figure 2-4: **Configuration 2: IOPB+DOPB+DLMB**

Purpose

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

Typical Applications

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

Characteristics

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

Configuration 3

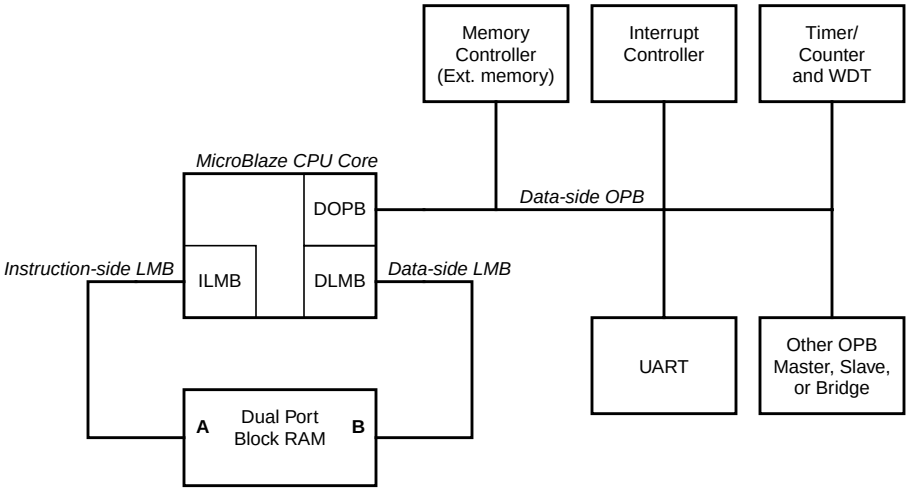


Figure 2-5: **Configuration 3: ILMB+DOPB+DLMB**

Purpose

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

Typical Applications

- Data-intensive controllers
- Small to medium state machines

Characteristics

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

Configuration 4

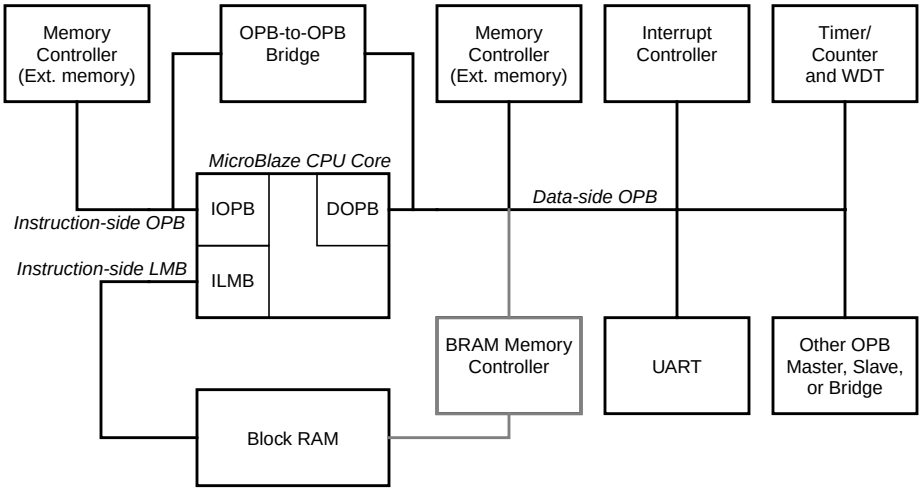


Figure 2-6: **Configuration 4: IOPB+ILMB+DOPB**

Purpose

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

Typical Applications

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

Characteristics

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

Configuration 5

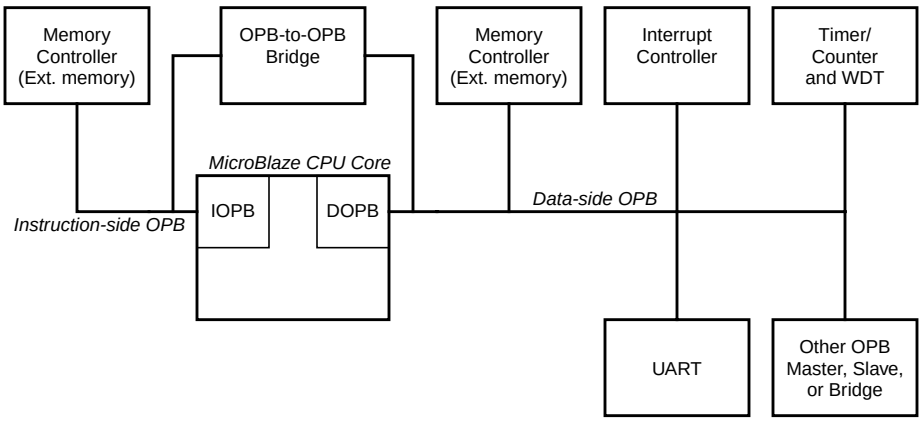


Figure 2-7: **Configuration 5: IOPB+DOPB**

Purpose

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

Typical Applications

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

Characteristics

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

Configuration 6

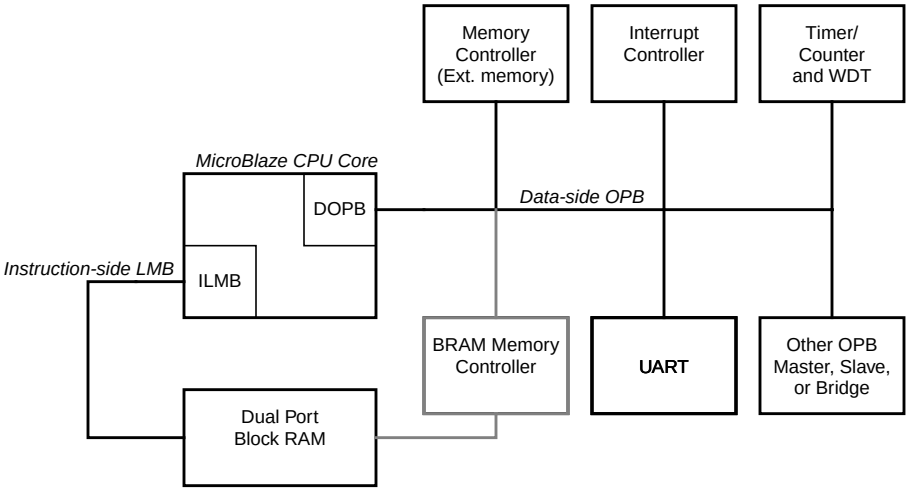


Figure 2-8: **Configuration 6: ILMB+DOPB**

Purpose

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

Typical Applications

- Minimal controllers
- Small to medium state machines

Characteristics

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

Bit and Byte Labeling

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

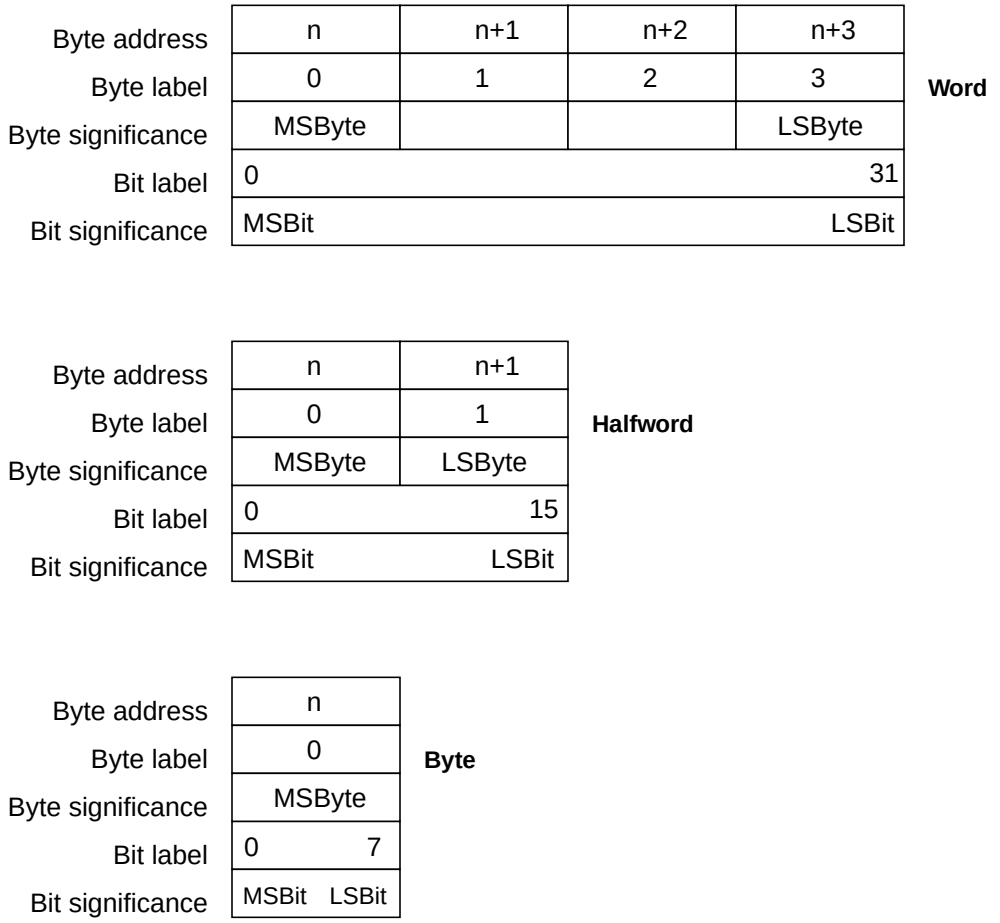


Figure 2-9: **MicroBlaze Big-Endian Data Types**

Core I/O

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

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

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

The core interfaces shown in the following table are defined as follows:

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)
Core: Miscellaneous signals

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

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

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

Signal	Interface	I/O	Description	Page
Data_Addr[0:31]	DLMB	O	Data interface LB address bus	37
Byte_Enable[0:3]	DLMB	O	Data interface LB byte enables	37
Data_Write[0:31]	DLMB	O	Data interface LB write data bus	38
D_AS	DLMB	O	Data interface LB address strobe	38
Read_Strobe	DLMB	O	Data interface LB read strobe	38
Write_Strobe	DLMB	O	Data interface LB write strobe	38
Data_Read[0:31]	DLMB	I	Data interface LB read data bus	38
DReady	DLMB	I	Data interface LB data ready	38
Instr_Addr[0:31]	ILMB	O	Instruction interface LB address bus	37
I_AS	ILMB	O	Instruction interface LB address strobe	38
IFetch	ILMB	O	Instruction interface LB instruction fetch	38
Instr[0:31]	ILMB	I	Instruction interface LB read data bus	38
IReady	ILMB	I	Instruction interface LB data ready	38
Interrupt	Core	I	Interrupt	
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	

Bus Organization

OPB Bus Configuration

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

The OPB data bus interconnects are illustrated in Figure 2-10. 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 2-11. 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.

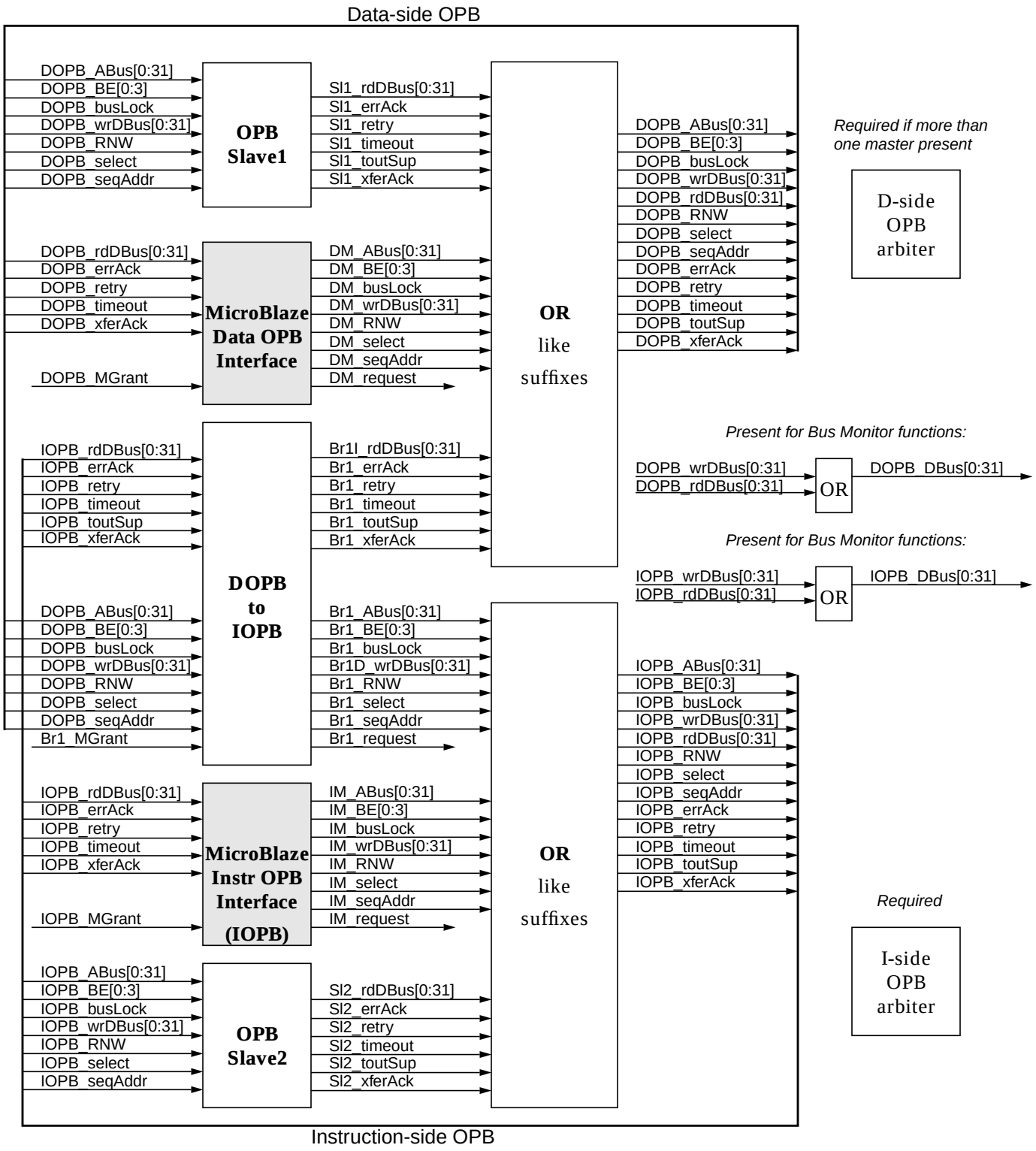


Figure 2-10: **OPB Interconnection (breaking up read and write buses)**

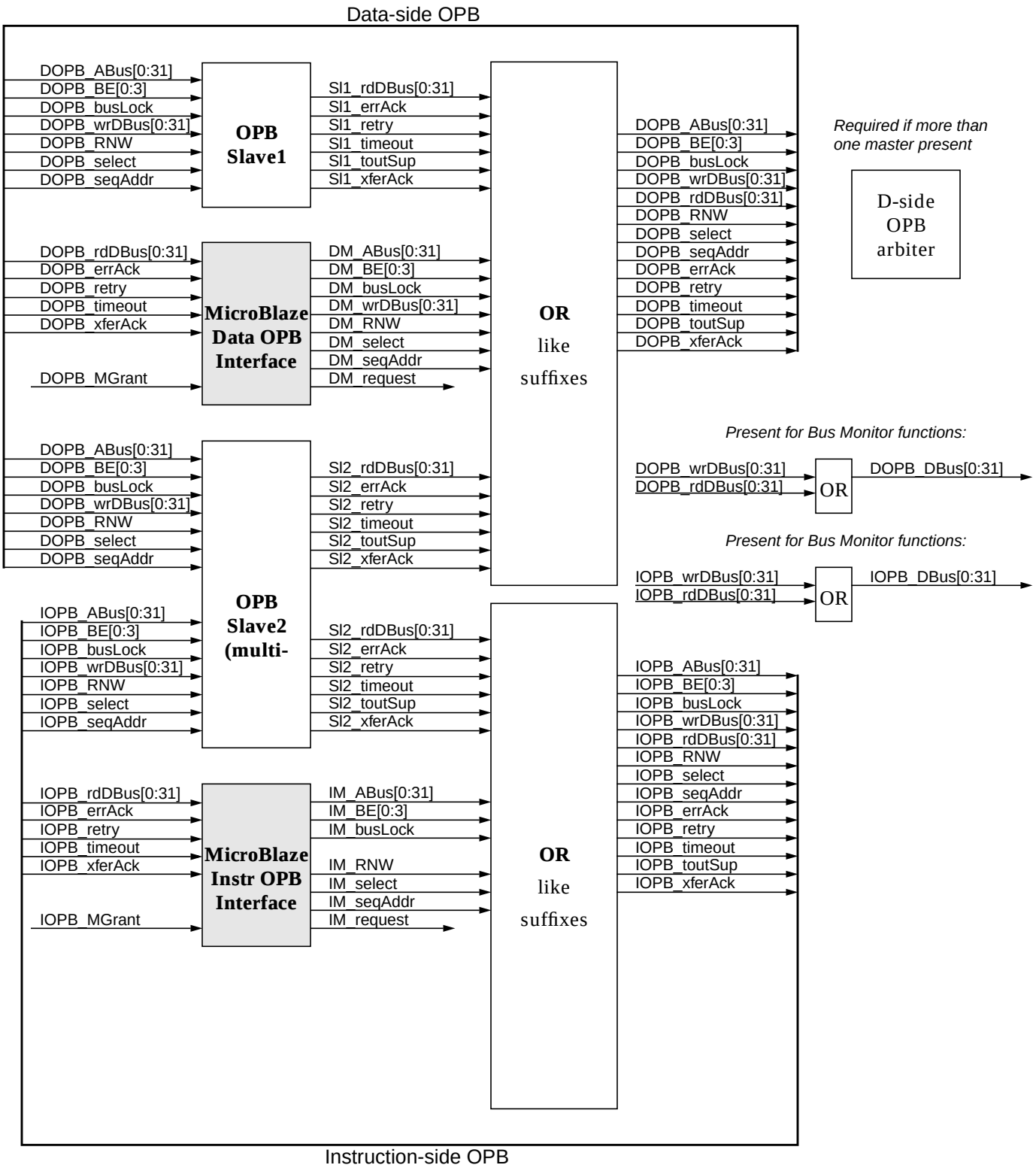


Figure 2-11: **OPB Interconnection (with multi-ported slave and no bridge)**

LMB Bus Definition

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

Table 2-3: **LMB Bus Signals**

Signal	Data Interface	Instr. 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 2-4: **Valid Values for Byte_Enable[0:3]**

	Byte Lanes Used			
Byte_Enable[0:3]	**Data[0:7]**	**Data[8:15]**	**Data[16:23]**	**Data[24:31]**
0000				
0001				x
0010			x	
0100		x		
1000	x			

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

	Byte Lanes Used			
Byte_Enable[0:3]	**Data[0:7]**	**Data[8:15]**	**Data[16:23]**	**Data[24:31]**
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 Bus Operations

The following diagrams provide examples of LMB bus operations.

Generic Write Operation

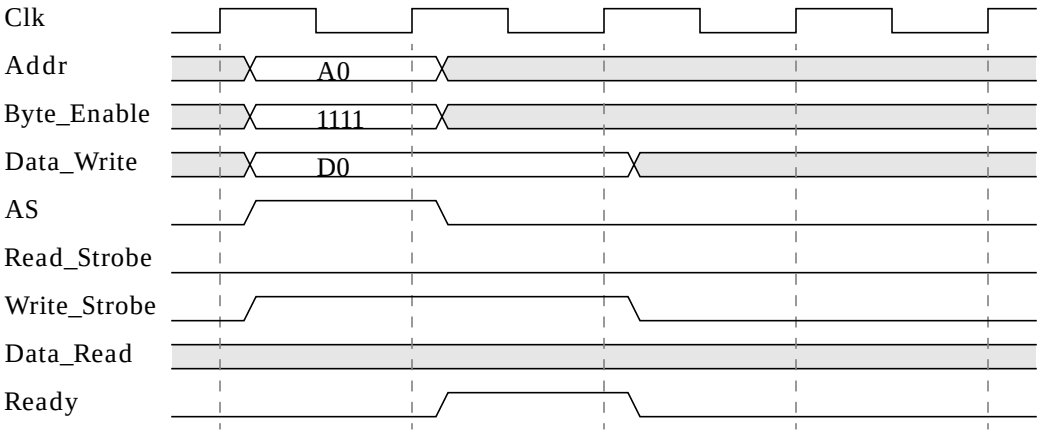


Figure 2-12: **LMB Generic Write Operation**

Generic Read Operation

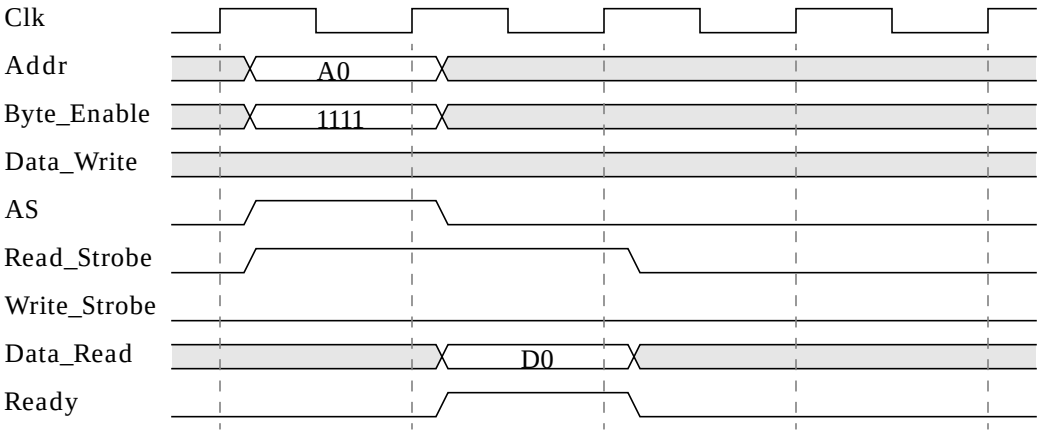


Figure 2-13: **LMB Generic Read Operation**

Back-to-Back Write Operation (Typical LMB access - 2 clocks per write)

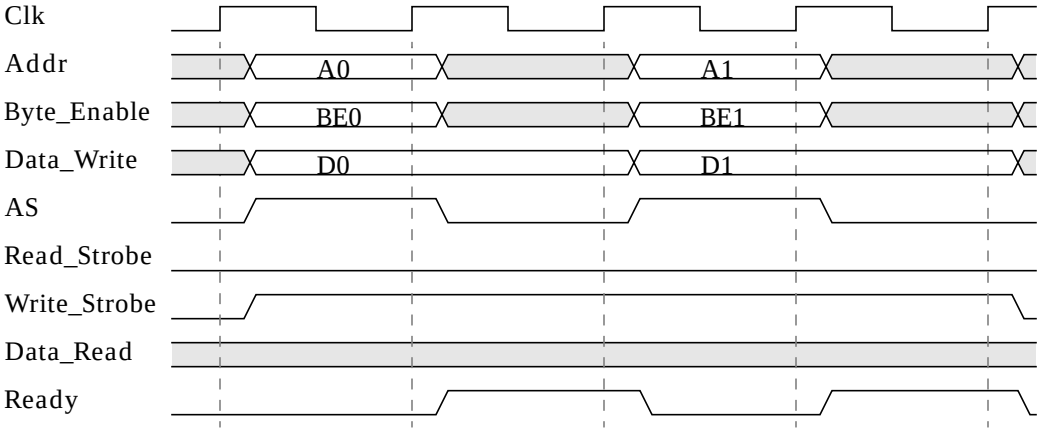


Figure 2-14: **LMB Back-to-Back Write Operation**

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

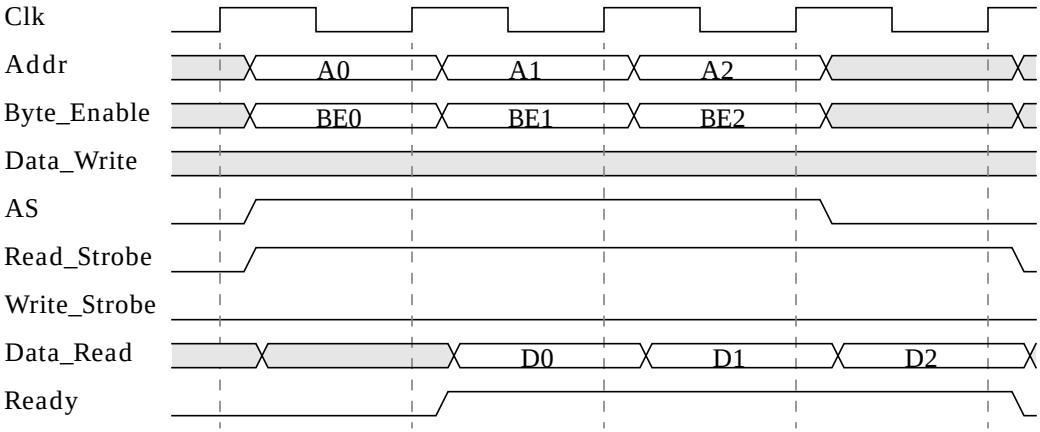


Figure 2-15: **LMB Single Cycle Back-to-Back Read Operation**

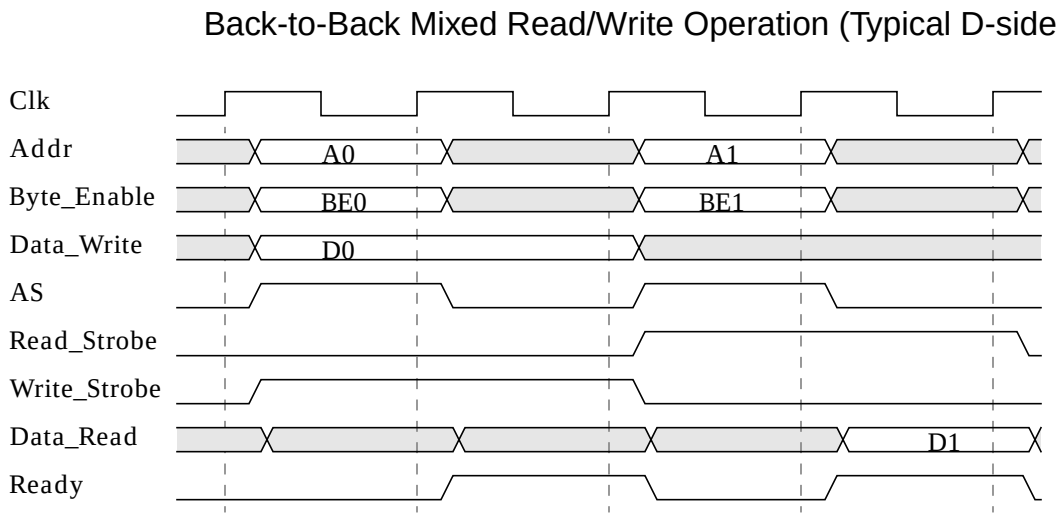


Figure 2-16: **Back-to-Back Mixed Read/Write Operation**

Read and Write Data Steering

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

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

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

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

			Register rD Data			
Address[30:31]	**Byte_Enable [0:3]**	**Transfer Size**	**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 2-6: **Write Data Steering (store from Register rD)**

			Write Data Bus Bytes			
Address[30:31]	**Byte_Enable [0:3]**	**Transfer Size**	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.

Implementation

Parameterization

The following characteristics of MicroBlaze can be parameterized:

- Data Interface options: OPB only, LMB+OPB
- Instruction Interface options: LMB only, LMB+OPB, OPB only
- Barrel shifter

Table 2-7: **MPD Parameters**

Feature/Description	Parameter Name	Allowable Values	Default Value	VHDL Type
Target Family	C_FAMILY	Xilinx FPGA families	virtex2	string
Data Size	C_DATA_SIZE	32	32	integer
Instance Name	C_INSTANCE	Any instance name	microblaze	string
Data side OPB interface	C_D_OPB	0, 1	1	integer
Data side LMB interface	C_D_LMB	0, 1	1	integer
Instruction side OPB interface	C_I_OPB	0, 1	1	integer
Instruction side LMB interface	C_I_LMB	0, 1	1	integer
Barrel Shifter	C_USE_BARREL	0, 1	0	integer

Chapter 3

MicroBlaze Endianness

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

Origin of Endian

The terms *Big-Endian* and *Little-Endian* come from Part I, Chapter 4, of Jonathan Swift's *Gulliver's Travels*. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

. . . our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of Lilliput and Blefuscu. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of Blefuscu; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the Big-Endians have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of Blefuscu did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet Lustrog, in the fifty-fourth Chapter of the Brundrecal, (which is their Alcoran.) This, however, is thought to be a mere Strain upon the text: For the Words are these; That all true Believers shall break their Eggs at the convenient End: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the Big-Endian Exiles have found so much Credit in the Emperor of Blefuscu's Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody

War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

Definitions

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

Note Endianness does not affect single byte data.

Bit Naming Conventions

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

Data Types and Endianness

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

Table 3-1: **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 3-2: **Half Word Data Type**

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

Table 3-2: **Half Word Data Type**

Bit label	0	15
Bit significance	MSBit	LSBit

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

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

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

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

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

Note For the MicroBlaze core, all operands in the ALU and GPRs, and all pipeline instructions are big-endian.

The big-endian mapping of “struct” is shown in the following table. (The data is highlighted in the structure mappings). Hexadecimal addresses are below the data stored at the address. The contents of each byte, as defined in the structure, are shown as a number (hexadecimal) or character (for the string elements).

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

11 0x00	12 0x01	13 0x02	14 0x03	0x04	0x05	0x06	0x07
21 0x08	22 0x09	23 0x0A	24 0x0B	25 0x0C	26 0x0D	27 0x0E	28 0x0F
31 0x10	32 0x11	33 0x12	34 0x13	‘A’ 0x14	‘B’ 0x15	‘C’ 0x16	‘D’ 0x17
‘E’ 0x18	‘F’ 0x19	‘G’ 0x1A	0x1B	51 0x1C	52 0x1D	0x1E	0x1F
61 0x20	62 0x21	63 0x22	64 0x23	0x24	0x25	0x26	0x27

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

14 0x00	13 0x01	12 0x02	11 0x03	0x04	0x05	0x06	0x07
28 0x08	27 0x09	26 0x0A	25 0x0B	24 0x0C	23 0x0D	22 0x0E	21 0x0F
34 0x10	33 0x11	32 0x12	31 0x13	‘A’ 0x14	‘B’ 0x15	‘C’ 0x16	‘D’ 0x17
‘E’ 0x18	‘F’ 0x19	‘G’ 0x1A	0x1B	52 0x1C	51 0x1D	0x1E	0x1F
64 0x20	63 0x21	62 0x22	61 0x23	0x24	0x25	0x26	0x27

VHDL Example

BRAM – LMB Example

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

Interface Between BRAM and MicroBlaze

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

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

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

end Local_Memory;

architecture IMP of Local_Memory is
```

BRAM Component Declaration (little-endian)

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

Swap BRAM Little-endian Data to Big-endian

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

BRAM Instantiation

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

BRAM – OPB Example

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

Interface Between BRAM and MicroBlaze

```
library IEEE;
use IEEE.std_logic_1164.all;

entity OPB_BRAM is
  generic (
    C_BASEADDR : std_logic_vector(0 to 31) := X"B000_0000";
    C_NO_BRAMS : natural      := 4;  -- Can be 4,8,16,32 only
    C_VIRTEXII    : boolean   := true
    );
  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;

    -- OPB signals
    OPB_ABus    : in std_logic_vector(0 to 31);
    OPB_BE      : in std_logic_vector(0 to 3);
    OPB_RNW     : in std_logic;
    OPB_select  : in std_logic;
    OPB_seqAddr : in std_logic;
    OPB_DBus    : in std_logic_vector(0 to 31);

    OPB_BRAM_DBus    : out std_logic_vector(0 to 31);
    OPB_BRAM_errAck  : out std_logic;
    OPB_BRAM_retry   : out std_logic;
    OPB_BRAM_toutSup : out std_logic;
    OPB_BRAM_xferAck : out std_logic;

    -- OPB_BRAM signals (other port)
    BRAM_Clk        : in  std_logic;
    BRAM_Addr       : in  std_logic_vector(0 to 31);
    BRAM_WE         : in  std_logic_vector(0 to 3);
    BRAM_Write_Data : in  std_logic_vector(0 to 31);
    BRAM_Read_Data  : out std_logic_vector(0 to 31)
    );

end entity OPB_BRAM;
```

```
architecture IMP of OPB_BRAM is
```

BRAM Component Declaration (little-endian)

```
component RAMB16_S9_S9
    port (
      DIA   : in  std_logic_vector (7 downto 0);
      DIB   : in  std_logic_vector (7 downto 0);
      DIPA  : in  std_logic_vector (0 downto 0);
      DIPB  : in  std_logic_vector (0 downto 0);
      ENA   : in  std_ulogic;
      ENB   : in  std_ulogic;
      WEA   : in  std_ulogic;
      WEB  : in  std_ulogic;
      SSRA  : in  std_ulogic;
      SSRB  : in  std_ulogic;
      CLKA  : in  std_ulogic;
      CLKB  : in  std_ulogic;
      ADDRA : in  std_logic_vector (10 downto 0);
      ADDRB : in  std_logic_vector (10 downto 0);
      DOA   : out std_logic_vector (7 downto 0);
      DOB   : out std_logic_vector (7 downto 0);
      DOPA  : out std_logic_vector (0 downto 0);
      DOPB  : out std_logic_vector (0 downto 0) );
  end component;
Swap BRAM Little-endian Data to Big-endian
  BE_to_LE : for I in 0 to 31 generate
    opb_dbus_le(I )       <= OPB_DBus(31-I);
    bram_write_data_le(I) <= BRAM_Write_Data(31-I);
    BRAM_Read_Data(I)    <= bram_Read_Data_LE(31-I);
    opb_ABus_LE(I)        <= OPB_ABus(31-I);
    bram_Addr_LE(I)       <= BRAM_Addr(31-I);
  end generate BE_to_LE;
```

BRAM Instantiation

```
All_Brams : for I in 0 to C_NO_BRAMS-1 generate

By_8 : if (C_NO_BRAMS = 4) generate

RAMB16_S9_S9_I : RAMB16_S9_S9
port map (
DIA => opb_DBUS_LE(((I+1)*8-1) downto I*8), --[in std_logic_vector(7
downto 0)]
DIB =>bram_Write_Data_LE(((I+1)*8)-1 downto I*8), --[in
std_logic_vector (downto 0)]
DIPA => null_1,            -- [in  std_logic_vector (7 downto 0)]
DIPB  => null_1,           -- [in  std_logic_vector (7 downto 0)]
ENA   => '1',               -- [in  std_ulogic]
ENB   => '1',               -- [in  std_ulogic]
WEA   => opb_WE(I),         -- [in  std_ulogic]
WEB   => BRAM_WE(I),        -- [in  std_ulogic]
SSRA  => '0',               -- [in  std_ulogic]
SSRB  => '0',               -- [in  std_ulogic]
CLKA  => OPB_Clk,           -- [in  std_ulogic]
CLKB  => BRAM_Clk,         -- [in  std_ulogic]
ADDRA => opb_ABus_LE(12 downto 2),  -- [in  std_logic_vector (10 downto
0)]
```

```
ADDRB => bram_Addr_LE(12 downto 2),  -- [in  std_logic_vector (10 downto
0)]
DOA=>opb_BRAM_DBus_LE_I(((I+1)*8-1)downto I*8),--[out
std_logic_vector(7 downto 0)]
DOB =>bram_Read_Data_LE(((I+1)*8-1) downto I*8),--[out
std_logic_vector(7 downto 0)]
DOPA  => open,              -- [out std_logic_vector (0 downto 0)]
DOPB  => open);             -- [out std_logic_vector (0 downto 0)]
end generate By_8;
```

Chapter 4

MicroBlaze Application Binary Interface

Scope

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

Data Types

The data types used by MicroBlaze assembly programs are shown in Table 4-1. Data types such as data8, data16, and data32 are used in place of the usual byte, halfword, and word.

Table 4-1: **Data types in MicroBlaze assembly programs**

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 4-2.

Table 4-2: **Register usage conventions**

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

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

- The volatile registers are used as temporaries and do not retain values across the function calls. Registers R3 through R12 are volatile, of which R3 and R4 are used for returning values to the caller function, if any. Registers R5 through R10 are used for passing parameters between sub-routines.
- Registers R19 through R31 retain their contents across function calls and are hence termed as non-volatile registers. The callee function is expected to save those non-volatile registers, which are being used. These are typically saved to the stack during the prologue and then reloaded during the epilogue.
- Certain registers are used as dedicated registers and programmers are not expected to use them for any other purpose.
  - Registers R14 through R17 are used for storing 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 4-1

The shaded area in Figure 4-1 denotes a part of the caller function's stack frame, while the unshaded area indicates the callee function's frame. The ABI conventions of the stack frame define the protocol for passing parameters, preserving non-volatile register values and allocating space for the local variables in a function. Functions which contain calls to other sub-routines are called as non-leaf functions, These non-leaf functions have to create a new stack frame area for its own use. When the program starts executing, the stack pointer will have the maximum value. As functions are called, the stack pointer is decremented by the number of words required by every function for its stack frame. The stack pointer of a caller function will always have a higher value as compared to the callee function.

Figure 4-1: **Stack Convention**

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)
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 4-2. After the call from Func 1 to Func 2, the value of the stack pointer (SP) is decremented. This value of SP is again decremented to accommodate the stack frame for Func3. On return from Func 3 the value of the stack pointer is increased to its original value in the function, Func 2.

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

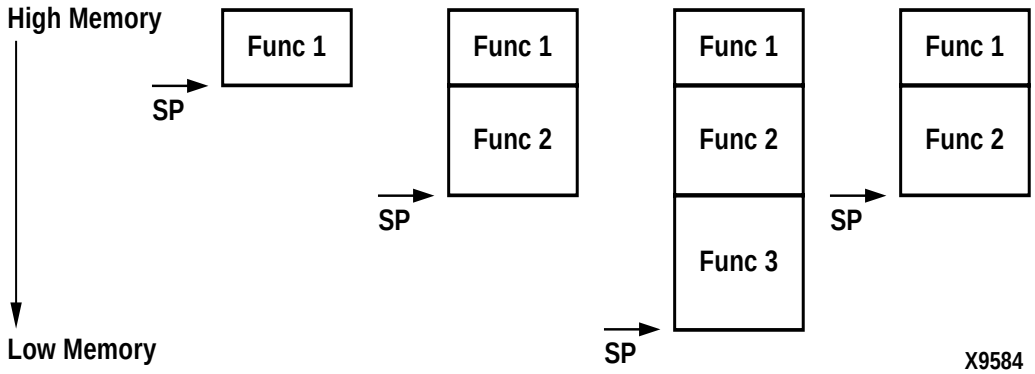


Figure 4-2: **Stack Frame**

Calling Convention

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

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

Memory Model

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

Small data area

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

Data area

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

Common un-initialized area

Un-initialized global variables are allocated to the comm area and can be accessed either using the absolute address or using the read-write small data area anchor R13.

Literals or constants

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

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

Interrupt and Exception Handling

MicroBlaze assumes certain address locations for handling interrupts and exceptions as indicated in Table 4-3. When the device is powered ON or on a reset, execution starts at 0x0. If an exception occurs, MicroBlaze jumps to address location 0x8, while in case of an interrupt, the control is passed to address location 0x10. At these locations, code is written to jump to the appropriate handlers.

Table 4-3: **Interrupt and Exception Handling**

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

The code expected at these locations is as shown in Figure 4-3. In case of programs compiled without the **-xl-mode-xmdstub** compiler option, the **crt0.o** initialization file is passed by the mb-gcc compiler to the **mb-ld** linker for linking. This file sets the appropriate addresses of the exception handlers.

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

Figure 4-3: **Code for passing control to exception and interrupt handlers**

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

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

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

Chapter 5

MicroBlaze Instruction Set Architecture

Summary

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

Notation

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

Table 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
r*x*	Register *x*
x[*i*]	Bit *i* in register *x*
x[*i*:*j*]	Bits *i* through *j* in register *x*
=	Equal comparison
≠	Not equal comparison
>	Greater than comparison
>=	Greater than or equal comparison
<	Less than comparison

Table 1: Symbol notation

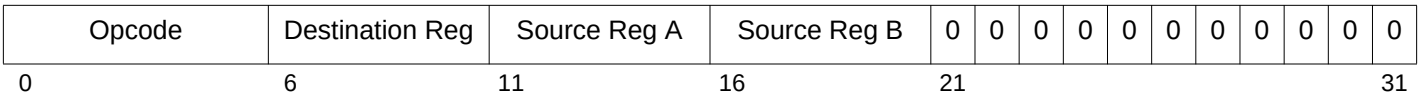
Symbol	Meaning
<=	Less than or equal comparison
sext(*x*)	Sign-extend *x*
Mem(*x*)	Memory location at address *x*

Formats

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

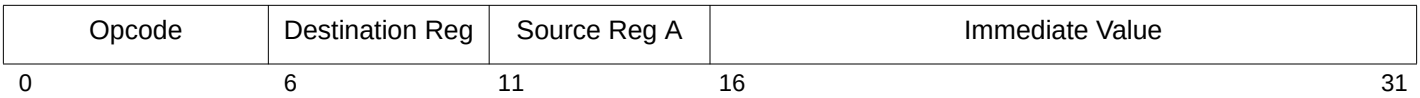
Type A

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



Type B

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



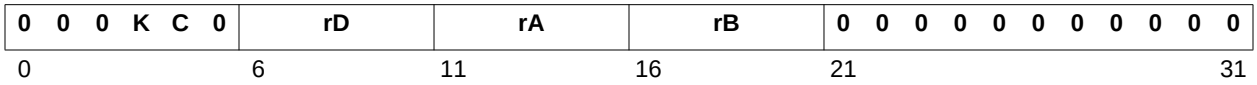
Instructions

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

add

Arithmetic Add

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



Description

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

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

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

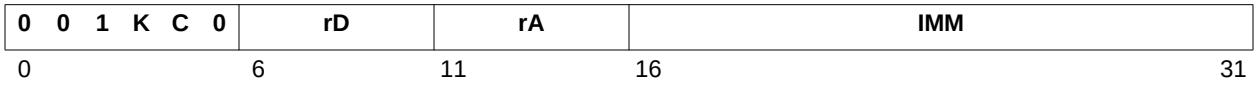
Note

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

addi

Arithmetic Add Immediate

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



Description

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

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

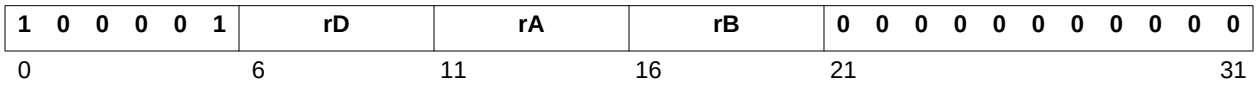
The C bit in the instruction opcode is not the same as the carry bit in the MSR register.

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

and

Logical AND

and rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- rD

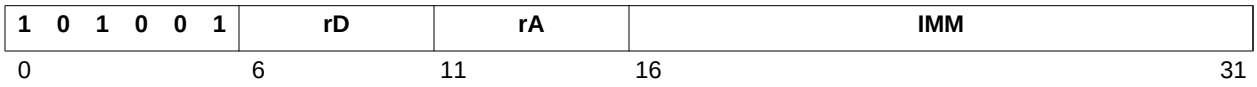
Latency

1 cycle

andi

Logial AND with Immediate

andi rD, rA, IMM



Description

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

Pseudocode

```
(rD) ← (rA) ∧ sext(IMM)
```

Registers Altered

- rD

Latency

1 cycle

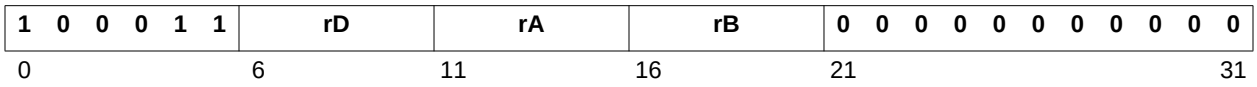
Note

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

andn

Logical AND NOT

andn rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- rD

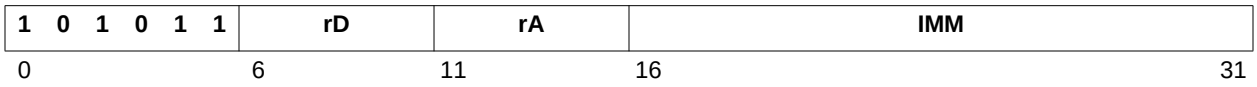
Latency

1 cycle

andni

Logical AND NOT with Immediate

andni rD, rA, IMM



Description

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

Pseudocode

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

Registers Altered

- rD

Latency

1 cycle

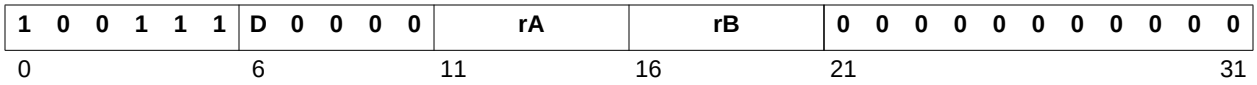
Note

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

beq

Branch if Equal

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

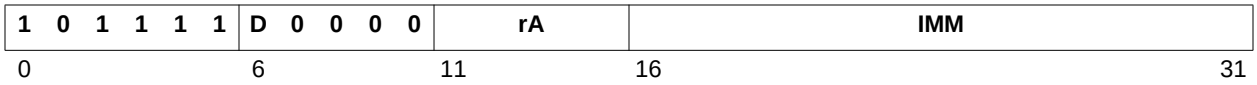
2 cycles (if branch is taken and the D bit is set)

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

beqi

Branch Immediate if Equal

beqi	rA, IMM	Branch Immediate if Equal
beqid	rA, IMM	Branch Immediate if Equal with Delay



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

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

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

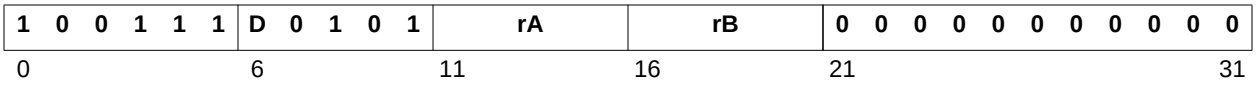
Note

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

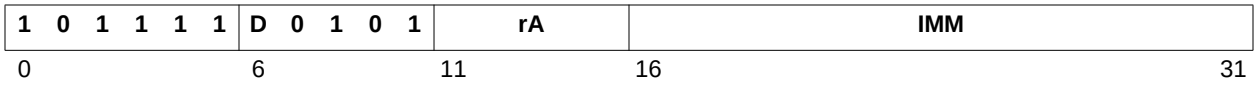
2 cycles (if branch is taken and the D bit is set)

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

bgei

Branch Immediate if Greater or Equal

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

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

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

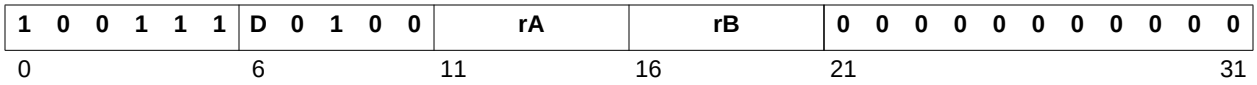
Note

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

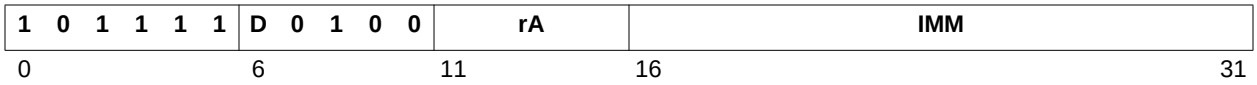
2 cycles (if branch is taken and the D bit is set)

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

bgti

Branch Immediate if Greater Than

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

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

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

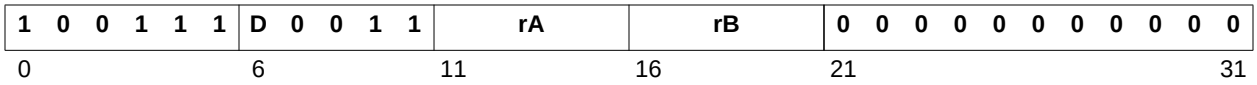
Note

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

ble

Branch if Less or Equal

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

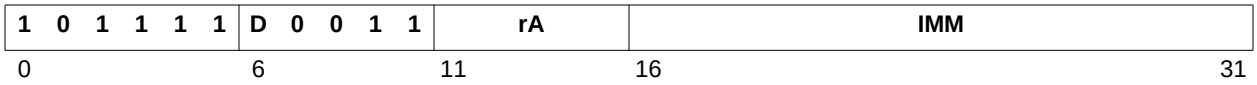
2 cycles (if branch is taken and the D bit is set)

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

blei

Branch Immediate if Less or Equal

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

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

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

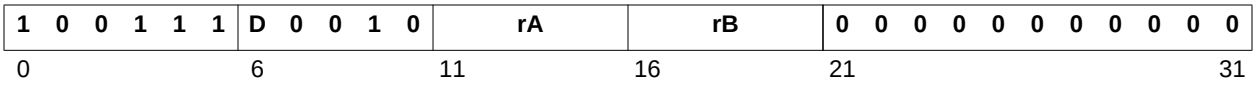
Note

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

blt

Branch if Less Than

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

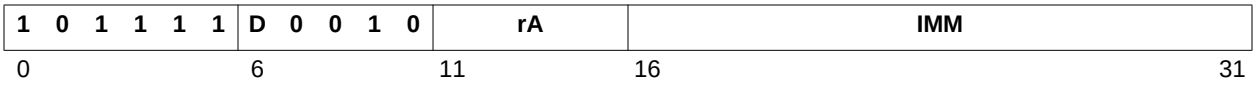
2 cycles (if branch is taken and the D bit is set)

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

blti

Branch Immediate if Less Than

blti	rA, IMM	Branch Immediate if Less Than
bltid	rA, IMM	Branch Immediate if Less Than with Delay



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

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

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

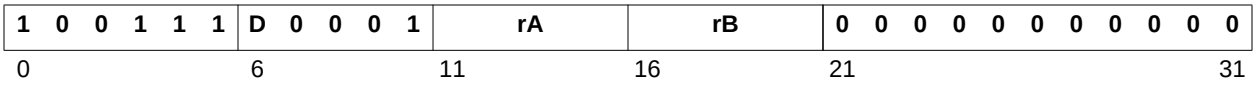
Note

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

bne

Branch if Not Equal

bne	rA, rB	Branch if Not Equal
bned	rA, rB	Branch if Not Equal with Delay



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

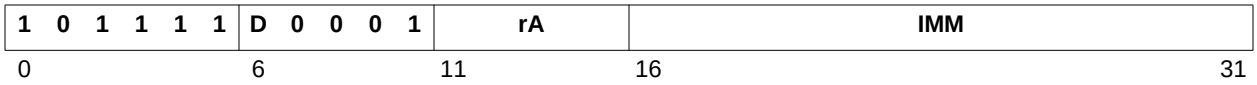
2 cycles (if branch is taken and the D bit is set)

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

bnei

Branch Immediate if Not Equal

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



Description

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

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

Pseudocode

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

Registers Altered

- PC

Latency

1 cycle (if branch is not taken)

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

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

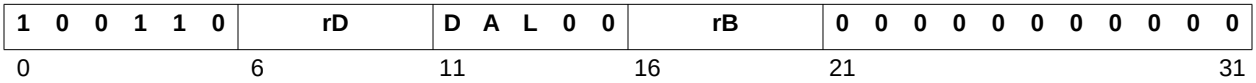
Note

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



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)

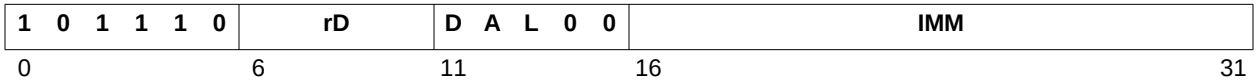
Note

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



Description

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

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

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

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

Pseudocode

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

Registers Altered

- rD
- PC

Latency

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

Notes

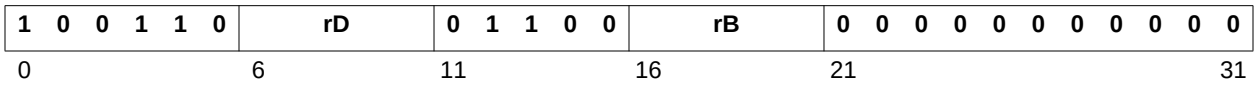
The instructions brli and brali are not available.

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



Description

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

Pseudocode

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

Registers Altered

- rD
- PC
- MSR[BIP]

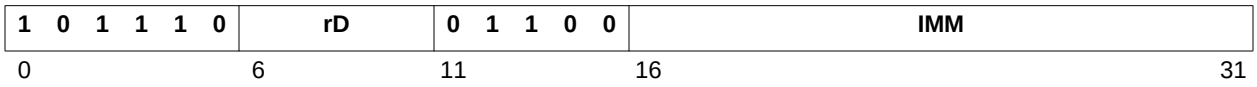
Latency

3 cycles

brki

Break Immediate

brki rD, IMM



Description

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

Pseudocode

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

Registers Altered

- rD
- PC
- MSR[BIP]

Latency

3 cycles

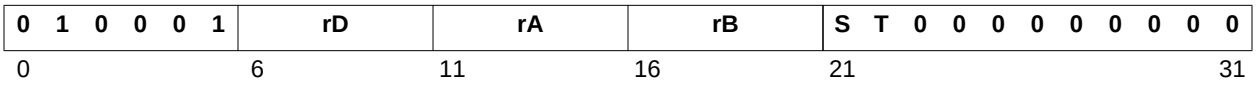
Note

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



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 perfomed 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
    (rD)[0] ← (rA)[0]
    (rD)[1:31] ← (rA) >> (rB)[27:31]
  else
    (rD) ← (rA) >> (rB)[27:31]
```

Registers Altered

- rD

Latency

2 cycles

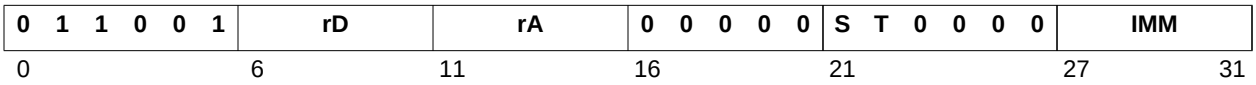
Note

These instructions are optional.

bsi

Barrel Shift Immediate

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



Description

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

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

The mnemonic bsra will set the T bit (Type bit). If the T bit is set, the barrel shift perfomed 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
    (rD)[0] ← (rA)[0]
    (rD)[1:31] ← (rA) >> IMM
  else
    (rD) ← (rA) >> IMM
```

Registers Altered

- rD

Latency

2 cycles

Notes

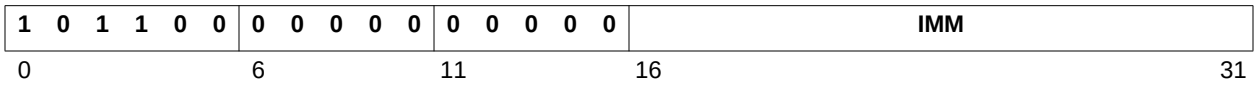
These are not Type B Instructions. There is no effect from a preceeding imm instruction.

These instructions are optional.

imm

Immediate

imm IMM



Description

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

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

Latency

1 cycle

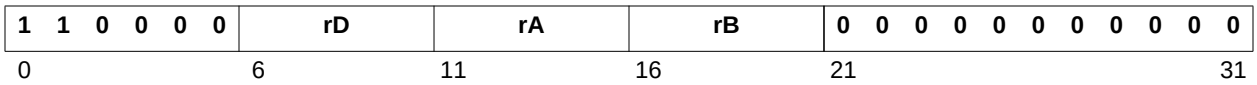
Note

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

lbu

Load Byte Unsigned

lbu rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- rD

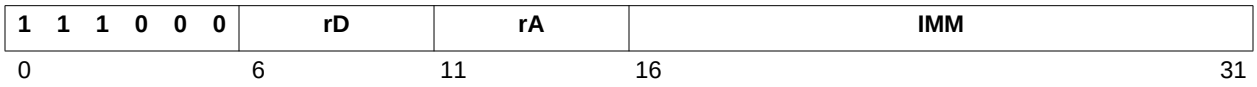
Latency

2 cycles

lbui

Load Byte Unsigned Immediate

lbui rD, rA, IMM



Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

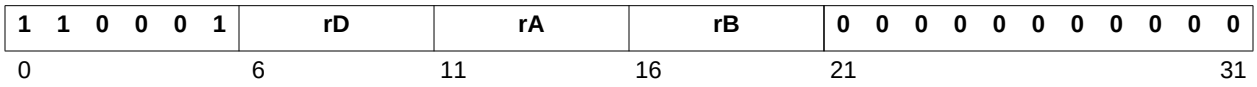
Note

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

lhu

Load Halfword Unsigned

lhu rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- rD

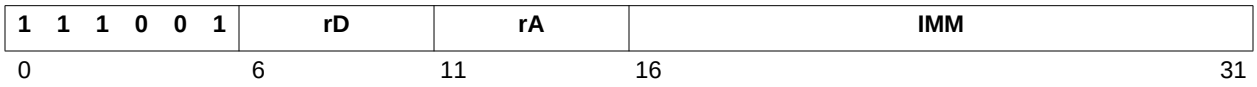
Latency

2 cycles

lhui

Load Halfword Unsigned Immediate

lhui rD, rA, IMM



Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

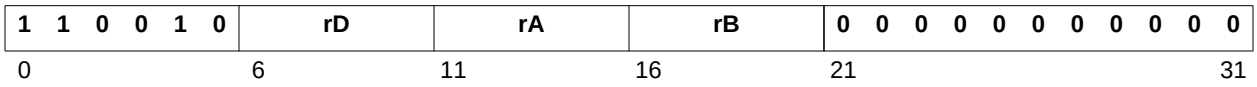
Note

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

lw

Load Word

lw rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- rD

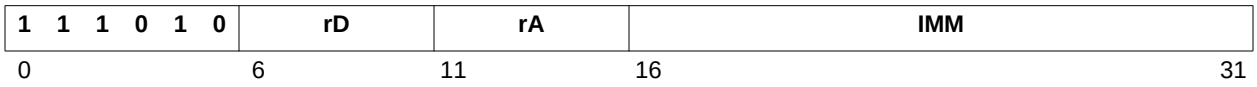
Latency

2 cycles

lwi

Load Word Immediate

lwi rD, rA, IMM



Description

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

Pseudocode

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

Registers Altered

- rD

Latency

2 cycles

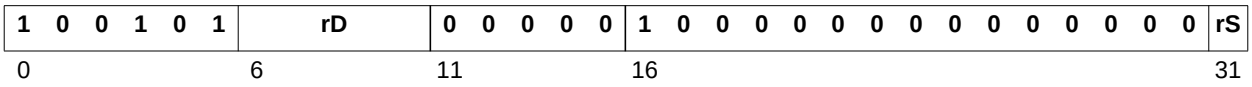
Note

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

mfs

Move From Special Purpose Register

mfs rD, rS



Description

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

Pseudocode

```
(rD) ← (rS)
```

Registers Altered

- rD

Latency

1 cycle

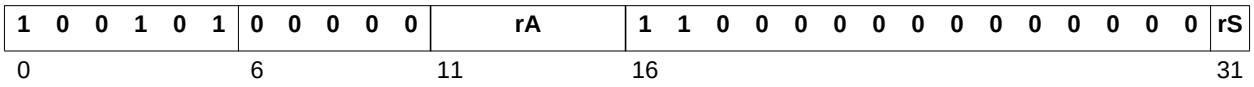
Note

To refer to special purpose registers in assembly language, use rpc for PC and rmsr for MSR.

mts

Move To Special Purpose Register

mts rS, rA



Description

Copies the contents of register rD into the MSR register.

Pseudocode

```
(rS) ← (rA)
```

Registers Altered

- rS

Latency

1 cycle

Notes

You cannot write to the PC using the MTS instruction.

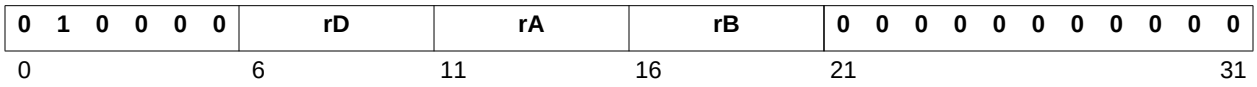
When writing to MSR using MTS, the value written will take effect one clock cycle after executing the MTS instruction.

To refer to special purpose registers in assembly language, use rpc for PC and rmsr for MSR.

mul

Multiply

mul rD, rA, rB



Description

Multiplies the contents of registers rA and rB and puts the result in register rD. This is a 32-bit by 32-bit multiplication that will produce a 64-bit result. The least significant word of this value is placed in rD.

Pseudocode

(rD) ← (rA) × (rB)

Registers Altered

- rD

Latency

3 cycles

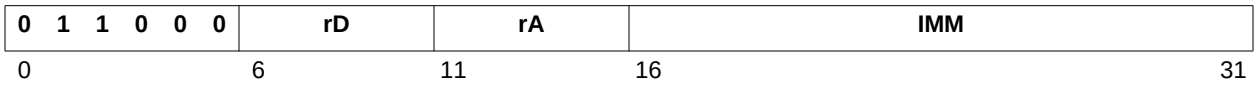
Note

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

muli

Multiply Immediate

muli rD, rA, IMM



Description

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

Pseudocode

```
(rD) ← (rA) × sext(IMM)
```

Registers Altered

- rD

Latency

3 cycles

Notes

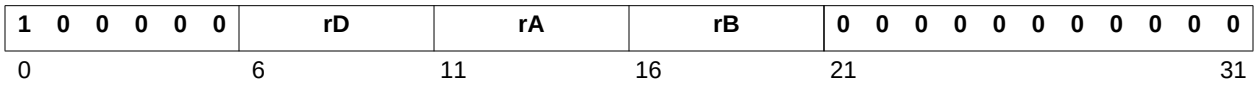
By default, Type B Instructions will take the 16-bit IMM field value and sign extend it to 32 bits to use as the immediate operand. This behavior can be overridden by preceding the Type B instruction with an imm instruction. See the imm instruction for details on using 32-bit immediate values.

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

or

Logical OR

or rD, rA, rB



Description

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

Pseudocode

```
(rD) ← (rA) ∨ (rB)
```

Registers Altered

- rD

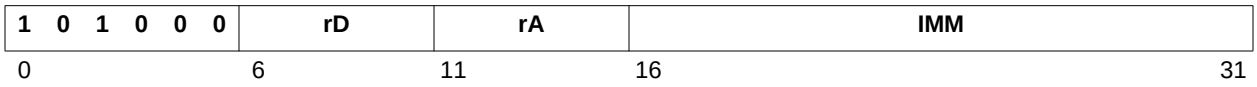
Latency

1 cycle

ori

Logical OR with Immediate

ori rD, rA, IMM



Description

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

Pseudocode

```
(rD) ← (rA) ∨ (IMM)
```

Registers Altered

- rD

Latency

1 cycle

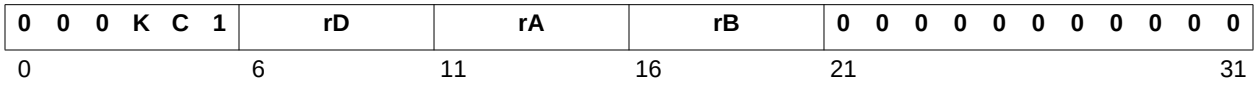
Note

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

rsub

Arithmetic Reverse Subtract

rsub	rD, rA, rB	Subtract
rsubc	rD, rA, rB	Subtract with Carry
rsubk	rD, rA, rB	Subtract and Keep Carry
rsubkc	rD, rA, rB	Subtract with Carry and Keep Carry



Description

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

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

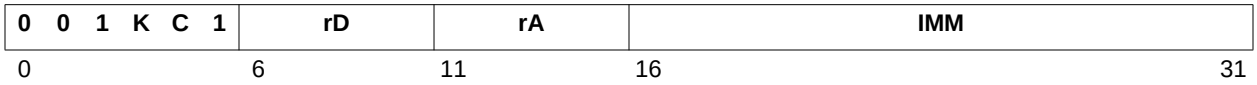
Notes

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

rsubi

Arithmetic Reverse Subtract Immediate

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



Description

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

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

Latency

1 cycle

Notes

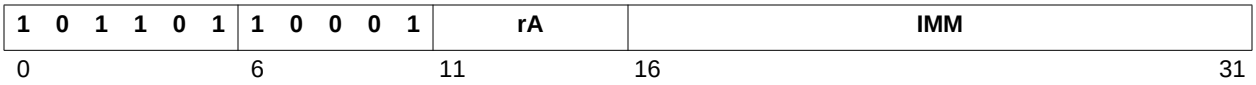
In subtractions, Carry = ($\overline{\text{Borrow}}$). When the Carry is set by a subtraction, it means that there is no Borrow, and when the Carry is cleared, it means that there is a Borrow.

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

rtbd

Return from Break

rtbd rA, IMM



Description

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

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

Pseudocode

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

Registers Altered

- PC
- MSR[BIP]

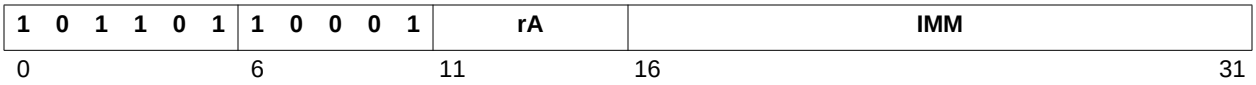
Latency

2 cycles

rtid

Return from Interrupt

rtid rA, IMM



Description

Return from interrupt will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits. It will also enable interrupts after execution.

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

Pseudocode

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

Registers Altered

- PC
- MSR[IE]

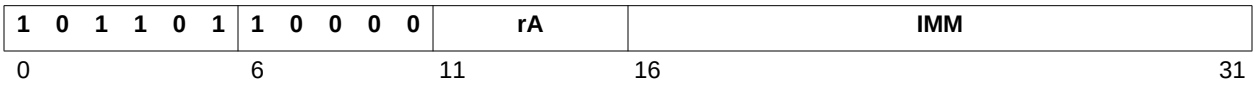
Latency

2 cycles

rtsd

Return from Subroutine

rtsd rA, IMM



Description

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

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

Pseudocode

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

Registers Altered

- PC

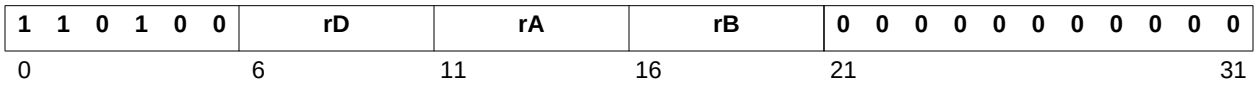
Latency

2 cycles

sb

Store Byte

sb rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- None

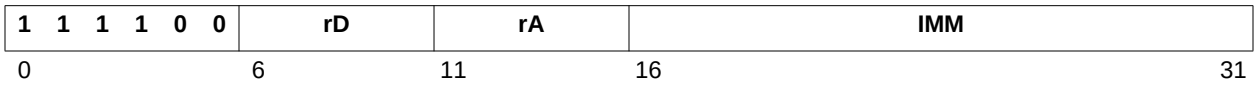
Latency

2 cycles

sbi

Store Byte Immediate

sbi rD, rA, IMM



Description

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

Pseudocode

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

Registers Altered

- None

Latency

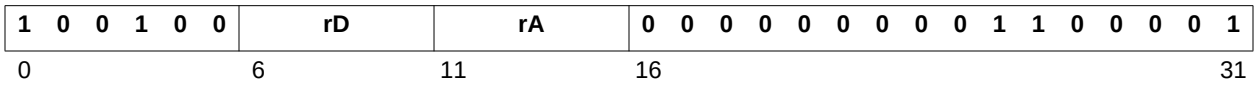
2 cycles

Note

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

sext16 Sign Extend Halfword

sext16 rD, rA



Description

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

Pseudocode

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

Registers Altered

- rD

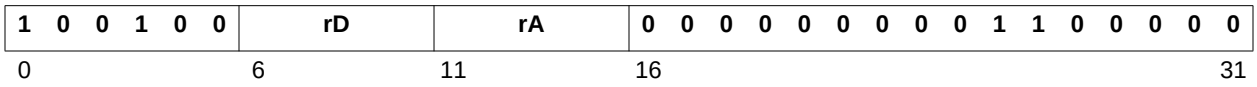
Latency

1 cycle

sext8

Sign Extend Byte

sext8 rD, rA



Description

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

Pseudocode

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

Registers Altered

- rD

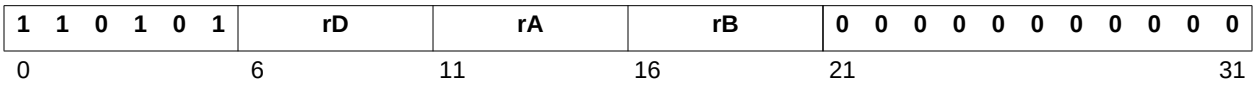
Latency

1 cycle

sh

Store Halfword

sh rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- None

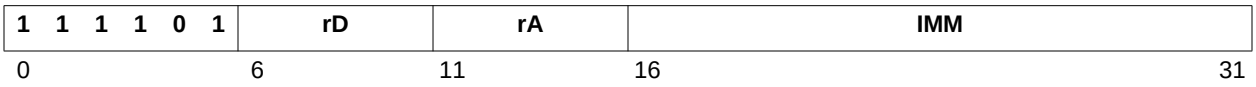
Latency

2 cycles

shi

Store Halfword Immediate

shi rD, rA, IMM



Description

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

Pseudocode

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

Registers Altered

- None

Latency

2 cycles

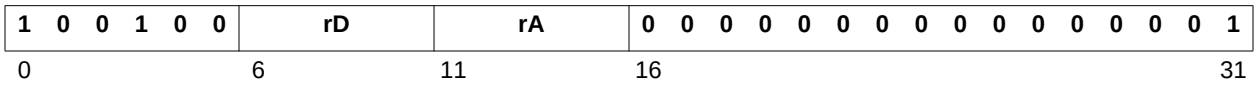
Note

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

sra

Shift Right Arithmetic

sra rD, rA



Description

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

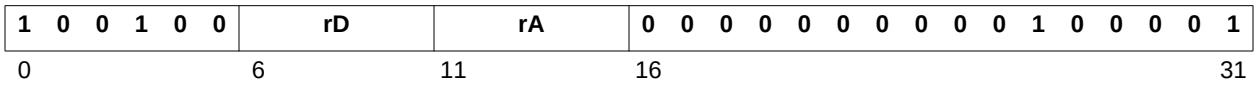
Latency

1 cycle

src

Shift Right with Carry

src rD, rA



Description

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

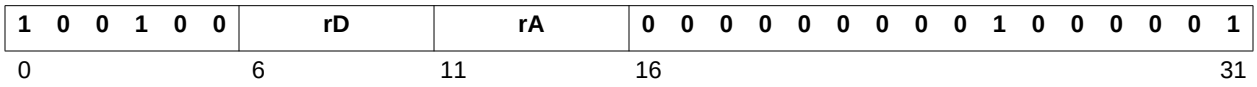
Latency

1 cycle

srl

Shift Right Logical

srl rD, rA



Description

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

Pseudocode

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

Registers Altered

- rD
- MSR[C]

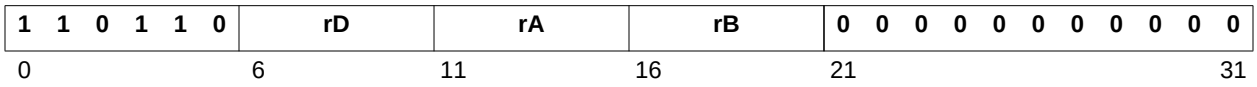
Latency

1 cycle

SW

Store Word

sw rD, rA, rB



Description

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

Pseudocode

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

Registers Altered

- None

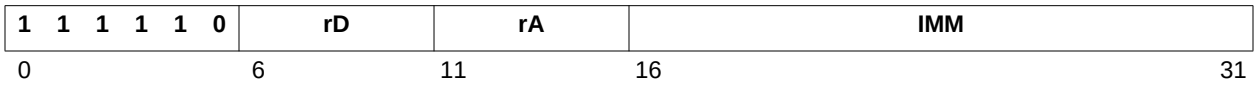
Latency

2 cycles

swi

Store Word Immediate

swi rD, rA, IMM



Description

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

Pseudocode

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

Register Altered

- None

Latency

2 cycles

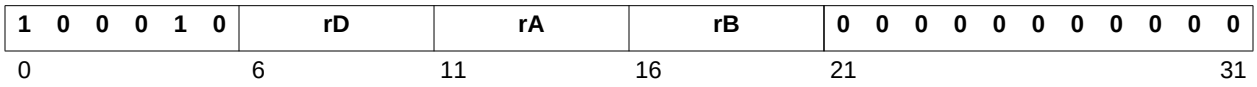
Note

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

xor

Logical Exclusive OR

xor rD, rA, rB



Description

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

Pseudocode

(rD) ← (rA) ⊕ (rB)

Registers Altered

- rD

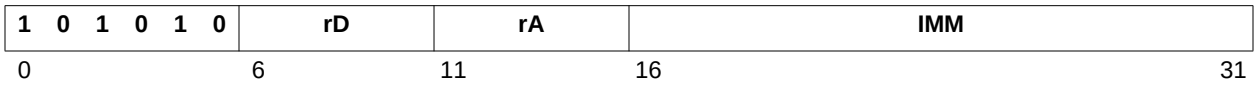
Latency

1 cycle

xori

Logical Exclusive OR with Immediate

xori rA, rD, IMM



Description

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

Pseudocode

```
(rD) ← (rA) ⊕ sext(IMM)
```

Registers Altered

- rD

Latency

1 cycle

Note

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