MicroBlaze Processor Reference Guide

2015.2

UG984 (v2015.2) June 24, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

06/24/2015: Released with Vivado Design Suite 2015.2 without changes from the previous version.

Date	Version	Revision
03/20/2013	2013.1	Initial Xilinx release. This User Guide is derived from UG081.
06/19/2013	2013.2	Updated for Vivado 2013.2 release.
10/02/2013	2013.3	Updated for Vivado 2013.3 release.
12/18/2013	2013.4	Updated for Vivado 2013.4 release.
04/02/2014	2014.1	Updated for Vivado 2014.1 release: • Added v9.3 to MicroBlaze release version code in PVR. • Clarified availability and behavior of stack protection registers. • Corrected description of LMB instruction and data bus exception. • Included description of extended debug features, new in version 9.3: performance monitoring, program trace and non-intrusive profiling. • Included definition of Reset Mode signals, new in version 9.3. • Clarified how the AXI4-Stream TLAST signal is handled. • Added UltraScale and updated performance and resource utilization for 2014.1.
10/01/2014	2014.3	Updated for Vivado 2014.3 release: • Corrected semantic description for PCMPEQ and PCMPNE in Table 2.1. • Added version 9.4 to MicroBlaze release version code in PVR. • Included description of external program trace, new in version 9.4
04/15/2015	2015.1	Updated for Vivado 2015.1 release: • Included description of 16 word cache line length, new in version 9.5. • Added version 9.5 to MicroBlaze release version code in PVR. • Corrected description of supported endianness and parameter C_ENDIANNESS. • Corrected description of outstanding reads for instruction and data cache. • Updated FPGA configuration memory protection document reference [Ref 10]. • Corrected Bus Index Range definitions for Lockstep Comparison in Table 3-12. • Clarified registers altered for IDIV instruction. • Corrected PVR assembler mnemonics for MFS instruction. • Updated performance and resource utilization for 2015.1. • Added references to training resources.

Table of Contents

Chapter 4: MicroBlaze Application Binary Interface

Chapter 5: MicroBlaze Instruction Set Architecture

Appendix A: Performance and Resource Utilization

Appendix B: Additional Resources and Legal Notices

Introduction

The MicroBlaze™ Processor Reference Guide provides information about the 32-bit soft processor, MicroBlaze, which is included in the Vivado release. The document is intended as a guide to the MicroBlaze hardware architecture.

Guide Contents

This guide contains the following chapters:

- Chapter 2, MicroBlaze Architecture, contains an overview of MicroBlaze features as well as information on Big-Endian and Little-Endian bit-reversed format, 32-bit general purpose registers, cache software support, and Fast Simplex Link interfaces.
- Chapter 3, MicroBlaze Signal Interface Description, describes the types of signal interfaces that can be used to connect MicroBlaze.
- Chapter 4, MicroBlaze Application Binary Interface, describes the Application Binary Interface important for developing software in assembly language for the soft processor.
- Chapter 5, MicroBlaze Instruction Set Architecture, provides notation, formats, and instructions for the Instruction Set Architecture of MicroBlaze.
- Appendix A, Performance and Resource Utilization, contains maximum frequencies and resource utilization numbers for different configurations.
- Appendix B, Additional Resources and Legal Notices, provides links to documentation and additional resources.

MicroBlaze Architecture

This chapter contains an overview of MicroBlaze™ features and detailed information on MicroBlaze architecture including Big-Endian or Little-Endian bit-reversed format, 32-bit general purpose registers, virtual-memory management, cache software support, and AXI4-Stream interfaces.

Overview

The MicroBlaze™ embedded processor soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx® Field Programmable Gate Arrays (FPGAs). Figure 2-1 shows a functional block diagram of the MicroBlaze core.

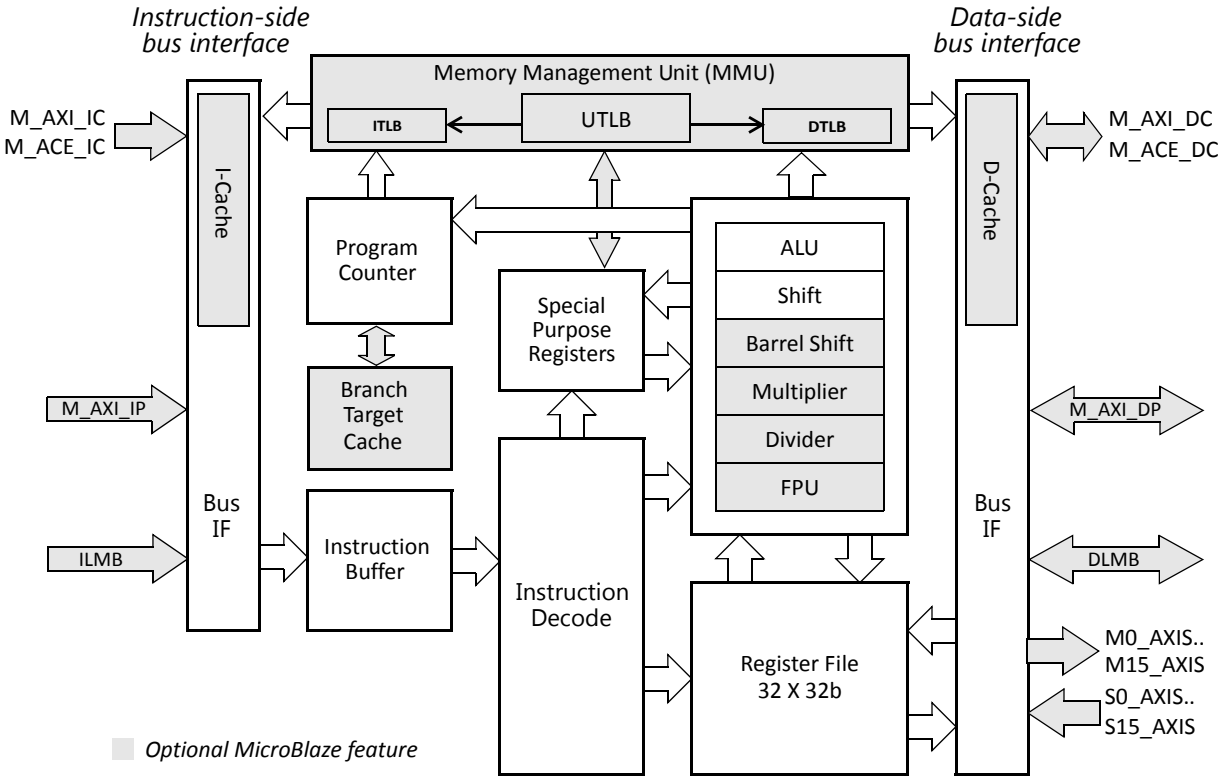

Figure 2-1: **MicroBlaze Core Block Diagram**

Features

The MicroBlaze soft core processor is highly configurable, allowing you to select a specific set of features required by your design.

The fixed feature set of the processor includes:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

In addition to these fixed features, the MicroBlaze processor is parameterized to allow selective enabling of additional functionality. Older (deprecated) versions of MicroBlaze support a subset of the optional features described in this manual. Only the latest (preferred) version of MicroBlaze (v9.5) supports all options.

Xilinx recommends that all new designs use the latest **preferred** version of the MicroBlaze processor.

Table 2-1, page 7 provides an overview of the configurable features by MicroBlaze versions.

Table 2-1: **Configurable Feature Overview by MicroBlaze Version**

Feature	MicroBlaze Versions					
	v9.0	v9.1	v9.2	v9.3	v9.4	v9.5
Version Status	deprecated	deprecated	deprecated	deprecated	deprecated	preferred
Processor pipeline depth	3/5	3/5	3/5	3/5	3/5	3/5
Local Memory Bus (LMB) data side interface	option	option	option	option	option	option
Local Memory Bus (LMB) instruction side interface	option	option	option	option	option	option
Hardware barrel shifter	option	option	option	option	option	option
Hardware divider	option	option	option	option	option	option
Hardware debug logic	option	option	option	option	option	option
Stream link interfaces	0-15 AXI	0-15 AXI	0-15 AXI	0-15 AXI	0-15 AXI	0-15 AXI
Machine status set and clear instructions	option	option	option	option	option	option
Cache line word length	4, 8	4, 8	4, 8	4, 8	4, 8	4, 8, 16
Hardware exception support	option	option	option	option	option	option
Pattern compare instructions	option	option	option	option	option	option
Floating point unit (FPU)	option	option	option	option	option	option
Disable hardware multiplier[1]	option	option	option	option	option	option

Table 2-1: **Configurable Feature Overview by MicroBlaze Version**

Feature	MicroBlaze Versions					
	v9.0	v9.1	v9.2	v9.3	v9.4	v9.5
Hardware debug readable ESR and EAR	Yes	Yes	Yes	Yes	Yes	Yes
Processor Version Register (PVR)	option	option	option	option	option	option
Area or speed optimized	option	option	option	option	option	option
Hardware multiplier 64-bit result	option	option	option	option	option	option
LUT cache memory	option	option	option	option	option	option
Floating point conversion and square root instructions	option	option	option	option	option	option
Memory Management Unit (MMU)	option	option	option	option	option	option
Extended stream instructions	option	option	option	option	option	option
Use Cache Interface for All I-Cache Memory Accesses	option	option	option	option	option	option
Use Cache Interface for All D-Cache Memory Accesses	option	option	option	option	option	option
Use Write-back Caching Policy for D-Cache	option	option	option	option	option	option
Branch Target Cache (BTC)	option	option	option	option	option	option
Streams for I-Cache	option	option	option	option	option	option
Victim handling for I-Cache	option	option	option	option	option	option
Victim handling for D-Cache	option	option	option	option	option	option
AXI4 (M_AXI_DP) data side interface	option	option	option	option	option	option
AXI4 (M_AXI_IP) instruction side interface	option	option	option	option	option	option
AXI4 (M_AXI_DC) protocol for D-Cache	option	option	option	option	option	option
AXI4 (M_AXI_IC) protocol for I-Cache	option	option	option	option	option	option
AXI4 protocol for stream accesses	option	option	option	option	option	option
Fault tolerant features	option	option	option	option	option	option
Force distributed RAM for cache tags	option	option	option	option	option	option
Configurable cache data widths	option	option	option	option	option	option
Count Leading Zeros instruction	option	option	option	option	option	option
Memory Barrier instruction	Yes	Yes	Yes	Yes	Yes	Yes
Stack overflow and underflow detection	option	option	option	option	option	option

Table 2-1: **Configurable Feature Overview by MicroBlaze Version**

Feature	MicroBlaze Versions					
	v9.0	v9.1	v9.2	v9.3	v9.4	v9.5
Allow stream instructions in user mode	option	option	option	option	option	option
Lockstep support	option	option	option	option	option	option
Configurable use of FPGA primitives	option	option	option	option	option	option
Low-latency interrupt mode	option	option	option	option	option	option
Swap instructions	option	option	option	option	option	option
Sleep mode and sleep instruction	Yes	Yes	Yes	Yes	Yes	Yes
Relocatable base vectors	option	option	option	option	option	option
ACE (M_ACE_DC) protocol for D-Cache	option	option	option	option	option	option
ACE (M_ACE_IC) protocol for I-Cache	option	option	option	option	option	option
Extended debug: performance monitoring, program trace, non-intrusive profiling				option	option	option
Reset mode: enter sleep or debug halt at reset				option	option	option
Extended debug: external program trace					option	option

1. Used for saving DSP48E primitives.

Data Types and Endianness

MicroBlaze uses Big-Endian or Little-Endian format to represent data, depending on the selected endianness. The parameter C_ENDIANNESS is fixed to 1 (little-endian).

The hardware supported data types for MicroBlaze are word, half word, and byte. When using the reversed load and store instructions LHUR, LWR, SHR and SWR, the bytes in the data are reversed, as indicated by the byte-reversed order.

The bit and byte organization for each type is shown in the following tables.

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

Big-Endian Byte Address	n	n+1	n+2	n+3
Big-Endian Byte Significance	MSByte			LSByte
Big-Endian Byte Order	n	n+1	n+2	n+3
Big-Endian Byte-Reversed Order	n+3	n+2	n+1	n
Little-Endian Byte Address	n+3	n+2	n+1	n
Little-Endian Byte Significance	MSByte			LSByte
Little-Endian Byte Order	n+3	n+2	n+1	n
Little-Endian Byte-Reversed Order	n	n+1	n+2	n+3
Bit Label	0			31
Bit Significance	MSBit			LSBit

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

Big-Endian Byte Address	n	n+1
Big-Endian Byte Significance	MSByte	LSByte
Big-Endian Byte Order	n	n+1
Big-Endian Byte-Reversed Order	n+1	n
Little-Endian Byte Address	n+1	n
Little-Endian Byte Significance	MSByte	LSByte
Little-Endian Byte Order	n+1	n
Little-Endian Byte-Reversed Order	n	n+1
Bit Label	0	15
Bit Significance	MSBit	LSBit

Table 2-4: **Byte Data Type**

Byte Address	n	
Bit Label	0	7
Bit Significance	MSBit	LSBit

Instructions

Instruction Summary

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 2-6 lists the MicroBlaze instruction set. Refer to Chapter 5, MicroBlaze Instruction Set Architecturefor more information on these instructions. Table 2-5 describes the instruction set nomenclature used in the semantics of each instruction.

Table 2-5: **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
SPR[*x*]	Special Purpose Register number *x*
MSR	Machine Status Register = SPR[1]
ESR	Exception Status Register = SPR[5]
EAR	Exception Address Register = SPR[3]
FSR	Floating Point Unit Status Register = SPR[7]
PVR*x*	Processor Version Register, where *x* is the register number = SPR[8192 + *x*]
BTR	Branch Target Register = SPR[11]
PC	Execute stage Program Counter = SPR[0]
x[*y*]	Bit *y* of register *x*
x[*y*:*z*]	Bit range *y* to *z* of register *x*
$\overline{x}$	Bit inverted value of register *x*
Imm	16 bit immediate value
Imm*x*	*x* bit immediate value
FSL*x*	4 bit AXI4-Stream port designator, where *x* is the port number
C	Carry flag, MSR[29]
Sa	Special Purpose Register, source operand
Sd	Special Purpose Register, destination operand
s(*x*)	Sign extend argument *x* to 32-bit value
*Addr	Memory contents at location Addr (data-size aligned)

Table 2-5: **Instruction Set Nomenclature *(Cont'd)***

Symbol	Description
:=	Assignment operator
=	Equality comparison
!=	Inequality comparison
>	Greater than comparison
>=	Greater than or equal comparison
<	Less than comparison
<=	Less than or equal comparison
+	Arithmetic add
*	Arithmetic multiply
/	Arithmetic divide
>> *x*	Bit shift right *x* bits
<< *x*	Bit shift left *x* bits
and	Logic AND
or	Logic OR
xor	Logic exclusive OR
op1 if cond else op2	Perform *op1* if condition *cond* is true, else perform *op2*
&	Concatenate. E.g. "0000100 & Imm7" is the concatenation of the fixed field "0000100" and a 7 bit immediate value.
signed	Operation performed on signed integer data type. All arithmetic operations are performed on signed word operands, unless otherwise specified
unsigned	Operation performed on unsigned integer data type
float	Operation performed on floating point data type
clz(r)	Count leading zeros

Table 2-6: **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
CMP Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000001	Rd := Rb + $\overline{Ra}$ + 1 Rd[0] := 0 if (Rb >= Ra) else Rd[0] := 1

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	**0-5**	**6-10**	**11-15**	**16-31**		
CMPU Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000011	Rd := Rb + $\overline{Ra}$ + 1 (unsigned) Rd[0] := 0 if (Rb >= Ra, unsigned) else Rd[0] := 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
MULH Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000001	Rd := (Ra * Rb) >> 32 (signed)
MULHU Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000011	Rd := (Ra * Rb) >> 32 (unsigned)
MULHSU Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000010	Rd := (Ra, signed * Rb, unsigned) >> 32 (signed)
BSRL Rd,Ra,Rb	010001	Rd	Ra	Rb	00000000000	Rd := 0 & (Ra >> Rb)
BSRA Rd,Ra,Rb	010001	Rd	Ra	Rb	01000000000	Rd := s(Ra >> Rb)
BSLL Rd,Ra,Rb	010001	Rd	Ra	Rb	10000000000	Rd := (Ra << Rb) & 0
IDIV Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000000	Rd := Rb/Ra
IDIVU Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000010	Rd := Rb/Ra, unsigned
*TNEA*GETD Rd,Rb	010011	Rd	00000	Rb	0*N*0*TAE* 00000	Rd := FSL Rb[28:31] (data read) MSR[FSL] := 1 if (FSL_S_Control = 1) MSR[C] := not FSL_S_Exists if *N* = 1
*TNA*PUTD Ra,Rb	010011	00000	Ra	Rb	0*N*0*TA*0 00000	FSL Rb[28:31] := Ra (data write) MSR[C] := FSL_M_Full if *N* = 1
*TNECA*GETD Rd,Rb	010011	Rd	00000	Rb	0*N*1*TAE* 00000	Rd := FSL Rb[28:31] (control read) MSR[FSL] := 1 if (FSL_S_Control = 0) MSR[C] := not FSL_S_Exists if *N* = 1
*TNCA*PUTD Ra,Rb	010011	00000	Ra	Rb	0*N*1*TA*0 00000	FSL Rb[28:31] := Ra (control write) MSR[C] := FSL_M_Full if *N* = 1
FADD Rd,Ra,Rb	010110	Rd	Ra	Rb	00000000000	Rd := Rb+Ra, float[1]
FRSUB Rd,Ra,Rb	010110	Rd	Ra	Rb	00010000000	Rd := Rb-Ra, float[1]
FMUL Rd,Ra,Rb	010110	Rd	Ra	Rb	00100000000	Rd := Rb*Ra, float[1]
FDIV Rd,Ra,Rb	010110	Rd	Ra	Rb	00110000000	Rd := Rb/Ra, float[1]

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
FCMP.UN Rd,Ra,Rb	010110	Rd	Ra	Rb	01000000000	Rd := 1 if (Rb = NaN or Ra = NaN, float[1]) else Rd := 0
FCMP.LT Rd,Ra,Rb	010110	Rd	Ra	Rb	01000010000	Rd := 1 if (Rb < Ra, float[1]) else Rd := 0
FCMP.EQ Rd,Ra,Rb	010110	Rd	Ra	Rb	01000100000	Rd := 1 if (Rb = Ra, float[1]) else Rd := 0
FCMP.LE Rd,Ra,Rb	010110	Rd	Ra	Rb	01000110000	Rd := 1 if (Rb <= Ra, float[1]) else Rd := 0
FCMP.GT Rd,Ra,Rb	010110	Rd	Ra	Rb	01001000000	Rd := 1 if (Rb > Ra, float[1]) else Rd := 0
FCMP.NE Rd,Ra,Rb	010110	Rd	Ra	Rb	01001010000	Rd := 1 if (Rb != Ra, float[1]) else Rd := 0
FCMP.GE Rd,Ra,Rb	010110	Rd	Ra	Rb	01001100000	Rd := 1 if (Rb >= Ra, float[1]) else Rd := 0
FLT Rd,Ra	010110	Rd	Ra	0	01010000000	Rd := float (Ra)[1]
FINT Rd,Ra	010110	Rd	Ra	0	01100000000	Rd := int (Ra)[1]
FSQRT Rd,Ra	010110	Rd	Ra	0	01110000000	Rd := sqrt (Ra)[1]
MULI Rd,Ra,Imm	011000	Rd	Ra	Imm		Rd := Ra * s(Imm)
BSRLI Rd,Ra,Imm	011001	Rd	Ra	00000000000 & Imm5		Rd : = 0 & (Ra >> Imm5)
BSRAI Rd,Ra,Imm	011001	Rd	Ra	00000010000 & Imm5		Rd := s(Ra >> Imm5)
BSLLI Rd,Ra,Imm	011001	Rd	Ra	00000100000 & Imm5		Rd := (Ra << Imm5) & 0
*TNEA*GET Rd,FSLx	011011	Rd	00000	0*N*0*TAE*000000 & FSLx		Rd := FSLx (data read, blocking if *N* = 0) MSR[FSL] := 1 if (FSL*x*_S_Control = 1) MSR[C] := not FSL*x*_S_Exists if *N* = 1
*TNA*PUT Ra,FSLx	011011	00000	Ra	1*N*0*TA*0000000 & FSLx		FSLx := Ra (data write, blocking if *N* = 0) MSR[C] := FSL*x*_M_Full if *N* = 1
*TNECA*GET Rd,FSLx	011011	Rd	00000	0*N*1*TAE*000000 & FSLx		Rd := FSLx (control read, blocking if *N* = 0) MSR[FSL] := 1 if (FSL*x*_S_Control = 0) MSR[C] := not FSL*x*_S_Exists if *N* = 1
*TNCA*PUT Ra,FSLx	011011	00000	Ra	1*N*1*TA*0000000 & FSLx		FSLx := Ra (control write, blocking if *N* = 0) MSR[C] := FSL*x*_M_Full if *N* = 1
OR Rd,Ra,Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
PCMPBF Rd,Ra,Rb	100000	Rd	Ra	Rb	10000000000	Rd := 1 if (Rb[0:7] = Ra[0:7]) else Rd := 2 if (Rb[8:15] = Ra[8:15]) else Rd := 3 if (Rb[16:23] = Ra[16:23]) else Rd := 4 if (Rb[24:31] = Ra[24:31]) else Rd := 0

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	**0-5**	**6-10**	**11-15**	**16-31**		
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
PCMPEQ Rd,Ra,Rb	100010	Rd	Ra	Rb	10000000000	Rd := 1 if (Rb = Ra) else Rd := 0
ANDN Rd,Ra,Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{Rb}$
PCMPNE Rd,Ra,Rb	100011	Rd	Ra	Rb	10000000000	Rd := 1 if (Rb != Ra) else Rd := 0
SRA Rd,Ra	100100	Rd	Ra	0000000000000001		Rd := s(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]
SEXT8 Rd,Ra	100100	Rd	Ra	0000000001100000		Rd := s(Ra[24:31])
SEXT16 Rd,Ra	100100	Rd	Ra	0000000001100001		Rd := s(Ra[16:31])
CLZ Rd, Ra	100100	Rd	Ra	0000000011100000		Rd = clz(Ra)
SWAPB Rd, Ra	100100	Rd	Ra	0000000111100000		Rd = (Ra)[24:31, 16:23, 8:15, 0:7]
SWAPH Rd, Ra	100100	Rd	Ra	0000000111100010		Rd = (Ra)[16:31, 0:15]
WIC Ra,Rb	100100	00000	Ra	Rb	00001101000	ICache_Line[Ra >> 4].Tag := 0 if (C_ICACHE_LINE_LEN = 4) ICache_Line[Ra >> 5].Tag := 0 if (C_ICACHE_LINE_LEN = 8) ICache_Line[Ra >> 6].Tag := 0 if (C_ICACHE_LINE_LEN = 16)
WDC Ra,Rb	100100	00000	Ra	Rb	00001100100	Cache line is cleared, discarding stored data. DCache_Line[Ra >> 4].Tag := 0 if (C_DCACHE_LINE_LEN = 4) DCache_Line[Ra >> 5].Tag := 0 if (C_DCACHE_LINE_LEN = 8) DCache_Line[Ra >> 6].Tag := 0 if (C_DCACHE_LINE_LEN = 16)
WDC.FLUSH Ra,Rb	100100	00000	Ra	Rb	00001110100	Cache line is flushed, writing stored data to memory, and then cleared. Used when C_DCACHE_USE_WRITEBACK = 1.
WDC.CLEAR Ra,Rb	100100	00000	Ra	Rb	00001110110	Cache line with matching address is cleared, discarding stored data. Used when C_DCACHE_USE_WRITEBACK = 1.

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
MTS Sd,Ra	100101	00000	Ra	11 & Sd		SPR[Sd] := Ra, where: · SPR[0x0001] is MSR · SPR[0x0007] is FSR · SPR[0x0800] is SLR · SPR[0x0802] is SHR · SPR[0x1000] is PID · SPR[0x1001] is ZPR · SPR[0x1002] is TLBX · SPR[0x1003] is TLBLO · SPR[0x1004] is TLBHI · SPR[0x1005] is TLBSX
MFS Rd,Sa	100101	Rd	00000	10 & Sa		Rd := SPR[Sa], where: · SPR[0x0000] is PC · SPR[0x0001] is MSR · SPR[0x0003] is EAR · SPR[0x0005] is ESR · SPR[0x0007] is FSR · SPR[0x000B] is BTR · SPR[0x000D] is EDR · SPR[0x0800] is SLR · SPR[0x0802] is SHR · SPR[0x1000] is PID · SPR[0x1001] is ZPR · SPR[0x1002] is TLBX · SPR[0x1003] is TLBLO · SPR[0x1004] is TLBHI · SPR[0x2000 to 0x200B] is PVR[0 to 12]
MSRCLR Rd,Imm	100101	Rd	00001	00 & Imm14		Rd := MSR MSR := MSR and $\overline{\text{Imm14}}$
MSRSET Rd,Imm	100101	Rd	00000	00 & Imm14		Rd := MSR MSR := MSR or Imm14
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

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
BRK Rd,Rb	100110	Rd	01100	Rb	00000000000	PC := Rb Rd := PC MSR[BIP] := 1
BEQ Ra,Rb	100111	00000	Ra	Rb	00000000000	PC := PC + Rb if Ra = 0
BNE Ra,Rb	100111	00001	Ra	Rb	00000000000	PC := PC + Rb if Ra != 0
BLT Ra,Rb	100111	00010	Ra	Rb	00000000000	PC := PC + Rb if Ra < 0
BLE Ra,Rb	100111	00011	Ra	Rb	00000000000	PC := PC + Rb if Ra <= 0
BGT Ra,Rb	100111	00100	Ra	Rb	00000000000	PC := PC + Rb if Ra > 0
BGE Ra,Rb	100111	00101	Ra	Rb	00000000000	PC := PC + Rb if Ra >= 0
BEQD Ra,Rb	100111	10000	Ra	Rb	00000000000	PC := PC + Rb if Ra = 0
BNED Ra,Rb	100111	10001	Ra	Rb	00000000000	PC := PC + Rb if Ra != 0
BLTD Ra,Rb	100111	10010	Ra	Rb	00000000000	PC := PC + Rb if Ra < 0
BLED Ra,Rb	100111	10011	Ra	Rb	00000000000	PC := PC + Rb if Ra <= 0
BGTD Ra,Rb	100111	10100	Ra	Rb	00000000000	PC := PC + Rb if Ra > 0
BGED Ra,Rb	100111	10101	Ra	Rb	00000000000	PC := PC + Rb if Ra >= 0
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
RTED Ra,Imm	101101	10100	Ra	Imm		PC := Ra + s(Imm) MSR[EE] := 1, MSR[EIP] := 0 ESR := 0
BRI Imm	101110	00000	00000	Imm		PC := PC + s(Imm)
MBAR Imm	101110	Imm	00010	0000000000000100		PC := PC + 4; Wait for memory accesses.
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)

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	**0-5**	**6-10**	**11-15**	**16-31**		
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		PC := PC + s(Imm) if Ra = 0
BNEI Ra,Imm	101111	00001	Ra	Imm		PC := PC + s(Imm) if Ra != 0
BLTI Ra,Imm	101111	00010	Ra	Imm		PC := PC + s(Imm) if Ra < 0
BLEI Ra,Imm	101111	00011	Ra	Imm		PC := PC + s(Imm) if Ra <= 0
BGTI Ra,Imm	101111	00100	Ra	Imm		PC := PC + s(Imm) if Ra > 0
BGEI Ra,Imm	101111	00101	Ra	Imm		PC := PC + s(Imm) if Ra >= 0
BEQID Ra,Imm	101111	10000	Ra	Imm		PC := PC + s(Imm) if Ra = 0
BNEID Ra,Imm	101111	10001	Ra	Imm		PC := PC + s(Imm) if Ra != 0
BLTID Ra,Imm	101111	10010	Ra	Imm		PC := PC + s(Imm) if Ra < 0
BLEID Ra,Imm	101111	10011	Ra	Imm		PC := PC + s(Imm) if Ra <= 0
BGTID Ra,Imm	101111	10100	Ra	Imm		PC := PC + s(Imm) if Ra > 0
BGEID Ra,Imm	101111	10101	Ra	Imm		PC := PC + s(Imm) if Ra >= 0
LBU Rd,Ra,Rb LBUR Rd,Ra,Rb	110000	Rd	Ra	Rb	00000000000 01000000000	Addr := Ra + Rb Rd[0:23] := 0 Rd[24:31] := *Addr[0:7]
LHU Rd,Ra,Rb LHUR Rd,Ra,Rb	110001	Rd	Ra	Rb	00000000000 01000000000	Addr := Ra + Rb Rd[0:15] := 0 Rd[16:31] := *Addr[0:15]
LW Rd,Ra,Rb LWR Rd,Ra,Rb	110010	Rd	Ra	Rb	00000000000 01000000000	Addr := Ra + Rb Rd := *Addr
LWX Rd,Ra,Rb	110010	Rd	Ra	Rb	10000000000	Addr := Ra + Rb Rd := *Addr Reservation := 1
SB Rd,Ra,Rb SBR Rd,Ra,Rb	110100	Rd	Ra	Rb	00000000000 01000000000	Addr := Ra + Rb *Addr[0:8] := Rd[24:31]
SH Rd,Ra,Rb SHR Rd,Ra,Rb	110101	Rd	Ra	Rb	00000000000 01000000000	Addr := Ra + Rb *Addr[0:16] := Rd[16:31]
SW Rd,Ra,Rb SWR Rd,Ra,Rb	110110	Rd	Ra	Rb	00000000000 01000000000	Addr := Ra + Rb *Addr := Rd
SWX Rd,Ra,Rb	110110	Rd	Ra	Rb	10000000000	Addr := Ra + Rb *Addr := Rd if Reservation = 1 Reservation := 0

Table 2-6: **MicroBlaze Instruction Set Summary *(Cont'd)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	**0-5**	**6-10**	**11-15**	**16-31**		
LBUI Rd,Ra,Imm	111000	Rd	Ra	Imm		Addr := Ra + s(Imm) Rd[0:23] := 0 Rd[24:31] := *Addr[0:7]
LHUI Rd,Ra,Imm	111001	Rd	Ra	Imm		Addr := Ra + s(Imm) Rd[0:15] := 0 Rd[16:31] := *Addr[0:15]
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[0:7] := Rd[24:31]
SHI Rd,Ra,Imm	111101	Rd	Ra	Imm		Addr := Ra + s(Imm) *Addr[0:15] := Rd[16:31]
SWI Rd,Ra,Imm	111110	Rd	Ra	Imm		Addr := Ra + s(Imm) *Addr := Rd

1. Due to the many different corner cases involved in floating point arithmetic, only the normal behavior is described. A full description of the behavior can be found in Chapter 5, "MicroBlaze Instruction Set Architecture."

Semaphore Synchronization

The LWX and SWX. instructions are used to implement common semaphore operations, including test and set, compare and swap, exchange memory, and fetch and add. They are also used to implement spinlocks.

These instructions are typically used by system programs and are called by application programs as needed. Generally, a program uses LWX to load a semaphore from memory, causing the reservation to be set (the processor maintains the reservation internally). The program can compute a result based on the semaphore value and conditionally store the result back to the same memory location using the SWX instruction. The conditional store is performed based on the existence of the reservation established by the preceding LWX instruction. If the reservation exists when the store is executed, the store is performed and MSR[C] is cleared to 0. If the reservation does not exist when the store is executed, the target memory location is not modified and MSR[C] is set to 1.

If the store is successful, the sequence of instructions from the semaphore load to the semaphore store appear to be executed atomically—no other device modified the semaphore location between the read and the update. Other devices can read from the semaphore location during the operation. For a semaphore operation to work properly, the LWX instruction must be paired with an SWX instruction, and both must specify identical addresses. The reservation granularity in MicroBlaze is a word. For both instructions, the address must be word aligned. No unaligned exceptions are generated for these instructions.

The conditional store is always attempted when a reservation exists, even if the store address does not match the load address that set the reservation.

Only one reservation can be maintained at a time. The address associated with the reservation can be changed by executing a subsequent LWX instruction. The conditional store is performed based upon the reservation established by the last LWX instruction executed. Executing an SWX instruction always clears a reservation held by the processor, whether the address matches that established by the LWX or not.

Reset, interrupts, exceptions, and breaks (including the BRK and BRKI instructions) all clear the reservation.

The following provides general guidelines for using the LWX and SWX instructions:

- The LWX and SWX instructions should be paired and use the same address.
- An unpaired SWX instruction to an arbitrary address can be used to clear any reservation held by the processor.
- A conditional sequence begins with an LWX instruction. It can be followed by memory accesses and/or computations on the loaded value. The sequence ends with an SWX instruction. In most cases, failure of the SWX instruction should cause a branch back to the LWX for a repeated attempt.
- An LWX instruction can be left unpaired when executing certain synchronization primitives if the value loaded by the LWX is not zero. An implementation of Test and Set exemplifies this:

```
loop:  lwx    r5,r3,r0  ; load and reserve
       bnei   r5,next   ; branch if not equal to zero
       addik  r5,r5,1   ; increment value
       swx    r5,r3,r0  ; try to store non-zero value
       addic  r5,r0,0   ; check reservation
       bnei   r5,loop   ; loop if reservation lost
next:
```

- Performance can be improved by minimizing looping on an LWX instruction that fails to return a desired value. Performance can also be improved by using an ordinary load instruction to do the initial value check. An implementation of a spinlock exemplifies this:

```
loop:  lw     r5,r3,r0  ; load the word
       bnei   r5,loop   ; loop back if word not equal to 0
       lwx    r5,r3,r0  ; try reserving again
       bnei   r5,loop   ; likely that no branch is needed
       addik  r5,r5,1   ; increment value
       swx    r5,r3,r0  ; try to store non-zero value
       addic  r5,r0,0   ; check reservation
       bnei   r5,loop   ; loop if reservation lost
```

- Minimizing the looping on an LWX/SWX instruction pair increases the likelihood that forward progress is made. The old value should be tested before attempting the store. If the order is reversed (store before load), more SWX instructions are executed and reservations are more likely to be lost between the LWX and SWX instructions.

Self-modifying Code

When using self-modifying code software must ensure that the modified instructions have been written to memory prior to fetching them for execution. There are several aspects to consider:

- The instructions to be modified may already have been fetched prior to modification:
  - into the instruction prefetch buffer,
  - into the instruction cache, if it is enabled,
  - into a stream buffer, if instruction cache stream buffers are used,
  - into the instruction cache, and then saved in a victim buffer, if victim buffers are used.

  To ensure that the modified code is always executed instead of the old unmodified code, software must handle all these cases.

- If one or more of the instructions to be modified is a branch, and the branch target cache is used, the branch target address may have been cached.

  To avoid using the cached branch target address, software must ensure that the branch target cache is cleared prior to executing the modified code.

- The modified instructions may not have been written to memory prior to execution:
  - they may be en route to memory, in temporary storage in the interconnect or the memory controller,
  - they may be stored in the data cache, if write-back cache is used,
  - they may be saved in a victim buffer, if write-back cache and victim buffers are used.

  Software must ensure that the modified instructions have been written to memory before being fetched by the processor.

The annotated code below shows how each of the above issues can be addressed. This code assumes that both instruction cache and write-back data cache is used. If not, the corresponding instructions can be omitted.

The following code exemplifies storing a modified instruction:

```
swi       r5,r6,0  ; r5 = new instruction
                   ; r6 = physical instruction address
wdc.flush r6,r0    ; flush write-back data cache line
mbar      1        ; ensure new instruction is written to memory
wic       r7,r0    ; invalidate line, empty stream & victim buffers
                   ; r7 = virtual instruction address
mbar      2        ; empty prefetch buffer, clear branch target cache
```

The physical and virtual addresses above are identical, unless MMU virtual mode is used. If the MMU is enabled, the code sequences must be executed in real mode, since WIC and WDC are privileged instructions. The first instruction after the code sequences above must not be modified, since it may have been prefetched.

Registers

MicroBlaze has an orthogonal instruction set architecture. It has thirty-two 32-bit general purpose registers and up to eighteen 32-bit special purpose registers, depending on configured options.

General Purpose Registers

The thirty-two 32-bit General Purpose Registers are numbered R0 through R31. The register file is reset on bit stream download (reset value is 0x00000000). Figure 2-2 is a representation of a General Purpose Register and Table 2-7 provides a description of each register and the register reset value (if existing).

Note: The register file is **not** reset by the external reset inputs: `Reset` and `Debug_Rst`.

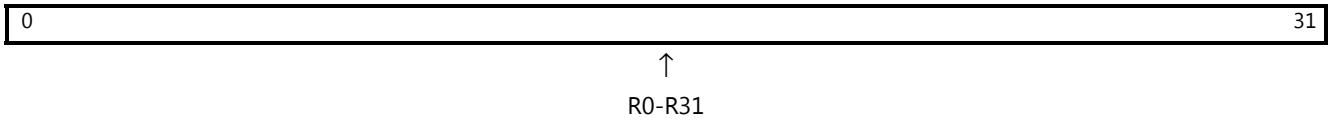

Figure 2-2: **R0-R31**

Table 2-7: **General Purpose Registers (R0-R31)**

Bits	Name	Description	Reset Value
0:31	R0	Always has a value of zero. Anything written to R0 is discarded	0x00000000
0:31	R1 through R13	32-bit general purpose registers	-
0:31	R14	32-bit register used to store return addresses for interrupts.	-
0:31	R15	32-bit general purpose register. Recommended for storing return addresses for user vectors.	-
0:31	R16	32-bit register used to store return addresses for breaks.	-
0:31	R17	If MicroBlaze is configured to support hardware exceptions, this register is loaded with the address of the instruction following the instruction causing the HW exception, except for exceptions in delay slots that use BTR instead (see "Branch Target Register (BTR)"); if not, it is a general purpose register.	-
0:31	R18 through R31	R18 through R31 are 32-bit general purpose registers.	-

Refer to Table 4-2 for software conventions on general purpose register usage.

Special Purpose Registers

Program Counter (PC)

The Program Counter (PC) is the 32-bit address of the execution instruction. It can be read with an MFS instruction, but it cannot be written with an MTS instruction. When used with the MFS instruction the PC register is specified by setting Sa = 0x0000. Figure 2-3 illustrates the PC and Table 2-8 provides a description and reset value.

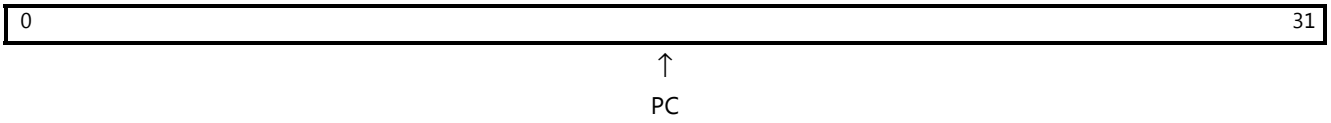

Figure 2-3: **PC**

Table 2-8: **Program Counter (PC)**

Bits	Name	Description	Reset Value
0:31	PC	Program Counter Address of executing instruction, that is, “mfs r2 0” stores the address of the mfs instruction itself in R2.	0x00000000

Machine Status Register (MSR)

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

When writing to the MSR using MSRSET or MSRCLR, the Carry bit takes effect immediately and the remaining bits take effect one clock cycle later. When writing using MTS, all bits take effect one clock cycle later. Any value written to bit 0 is discarded.

When used with an MTS or MFS instruction, the MSR is specified by setting S*x* = 0x0001. Figure 2-4 illustrates the MSR register and Table 2-9 provides the bit description and reset values.

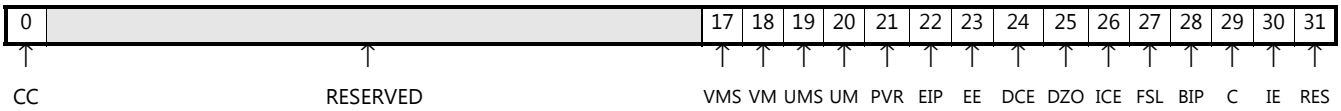

Figure 2-4: **MSR**

Table 2-9: **Machine Status Register (MSR)**

Bits	Name	Description	Reset Value
0	CC	Arithmetic Carry Copy Copy of the Arithmetic Carry (bit 29). CC is always the same as bit C.	0
1:16	Reserved		
17	VMS	Virtual Protected Mode Save Only available when configured with an MMU (if C_USE_MMU > 1 and C_AREA_OPTIMIZED = 0) Read/Write	0
18	VM	Virtual Protected Mode 0 = MMU address translation and access protection disabled, with C_USE_MMU = 3 (Virtual). Access protection disabled with C_USE_MMU = 2 (Protection) 1 = MMU address translation and access protection enabled, with C_USE_MMU = 3 (Virtual). Access protection enabled, with C_USE_MMU = 2 (Protection). Only available when configured with an MMU (if C_USE_MMU > 1 and C_AREA_OPTIMIZED = 0) Read/Write	0
19	UMS	User Mode Save Only available when configured with an MMU (if C_USE_MMU > 0 and C_AREA_OPTIMIZED = 0) Read/Write	0
20	UM	User Mode 0 = Privileged Mode, all instructions are allowed 1 = User Mode, certain instructions are not allowed Only available when configured with an MMU (if C_USE_MMU > 0 and C_AREA_OPTIMIZED = 0) Read/Write	0
21	PVR	Processor Version Register exists 0 = No Processor Version Register 1 = Processor Version Register exists Read only	Based on parameter C_PVR

Table 2-9: **Machine Status Register (MSR) *(Cont'd)***

Bits	Name	Description	Reset Value
22	EIP	Exception In Progress 0 = No hardware exception in progress 1 = Hardware exception in progress Only available if configured with exception support (C_*_EXCEPTION or C_USE_MMU > 0) Read/Write	0
23	EE	Exception Enable 0 = Hardware exceptions disabled[1] 1 = Hardware exceptions enabled Only available if configured with exception support (C_*_EXCEPTION or C_USE_MMU > 0) Read/Write	0
24	DCE	Data Cache Enable 0 = Data Cache disabled 1 = Data Cache enabled Only available if configured to use data cache (C_USE_DCACHE = 1) Read/Write	0
25	DZO	Division by Zero or Division Overflow[2] 0 = No division by zero or division overflow has occurred 1 = Division by zero or division overflow has occurred Only available if configured to use hardware divider (C_USE_DIV = 1) Read/Write	0
26	ICE	Instruction Cache Enable 0 = Instruction Cache disabled 1 = Instruction Cache enabled Only available if configured to use instruction cache (C_USE_ICACHE = 1) Read/Write	0

Table 2-9: **Machine Status Register (MSR) *(Cont'd)***

Bits	Name	Description	Reset Value
27	FSL	AXI4-Stream Error 0 = get or getd had no error 1 = get or getd control type mismatch This bit is sticky, i.e. it is set by a get or getd instruction when a control bit mismatch occurs. To clear it an mts or msrclr instruction must be used. Only available if configured to use stream links (C_FSL_LINKS > 0) Read/Write	0
28	BIP	Break in Progress 0 = No Break in Progress 1 = Break in Progress Break Sources can be software break instruction or hardware break from Ext_Brk or Ext_NM_Brk pin. Read/Write	0
29	C	Arithmetic Carry 0 = No Carry (Borrow) 1 = Carry (No Borrow) Read/Write	0
30	IE	Interrupt Enable 0 = Interrupts disabled 1 = Interrupts enabled Read/Write	0
31	-	Reserved	0

1. The MMU exceptions (Data Storage Exception, Instruction Storage Exception, Data TLB Miss Exception, Instruction TLB Miss Exception) cannot be disabled, and are not affected by this bit.
2. This bit is only used for integer divide-by-zero or divide overflow signaling. There is a floating point equivalent in the FSR. The DZO-bit flags divide by zero or divide overflow conditions regardless if the processor is configured with exception handling or not.

Exception Address Register (EAR)

The Exception Address Register stores the full load/store address that caused the exception for the following:

- An unaligned access exception that means the unaligned access address
- An M_AXI_DP exception that specifies the failing AXI4 data access address
- A data storage exception that specifies the (virtual) effective address accessed
- An instruction storage exception that specifies the (virtual) effective address read
- A data TLB miss exception that specifies the (virtual) effective address accessed
- An instruction TLB miss exception that specifies the (virtual) effective address read

The contents of this register is undefined for all other exceptions. When read with the MFS instruction, the EAR is specified by setting Sa = 0x0003. The EAR register is illustrated in Figure 2-5 and Table 2-10 provides bit descriptions and reset values.

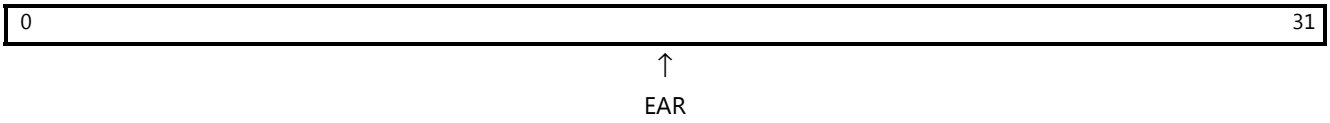

Figure 2-5: **EAR**

Table 2-10: **Exception Address Register (EAR)**

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

Exception Status Register (ESR)

The Exception Status Register contains status bits for the processor. When read with the MFS instruction, the ESR is specified by setting Sa = 0x0005. The ESR register is illustrated in Figure 2-6, Table 2-11 provides bit descriptions and reset values, and Table 2-12 provides the Exception Specific Status (ESS).

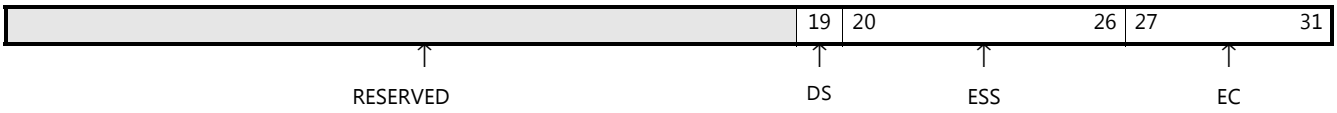

Figure 2-6: **ESR**

Table 2-11: **Exception Status Register (ESR)**

Bits	Name	Description	Reset Value
0:18	Reserved		
19	DS	Delay Slot Exception. 0 = not caused by delay slot instruction 1 = caused by delay slot instruction Read-only	0

Table 2-11: **Exception Status Register (ESR) *(Cont'd)***

Bits	Name	Description	Reset Value
20:26	ESS	Exception Specific Status For details refer to Table 2-12. Read-only	See Table 2-12
27:31	EC	Exception Cause 00000 = Stream exception 00001 = Unaligned data access exception 00010 = Illegal op-code exception 00011 = Instruction bus error exception 00100 = Data bus error exception 00101 = Divide exception 00110 = Floating point unit exception 00111 = Privileged instruction exception 00111 = Stack protection violation exception 10000 = Data storage exception 10001 = Instruction storage exception 10010 = Data TLB miss exception 10011 = Instruction TLB miss exception Read-only	0

Table 2-12: **Exception Specific Status (ESS)**

Exception Cause	Bits	Name	Description	Reset Value
Unaligned Data Access	20	W	Word Access Exception 0 = unaligned halfword access 1 = unaligned word access	0
	21	S	Store Access Exception 0 = unaligned load access 1 = unaligned store access	0
	22:26	Rx	Source/Destination Register General purpose register used as source (Store) or destination (Load) in unaligned access	0
Illegal Instruction	20:26	Reserved		0
Instruction bus error	20	ECC	Exception caused by ILMB correctable or uncorrectable error	0
	21:26	Reserved		0
Data bus error	20	ECC	Exception caused by DLMB correctable or uncorrectable error	0
	21:26	Reserved		0
Divide	20	DEC	Divide - Division exception cause 0 = Divide-By-Zero 1 = Division Overflow	0
	21:26	Reserved		0
Floating point unit	20:26	Reserved		0

Table 2-12: **Exception Specific Status (ESS) *(Cont'd)***

Exception Cause	Bits	Name	Description	Reset Value
Privileged instruction	20:26	Reserved		0
Stack protection violation	20:26	Reserved		0
Stream	20:22	Reserved		0
	23:26	FSL	AXI4-Stream index that caused the exception	0
Data storage	20	DIZ	Data storage - Zone protection 0 = Did not occur 1 = Occurred	0
	21	S	Data storage - Store instruction 0 = Did not occur 1 = Occurred	0
	22:26	Reserved		0
Instruction storage	20	DIZ	Instruction storage - Zone protection 0 = Did not occur 1 = Occurred	0
	21:26	Reserved		0
Data TLB miss	20	Reserved		0
	21	S	Data TLB miss - Store instruction 0 = Did not occur 1 = Occurred	0
	22:26	Reserved		0
Instruction TLB miss	20:26	Reserved		0

Branch Target Register (BTR)

The Branch Target Register only exists if the MicroBlaze processor is configured to use exceptions. The register stores the branch target address for all delay slot branch instructions executed while MSR[EIP] = 0. If an exception is caused by an instruction in a delay slot (that is, ESR[DS]=1), the exception handler should return execution to the address stored in BTR instead of the normal exception return address stored in R17. When read with the MFS instruction, the BTR is specified by setting Sa = 0x000B. The BTR register is illustrated in Figure 2-7 and Table 2-13 provides bit descriptions and reset values.

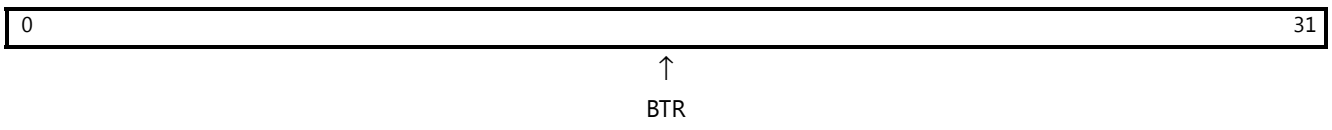

Figure 2-7: **BTR**

Table 2-13: **Branch Target Register (BTR)**

Bits	Name	Description	Reset Value
0:31	BTR	Branch target address used by handler when returning from an exception caused by an instruction in a delay slot. Read-only	0x00000000

Floating Point Status Register (FSR)

The Floating Point Status Register contains status bits for the floating point unit. It can be read with an MFS, and written with an MTS instruction. When read or written, the register is specified by setting Sa = 0x0007. The bits in this register are sticky – floating point instructions can only set bits in the register, and the only way to clear the register is by using the MTS instruction. Figure 2-8 illustrates the FSR register and Table 2-14 provides bit descriptions and reset values.

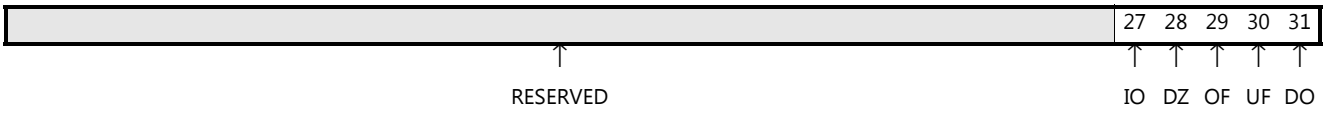

Figure 2-8: **FSR**

Table 2-14: **Floating Point Status Register (FSR)**

Bits	Name	Description	Reset Value
0:26	Reserved		undefined
27	IO	Invalid operation	0
28	DZ	Divide-by-zero	0
29	OF	Overflow	0
30	UF	Underflow	0
31	DO	Denormalized operand error	0

Exception Data Register (EDR)

The Exception Data Register stores data read on an AXI4-Stream link that caused a stream exception.

The contents of this register is undefined for all other exceptions. When read with the MFS instruction, the EDR is specified by setting Sa = 0x000D. Figure 2-9 illustrates the EDR register and Table 2-15 provides bit descriptions and reset values.

Note: The register is only implemented if `C_FSL_LINKS` is greater than 0 and `C_FSL_EXCEPTION` is set to 1.

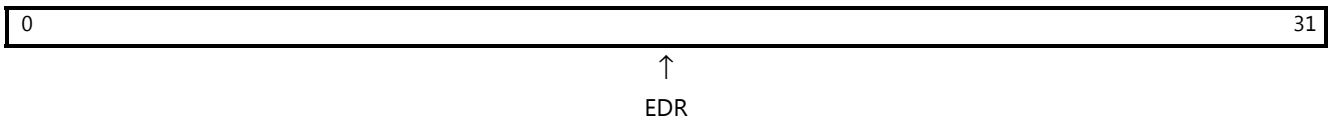

Figure 2-9: **EDR**

Table 2-15: **Exception Data Register (EDR)**

Bits	Name	Description	Reset Value
0:31	EDR	Exception Data Register	0x00000000

Stack Low Register (SLR)

The Stack Low Register stores the stack low limit use to detect stack overflow. When the address of a load or store instruction using the stack pointer (register R1) as rA is less than the Stack Low Register, a stack overflow occurs, causing a Stack Protection Violation exception if exceptions are enabled in MSR.

When read with the MFS instruction, the SLR is specified by setting Sa = 0x0800. Figure 2-10 illustrates the SLR register and Table 2-16 provides bit descriptions and reset values.

Note: The register is only implemented if stack protection is enabled by setting the parameter C_USE_STACK_PROTECTION to 1. If stack protection is not implemented, writing to the register has no effect.

Note: Stack protection is not available when the MMU is enabled (C_USE_MMU > 0). With the MMU page-based memory protection is provided through the UTLB instead.

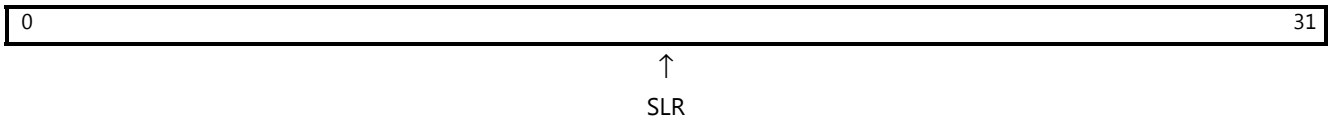

Figure 2-10: **SLR**

Table 2-16: **Stack Low Register (SLR)**

Bits	Name	Description	Reset Value
0:31	SLR	Stack Low Register	0x00000000

Stack High Register (SHR)

The Stack High Register stores the stack high limit use to detect stack underflow. When the address of a load or store instruction using the stack pointer (register R1) as rA is greater than the Stack High Register, a stack underflow occurs, causing a Stack Protection Violation exception if exceptions are enabled in MSR.

When read with the MFS instruction, the SHR is specified by setting Sa = 0x0802. Figure 2-11 illustrates the SHR register and Table 2-17 provides bit descriptions and reset values.

Note: The register is only implemented if stack protection is enabled by setting the parameter C_USE_STACK_PROTECTION to 1. If stack protection is not implemented, writing to the register has no effect.

Note: Stack protection is not available when the MMU is enabled (C_USE_MMU > 0). With the MMU page-based memory protection is provided through the UTLB instead.

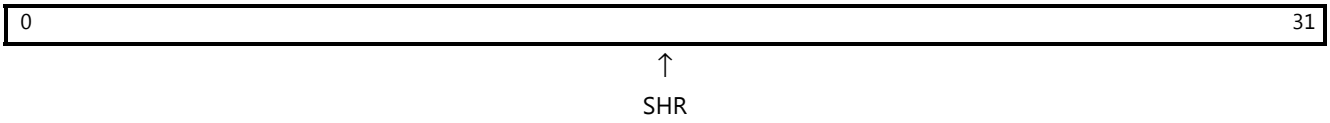

Figure 2-11: **SHR**

Table 2-17: **Stack High Register (SHR)**

Bits	Name	Description	Reset Value
0:31	SHR	Stack High Register	0xFFFFFFFF

Process Identifier Register (PID)

The Process Identifier Register is used to uniquely identify a software process during MMU address translation. It is controlled by the C_USE_MMU configuration option on MicroBlaze. The register is only implemented if C_USE_MMU is greater than 1 (User Mode) and C_AREA_OPTIMIZED is set to 0. When accessed with the MFS and MTS instructions, the PID is specified by setting Sa = 0x1000. The register is accessible according to the memory management special registers parameter C_MMU_TLB_ACCESS.

PID is also used when accessing a TLB entry:

- When writing Translation Look-Aside Buffer High (TLBHI) the value of PID is stored in the TID field of the TLB entry
- When reading TLBHI and MSR[UM] is not set, the value in the TID field is stored in PID

Figure 2-12 illustrates the PID register and Table 2-18 provides bit descriptions and reset values.

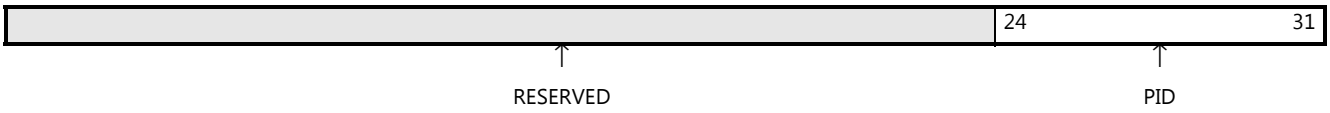

Figure 2-12: **PID**

Table 2-18: **Process Identifier Register (PID)**

Bits	Name	Description	Reset Value
0:23	Reserved		
24:31	PID	Used to uniquely identify a software process during MMU address translation. Read/Write	0x00

Zone Protection Register (ZPR)

The Zone Protection Register is used to override MMU memory protection defined in TLB entries. It is controlled by the C_USE_MMU configuration option on MicroBlaze. The register is only implemented if C_USE_MMU is greater than 1 (User Mode), C_AREA_OPTIMIZED is set to 0, and if the number of specified memory protection zones is greater than zero (C_MMU_ZONES > 0). The implemented register bits depend on the number of specified memory protection zones (C_MMU_ZONES). When accessed with the MFS and MTS instructions, the ZPR is specified by setting Sa = 0x1001. The register is accessible according to the memory management special registers parameter C_MMU_TLB_ACCESS. Figure 2-13 illustrates the ZPR register and Table 2-19 provides bit descriptions and reset values.

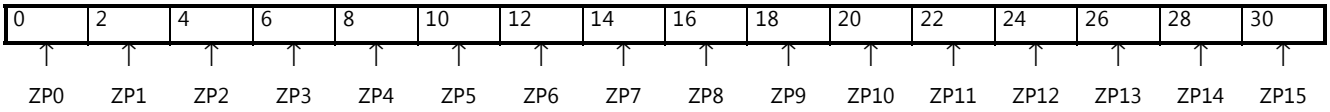

Figure 2-13: **ZPR**

Table 2-19: **Zone Protection Register (ZPR)**

Bits	Name	Description	Reset Value
0:1 2:3 ... 30:31	ZP0 ZP1 ... ZP15	Zone Protect User mode (MSR[UM] = 1): 00 = Override V in TLB entry. No access to the page is allowed 01 = No override. Use V, WR and EX from TLB entry 10 = No override. Use V, WR and EX from TLB entry 11 = Override WR and EX in TLB entry. Access the page as writable and executable Privileged mode (MSR[UM] = 0): 00 = No override. Use V, WR and EX from TLB entry 01 = No override. Use V, WR and EX from TLB entry 10 = Override WR and EX in TLB entry. Access the page as writable and executable 11 = Override WR and EX in TLB entry. Access the page as writable and executable Read/Write	0x00000000

Translation Look-Aside Buffer Low Register (TLBLO)

The Translation Look-Aside Buffer Low Register is used to access MMU Unified Translation Look-Aside Buffer (UTLB) entries. It is controlled by the C_USE_MMU configuration option on MicroBlaze. The register is only implemented if C_USE_MMU is greater than 1 (User Mode), and C_AREA_OPTIMIZED is set to 0. When accessed with the MFS and MTS instructions, the TLBLO is specified by setting Sa = 0x1003. When reading or writing TLBLO, the UTLB entry indexed by the TLBX register is accessed. The register is readable according to the memory management special registers parameter C_MMU_TLB_ACCESS.

The UTLB is reset on bit stream download (reset value is 0x00000000 for all TLBLO entries).

Note: The UTLB is **not** reset by the external reset inputs: Reset and Debug_Rst. This means that the entire UTLB must be initialized after reset, to avoid any stale data.

Figure 2-14 illustrates the TLBLO register and Table 2-20 provides bit descriptions and reset values.

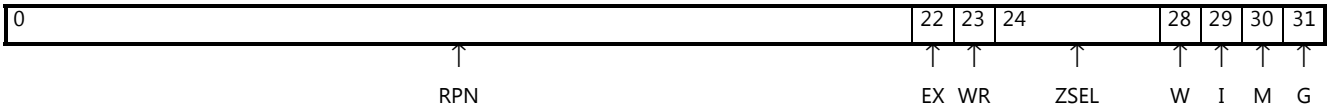

Figure 2-14: **TLBLO**

Table 2-20: **Translation Look-Aside Buffer Low Register (TLBLO)**

Bits	Name	Description	Reset Value
0:21	RPN	Real Page Number or Physical Page Number When a TLB hit occurs, this field is read from the TLB entry and is used to form the physical address. Depending on the value of the SIZE field, some of the RPN bits are not used in the physical address. Software must clear unused bits in this field to zero. Only defined when C_USE_MMU=3 (Virtual). Read/Write	0x000000
22	EX	Executable When bit is set to 1, the page contains executable code, and instructions can be fetched from the page. When bit is cleared to 0, instructions cannot be fetched from the page. Attempts to fetch instructions from a page with a clear EX bit cause an instruction-storage exception. Read/Write	0
23	WR	Writable When bit is set to 1, the page is writable and store instructions can be used to store data at addresses within the page. When bit is cleared to 0, the page is read-only (not writable). Attempts to store data into a page with a clear WR bit cause a data storage exception. Read/Write	0
24:27	ZSEL	Zone Select This field selects one of 16 zone fields (Z0-Z15) from the zone-protection register (ZPR). For example, if ZSEL 0x5, zone field Z5 is selected. The selected ZPR field is used to modify the access protection specified by the TLB entry EX and WR fields. It is also used to prevent access to a page by overriding the TLB V (valid) field. Read/Write	0x0
28	W	Write Through When the parameter C_DCACHE_USE_WRITEBACK is set to 1, this bit controls caching policy. A write-through policy is selected when set to 1, and a write-back policy is selected otherwise. This bit is fixed to 1, and write-through is always used, when C_DCACHE_USE_WRITEBACK is cleared to 0. Read/Write	0/1
29	I	Inhibit Caching When bit is set to 1, accesses to the page are not cached (caching is inhibited). When cleared to 0, accesses to the page are cacheable. Read/Write	0

Table 2-20: **Translation Look-Aside Buffer Low Register (TLBLO) *(Cont'd)***

Bits	Name	Description	Reset Value
30	M	Memory Coherent This bit is fixed to 0, because memory coherence is not implemented on MicroBlaze. Read Only	0
31	G	Guarded When bit is set to 1, speculative page accesses are not allowed (memory is guarded). When cleared to 0, speculative page accesses are allowed. The G attribute can be used to protect memory-mapped I/O devices from inappropriate instruction accesses. Read/Write	0

Translation Look-Aside Buffer High Register (TLBHI)

The Translation Look-Aside Buffer High Register is used to access MMU Unified Translation Look-Aside Buffer (UTLB) entries. It is controlled by the C_USE_MMU configuration option on MicroBlaze. The register is only implemented if C_USE_MMU is greater than 1 (User Mode), and C_AREA_OPTIMIZED is set to 0. When accessed with the MFS and MTS instructions, the TLBHI is specified by setting Sa = 0x1004. When reading or writing TLBHI, the UTLB entry indexed by the TLBX register is accessed. The register is readable according to the memory management special registers parameter C_MMU_TLB_ACCESS.

PID is also used when accessing a TLB entry:

- When writing TLBHI the value of PID is stored in the TID field of the TLB entry
- When reading TLBHI and MSR[UM] is not set, the value in the TID field is stored in PID

The UTLB is reset on bit stream download (reset value is 0x00000000 for all TLBHI entries).

Note: The UTLB is **not** reset by the external reset inputs: Reset and Debug_Rst.

Figure 2-15 illustrates the TLBHI register and Table 2-21 provides bit descriptions and reset values.

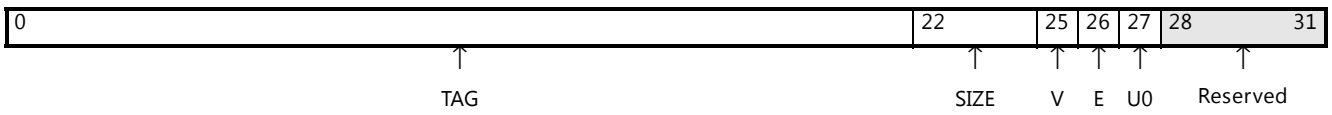

Figure 2-15: **TLBHI**

Table 2-21: **Translation Look-Aside Buffer High Register (TLBHI)**

Bits	Name	Description	Reset Value
0:21	TAG	TLB-entry tag Is compared with the page number portion of the virtual memory address under the control of the SIZE field. Read/Write	0x000000
22:24	SIZE	Size Specifies the page size. The SIZE field controls the bit range used in comparing the TAG field with the page number portion of the virtual memory address. The page sizes defined by this field are listed in Table 2-37. Read/Write	000
25	V	Valid When this bit is set to 1, the TLB entry is valid and contains a page-translation entry. When cleared to 0, the TLB entry is invalid. Read/Write	0
26	E	Endian When this bit is set to 1, the page is accessed as a big endian page. When cleared to 0, the page is accessed as a little endian page. The E bit only affects data read or data write accesses. Instruction accesses are not affected. The E bit is only implemented when the parameter C_USE_REORDER_INSTR is set to 1, otherwise it is fixed to 0. Read/Write	0
27	U0	User Defined This bit is fixed to 0, since there are no user defined storage attributes on MicroBlaze. Read Only	0
28:31	Reserved		

Translation Look-Aside Buffer Index Register (TLBX)

The Translation Look-Aside Buffer Index Register is used as an index to the Unified Translation Look-Aside Buffer (UTLB) when accessing the TLBLO and TLBHI registers. It is controlled by the C_USE_MMU configuration option on MicroBlaze. The register is only implemented if C_USE_MMU is greater than 1 (User Mode), and C_AREA_OPTIMIZED is set to 0. When accessed with the MFS and MTS instructions, the TLBX is specified by setting Sa = 0x1002. Figure 2-16 illustrates the TLBX register and Table 2-22 provides bit descriptions and reset values.

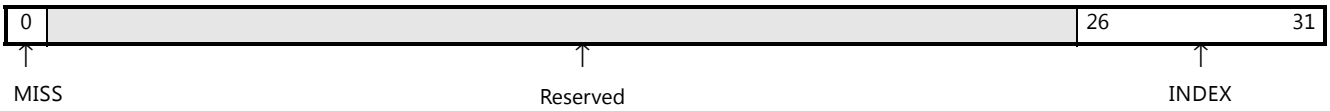

Figure 2-16: **TLBX**

Table 2-22: **Translation Look-Aside Buffer Index Register (TLBX)**

Bits	Name	Description	Reset Value
0	MISS	TLB Miss This bit is cleared to 0 when the TLBSX register is written with a virtual address, and the virtual address is found in a TLB entry. The bit is set to 1 if the virtual address is not found. It is also cleared when the TLBX register itself is written. Read Only Can be read if the memory management special registers parameter C_MMU_TLB_ACCESS > 0 (MINIMAL).	0
1:25	Reserved		
26:31	INDEX	TLB Index This field is used to index the Translation Look-Aside Buffer entry accessed by the TLBLO and TLBHI registers. The field is updated with a TLB index when the TLBSX register is written with a virtual address, and the virtual address is found in the corresponding TLB entry. Read/Write Can be read and written if the memory management special registers parameter C_MMU_TLB_ACCESS > 0 (MINIMAL).	000000

Translation Look-Aside Buffer Search Index Register (TLBSX)

The Translation Look-Aside Buffer Search Index Register is used to search for a virtual page number in the Unified Translation Look-Aside Buffer (UTLB). It is controlled by the C_USE_MMU configuration option on MicroBlaze. The register is only implemented if C_USE_MMU is greater than 1 (User Mode), and C_AREA_OPTIMIZED is set to 0. When written with the MTS instruction, the TLBSX is specified by setting Sa = 0x1005. Figure 2-17 illustrates the TLBSX register and Table 2-23 provides bit descriptions and reset values.

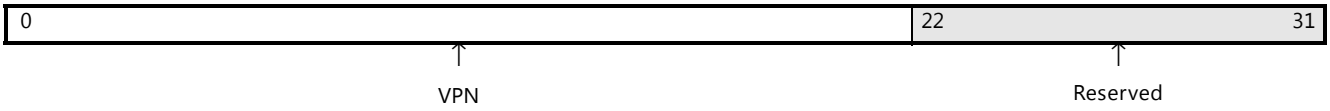

Figure 2-17: **TLBSX**

Table 2-23: **Translation Look-Aside Buffer Index Search Register (TLBSX)**

Bits	Name	Description	Reset Value
0:21	VPN	Virtual Page Number This field represents the page number portion of the virtual memory address. It is compared with the page number portion of the virtual memory address under the control of the SIZE field, in each of the Translation Look-Aside Buffer entries that have the V bit set to 1. If the virtual page number is found, the TLBX register is written with the index of the TLB entry and the MISS bit in TLBX is cleared to 0. If the virtual page number is not found in any of the TLB entries, the MISS bit in the TLBX register is set to 1. Write Only	
22:31	Reserved		

Processor Version Register (PVR)

The Processor Version Register is controlled by the C_PVR configuration option on MicroBlaze.

- When C_PVR is set to 0 (None) the processor does not implement any PVR and MSR[PVR]=0.
- When C_PVR is set to 1 (Basic), MicroBlaze implements only the first register: PVR0, and if set to 2 (Full), all 13 PVR registers (PVR0 to PVR12) are implemented.

When read with the MFS instruction the PVR is specified by setting Sa = 0x200*x*, with *x* being the register number between 0x0 and 0xB.

Table 2-24 through Table 2-35 provide bit descriptions and values.

Table 2-24: **Processor Version Register 0 (PVR0)**

Bits	Name	Description	Value
0	CFG	PVR implementation: 0 = Basic, 1 = Full	Based on C_PVR
1	BS	Use barrel shifter	C_USE_BARREL
2	DIV	Use divider	C_USE_DIV
3	MUL	Use hardware multiplier	C_USE_HW_MUL > 0 (None)
4	FPU	Use FPU	C_USE_FPU > 0 (None)
5	EXC	Use any type of exceptions	Based on C_*_EXCEPTION Also set if C_USE_MMU > 0 (None)
6	ICU	Use instruction cache	C_USE_ICACHE
7	DCU	Use data cache	C_USE_DCACHE
8	MMU	Use MMU	C_USE_MMU > 0 (None)
9	BTC	Use branch target cache	C_USE_BRANCH_TARGET_CACHE
10	ENDI	Selected endianness: Always 1 = Little endian	C_ENDIANNESS

Table 2-24: **Processor Version Register 0 (PVR0) *(Cont'd)***

Bits	Name	Description	Value
11	FT	Implement fault tolerant features	C_FAULT_TOLERANT
12	SPROT	Use stack protection	C_USE_STACK_PROTECTION
13	REORD	Implement reorder instructions	C_USE_REORDER_INSTR
14:15	Reserved		0
16:23	MBV	MicroBlaze release version code 0x19 = v8.40.b 0x1B = v9.0 0x1D = v9.1 0x1F = v9.2 0x20 = v9.3 0x21 = v9.4 0x22 = v9.5	Release Specific
24:31	USR1	User configured value 1	C_PVR_USER1

Table 2-25: **Processor Version Register 1 (PVR1)**

Bits	Name	Description	Value
0:31	USR2	User configured value 2	C_PVR_USER2

Table 2-26: **Processor Version Register 2 (PVR2)**

Bits	Name	Description	Value
0	DAXI	Data side AXI4 or ACE in use	C_D_AXI
1	DLMB	Data side LMB in use	C_D_LMB
2	IAXI	Instruction side AXI4 or ACE in use	C_I_AXI
3	ILMB	Instruction side LMB in use	C_I_LMB
4	IRQEDGE	Interrupt is edge triggered	C_INTERRUPT_IS_EDGE
5	IRQPOS	Interrupt edge is positive	C_EDGE_IS_POSITIVE
6	CEEXC	Generate bus exceptions for ECC correctable errors in LMB memory	C_ECC_USE_CE_EXCEPTION
7:8	Reserved		0
9	Reserved		1
10	ACE	Use ACE interconnect	C_INTERCONNECT = 3 (ACE)
11	AXI4DP	Data Peripheral AXI interface uses AXI4 protocol, with support for exclusive access	C_M_AXI_DP_EXCLUSIVE_ACCESS
12	FSL	Use extended AXI4-Stream instructions	C_USE_EXTENDED_FSL_INSTR
13	FSLEXC	Generate exception for AXI4-Stream control bit mismatch	C_FSL_EXCEPTION
14	MSR	Use msrset and msrclr instructions	C_USE_MSR_INSTR
15	PCMP	Use pattern compare and CLZ instructions	C_USE_PCMP_INSTR

Table 2-26: **Processor Version Register 2 (PVR2) *(Cont'd)***

Bits	Name	Description	Value
16	AREA	Select implementation to optimize area with lower instruction throughput	C_AREA_OPTIMIZED
17	BS	Use barrel shifter	C_USE_BARREL
18	DIV	Use divider	C_USE_DIV
19	MUL	Use hardware multiplier	C_USE_HW_MUL > 0 (None)
20	FPU	Use FPU	C_USE_FPU > 0 (None)
21	MUL64	Use 64-bit hardware multiplier	C_USE_HW_MUL = 2 (Mul64)
22	FPU2	Use floating point conversion and square root instructions	C_USE_FPU = 2 (Extended)
23	IMPEXC	Allow imprecise exceptions for ECC errors in LMB memory	C_IMPRECISE_EXCEPTIONS
24	Reserved		0
25	OP0EXC	Generate exception for 0x0 illegal opcode	C_OPCODE_0x0_ILLEGAL
26	UNEXC	Generate exception for unaligned data access	C_UNALIGNED_EXCEPTIONS
27	OPEXC	Generate exception for any illegal opcode	C_ILL_OPCODE_EXCEPTION
28	AXIDEXC	Generate exception for M_AXI_D error	C_M_AXI_D_BUS_EXCEPTION
29	AXIIEXC	Generate exception for M_AXI_I error	C_M_AXI_I_BUS_EXCEPTION
30	DIVEXC	Generate exception for division by zero or division overflow	C_DIV_ZERO_EXCEPTION
31	FPUEXC	Generate exceptions from FPU	C_FPU_EXCEPTION

Table 2-27: **Processor Version Register 3 (PVR3)**

Bits	Name	Description	Value
0	DEBUG	Use debug logic	C_DEBUG_ENABLED > 0
1	EXT_DEBUG	Use extended debug logic	C_DEBUG_ENABLED = 2 (Extended)
2	Reserved		
3:6	PCBRK	Number of PC breakpoints	C_NUMBER_OF_PC_BRK
7:9	Reserved		
10:12	RDADDR	Number of read address breakpoints	C_NUMBER_OF_RD_ADDR_BRK
13:15	Reserved		
16:18	WRADDR	Number of write address breakpoints	C_NUMBER_OF_WR_ADDR_BRK
19	Reserved		0
20:24	FSL	Number of AXI4-Stream links	C_FSL_LINKS
25:28	Reserved		
29:31	BTC_SIZE	Branch Target Cache size	C_BRANCH_TARGET_CACHE_SIZE

Table 2-28: **Processor Version Register 4 (PVR4)**

Bits	Name	Description	Value
0	ICU	Use instruction cache	C_USE_ICACHE
1:5	ICTS	Instruction cache tag size	C_ADDR_TAG_BITS
6	Reserved		1
7	ICW	Allow instruction cache write	C_ALLOW_ICACHE_WR
8:10	ICLL	The base two logarithm of the instruction cache line length	log2(C_ICACHE_LINE_LEN)
11:15	ICBS	The base two logarithm of the instruction cache byte size	log2(C_CACHE_BYTE_SIZE)
16	IAU	The instruction cache is used for all memory accesses within the cacheable range	C_ICACHE_ALWAYS_USED
17:18	Reserved		0
19:21	ICV	Instruction cache victims	0-3: C_ICACHE_VICTIMS = 0,2,4,8
22:23	ICS	Instruction cache streams	C_ICACHE_STREAMS
24	IFTL	Instruction cache tag uses distributed RAM	C_ICACHE_FORCE_TAG_LUTRAM
25	ICDW	Instruction cache data width	C_ICACHE_DATA_WIDTH > 0
26:31	Reserved		0

Table 2-29: **Processor Version Register 5 (PVR5)**

Bits	Name	Description	Value
0	DCU	Use data cache	C_USE_DCACHE
1:5	DCTS	Data cache tag size	C_DCACHE_ADDR_TAG
6	Reserved		1
7	DCW	Allow data cache write	C_ALLOW_DCACHE_WR
8:10	DCLL	The base two logarithm of the data cache line length	log2(C_DCACHE_LINE_LEN)
11:15	DCBS	The base two logarithm of the data cache byte size	log2(C_DCACHE_BYTE_SIZE)
16	DAU	The data cache is used for all memory accesses within the cacheable range	C_DCACHE_ALWAYS_USED
17	DWB	Data cache policy is write-back	C_DCACHE_USE_WRITEBACK
18	Reserved		0
19:21	DCV	Data cache victims	0-3: C_DCACHE_VICTIMS = 0,2,4,8
22:23	Reserved		0
24	DFTL	Data cache tag uses distributed RAM	C_DCACHE_FORCE_TAG_LUTRAM

Table 2-29: **Processor Version Register 5 (PVR5) *(Cont'd)***

Bits	Name	Description	Value
25	DCDW	Data cache data width	C_DCACHE_DATA_WIDTH > 0
26	AXI4DC	Data Cache AXI interface uses AXI4 protocol, with support for exclusive access	C_M_AXI_DC_EXCLUSIVE_ACCESS
27:31	Reserved		0

Table 2-30: **Processor Version Register 6 (PVR6)**

Bits	Name	Description	Value
0:31	ICBA	Instruction Cache Base Address	C_ICACHE_BASEADDR

Table 2-31: **Processor Version Register 7 (PVR7)**

Bits	Name	Description	Value
0:31	ICHA	Instruction Cache High Address	C_ICACHE_HIGHADDR

Table 2-32: **Processor Version Register 8 (PVR8)**

Bits	Name	Description	Value
0:31	DCBA	Data Cache Base Address	C_DCACHE_BASEADDR

Table 2-33: **Processor Version Register 9 (PVR9)**

Bits	Name	Description	Value
0:31	DCHA	Data Cache High Address	C_DCACHE_HIGHADDR

Table 2-34: **Processor Version Register 10 (PVR10)**

Bits	Name	Description	Value
0:7	ARCH	Target architecture: 0xF = Virtex-7, Defence Grade Virtex-7 Q 0x10 = Kintex™-7, Defence Grade Kintex-7 Q 0x11 = Artix™-7, Automotive Artix-7, Defence Grade Artix-7 Q 0x12 = Zynq™-7000, Automotive Zynq-7000, Defence Grade Zynq-7000 Q 0x13 = UltraScale™ Virtex 0x14 = UltraScale Kintex	Defined by parameter C_FAMILY
8:31	Reserved		0

Table 2-35: **Processor Version Register 11 (PVR11)**

Bits	Name	Description	Value
0:1	MMU	Use MMU: 0 = None 2 = Protection 1 = User Mode 3 = Virtual	C_USE_MMU
2:4	ITLB	Instruction Shadow TLB size	log2(C_MMU_ITLB_SIZE)
5:7	DTLB	Data Shadow TLB size	log2(C_MMU_DTLB_SIZE)
8:9	TLBACC	TLB register access: 0 = Minimal 2 = Write 1 = Read 3 = Full	C_MMU_TLB_ACCESS
10:14	ZONES	Number of memory protection zones	C_MMU_ZONES
15	PRIVINS	Privileged instructions: 0 = Full protection 1 = Allow stream instructions	C_MMU_PRIVILEGED_INSTR
16:16	Reserved	Reserved for future use	0
17:31	RSTMSR	Reset value for MSR	C_RESET_MSR

Table 2-36: **Processor Version Register 12 (PVR12)**

Bits	Name	Description	Value
0:31	VECTORS	Location of MicroBlaze vectors	C_BASE_VECTORS

Pipeline Architecture

MicroBlaze instruction execution is pipelined. For most instructions, each stage takes one clock cycle to complete. Consequently, the number of clock cycles necessary for a specific instruction to complete is equal to the number of pipeline stages, and one instruction is completed on every cycle. A few instructions require multiple clock cycles in the execute stage to complete. This is achieved by stalling the pipeline.

When executing from slower memory, instruction fetches may take multiple cycles. This additional latency directly affects the efficiency of the pipeline. MicroBlaze implements an instruction prefetch buffer that reduces the impact of such multi-cycle instruction memory latency. While the pipeline is stalled by a multi-cycle instruction in the execution stage, the prefetch buffer continues to load sequential instructions. When the pipeline resumes execution, the fetch stage can load new instructions directly from the prefetch buffer instead of waiting for the instruction memory access to complete. If instructions are modified during execution (e.g. with self-modifying code), the prefetch buffer should be emptied before executing the modified instructions, to ensure that it does not contain the old unmodified instructions. The recommended way to do this is using an MBAR instruction, although it is also possible to use a synchronizing branch instruction, for example BRI 4.

Three Stage Pipeline

With C_AREA_OPTIMIZED set to 1, the pipeline is divided into three stages to minimize hardware cost: Fetch, Decode, and Execute.

	cycle1	cycle2	cycle3	cycle4	cycle5	cycle6	cycle7
instruction 1	Fetch	Decode	Execute				
instruction 2		Fetch	Decode	Execute	Execute	Execute	
instruction 3			Fetch	Decode	Stall	Stall	Execute

Five Stage Pipeline

With C_AREA_OPTIMIZED set to 0, the pipeline is divided into five stages to maximize performance: Fetch (IF), Decode (OF), Execute (EX), Access Memory (MEM), and Writeback (WB).

	cycle1	cycle2	cycle3	cycle4	cycle5	cycle6	cycle7	cycle8	cycle9
instruction 1	IF	OF	EX	MEM	WB				
instruction 2		IF	OF	EX	MEM	MEM	MEM	WB	
instruction 3			IF	OF	EX	Stall	Stall	MEM	WB

Branches

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

Delay Slots

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

A delay slot must not contain the following instructions: IMM, branch, or break. Interrupts and external hardware breaks are deferred until after the delay slot branch has been completed.

Instructions that could cause recoverable exceptions (e.g. unaligned word or halfword load and store) are allowed in the delay slot. If an exception is caused in a delay slot the ESR[DS] bit is set, and the exception handler is responsible for returning the execution to the branch target (stored in the special purpose register BTR). If the ESR[DS] bit is set, register R17 is not valid (otherwise it contains the address following the instruction causing the exception).

Branch Target Cache

To improve branch performance, MicroBlaze provides a Branch Target Cache (BTC) coupled with a branch prediction scheme. With the BTC enabled, a correctly predicted immediate branch or return instruction incurs no overhead.

The BTC operates by saving the target address of each immediate branch and return instruction the first time the instruction is encountered. The next time it is encountered, it is usually found in the Branch Target Cache, and the Instruction Fetch Program Counter is then simply changed to the saved target address, in case the branch should be taken. Unconditional branches and return instructions are always taken, whereas conditional branches use branch prediction, to avoid taking a branch that should not have been taken and vice versa.

The BTC is cleared when a memory barrier (MBAR 0) or synchronizing branch (BRI 4) is executed. This also occurs when the memory barrier or synchronizing branch follows immediately after a branch instruction, even if that branch is taken. To avoid inadvertently clearing the BTC, the memory barrier or synchronizing branch should not be placed immediately after a branch instruction.

There are three cases where the branch prediction can cause a mispredict, namely:

- A conditional branch that should not have been taken, is actually taken,
- A conditional branch that should actually have been taken, is not taken,
- The target address of a return instruction is incorrect, which may occur when returning from a function called from different places in the code.

All of these cases are detected and corrected when the branch or return instruction reaches the execute stage, and the branch prediction bits or target address are updated in the BTC, to reflect the actual instruction behavior. This correction incurs a penalty of two clock cycles.

The size of the BTC can be selected with `C_BRANCH_TARGET_CACHE_SIZE`. The default recommended setting uses one block RAM, and provides 512 entries. When selecting 64 entries or below, distributed RAM is used to implement the BTC, otherwise block RAM is used.

When the BTC uses block RAM, and `C_FAULT_TOLERANT` is set to 1, block RAMs are protected by parity. In case of a parity error, the branch is not predicted. To avoid accumulating errors in this case, the BTC should be cleared periodically by a synchronizing branch.

The Branch Target Cache is available when `C_USE_BRANCH_TARGET_CACHE` is set to 1 and `C_AREA_OPTIMIZED` is set to 0.

Memory Architecture

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

Both instruction and data interfaces of MicroBlaze are default 32 bits wide and use big endian or little endian, bit-reversed format, depending on the selected endianness. MicroBlaze supports word, halfword, and byte accesses to data memory.

Data accesses must be aligned (word accesses must be on word boundaries, halfword on halfword boundaries), unless the processor is configured to support unaligned exceptions. All instruction accesses must be word aligned.

MicroBlaze prefetches instructions to improve performance, using the instruction prefetch buffer and (if enabled) instruction cache streams. To avoid attempts to prefetch instructions beyond the end of physical memory, which may cause an instruction bus error or a processor stall, instructions must not be located too close to the end of physical memory. The instruction prefetch buffer requires 16 bytes margin, and using instruction cache streams adds two additional cache lines (32, 64 or 128 bytes).

MicroBlaze does not separate data accesses to I/O and memory (it uses memory mapped I/O). The processor has up to three interfaces for memory accesses:

- Local Memory Bus (LMB)
- Advanced eXtensible Interface (AXI4) for peripheral access
- Advanced eXtensible Interface (AXI4) or AXI Coherency Extension (ACE) for cache access

The LMB memory address range must not overlap with AXI4 ranges.

The C_ENDIANNESS parameter is always set to little endian.

MicroBlaze has a single cycle latency for accesses to local memory (LMB) and for cache read hits, except with C_AREA_OPTIMIZED set to 1, when data side accesses and data cache read hits require two clock cycles, and with C_FAULT_TOLERANT set to 1, when byte writes and halfword writes to LMB normally require two clock cycles.

The data cache write latency depends on C_DCACHE_USE_WRITEBACK. When C_DCACHE_USE_WRITEBACK is set to 1, the write latency normally is one cycle (more if the cache needs to do memory accesses). When C_DCACHE_USE_WRITEBACK is cleared to 0, the write latency normally is two cycles (more if the posted-write buffer in the memory controller is full).

The MicroBlaze instruction and data caches can be configured to use 4, 8 or 16 word cache lines. When using a longer cache line, more bytes are prefetched, which generally improves performance for software with sequential access patterns. However, for software with a more random access pattern the performance can instead decrease for a given cache size. This is caused by a reduced cache hit rate due to fewer available cache lines.

For details on the different memory interfaces refer to Chapter 3, MicroBlaze Signal Interface Description.

Privileged Instructions

The following MicroBlaze instructions are privileged:

- GET, GETD, PUT, PUTD (except when explicitly allowed)
- WIC, WDC
- MTS
- MSRCLR, MSRSET (except when only the C bit is affected)
- BRK
- RTID, RTBD, RTED
- BRKI (except when jumping to physical address C_BASE_VECTORS + 0x8 or C_BASE_VECTORS + 0x18)
- SLEEP

Attempted use of these instructions when running in user mode causes a privileged instruction exception.

When setting the parameter C_MMU_PRIVILEGED_INSTR to 1, the instructions GET, GETD, PUT, and PUTD are not considered privileged, and can be executed when running in user mode. It is strongly discouraged to do this, unless absolutely necessary for performance reasons, since it allows application programs to interfere with each other.

There are six ways to leave user mode and virtual mode:

1. Hardware generated reset (including debug reset)
2. Hardware exception
3. Non-maskable break or hardware break
4. Interrupt
5. Executing "BRALID Re,C_BASE_VECTORS + 0x8" to perform a user vector exception
6. Executing the software break instructions "BRKI" jumping to physical address C_BASE_VECTORS + 0x8 or C_BASE_VECTORS + 0x18

In all of these cases, except hardware generated reset, the user mode and virtual mode status is saved in the MSR UMS and VMS bits.

Application (user-mode) programs transfer control to system-service routines (privileged mode programs) using the BRALID or BRKI instruction, jumping to physical address C_BASE_VECTORS + 0x8. Executing this instruction causes a system-call exception to occur. The exception handler determines which system-service routine to call and whether the calling application has permission to call that service. If permission is granted, the

exception handler performs the actual procedure call to the system-service routine on behalf of the application program.

The execution environment expected by the system-service routine requires the execution of prologue instructions to set up that environment. Those instructions usually create the block of storage that holds procedural information (the activation record), update and initialize pointers, and save volatile registers (registers the system-service routine uses). Prologue code can be inserted by the linker when creating an executable module, or it can be included as stub code in either the system-call interrupt handler or the system-library routines.

Returns from the system-service routine reverse the process described above. Epilog code is executed to unwind and deallocate the activation record, restore pointers, and restore volatile registers. The interrupt handler executes a return from exception instruction (RTED) to return to the application.

Virtual-Memory Management

Programs running on MicroBlaze use effective addresses to access a flat 4 GB address space. The processor can interpret this address space in one of two ways, depending on the translation mode:

- In real mode, effective addresses are used to directly access physical memory
- In virtual mode, effective addresses are translated into physical addresses by the virtual-memory management hardware in the processor

Virtual mode provides system software with the ability to relocate programs and data anywhere in the physical address space. System software can move inactive programs and data out of physical memory when space is required by active programs and data.

Relocation can make it appear to a program that more memory exists than is actually implemented by the system. This frees the programmer from working within the limits imposed by the amount of physical memory present in a system. Programmers do not need to know which physical-memory addresses are assigned to other software processes and hardware devices. The addresses visible to programs are translated into the appropriate physical addresses by the processor.

Virtual mode provides greater control over memory protection. Blocks of memory as small as 1 KB can be individually protected from unauthorized access. Protection and relocation enable system software to support multitasking. This capability gives the appearance of simultaneous or near-simultaneous execution of multiple programs.

In MicroBlaze, virtual mode is implemented by the memory-management unit (MMU), available when `C_USE_MMU` is set to 3 (Virtual) and `C_AREA_OPTIMIZED` is set to 0. The MMU controls effective-address to physical-address mapping and supports memory protection. Using these capabilities, system software can implement demand-paged virtual memory and other memory management schemes.

The MicroBlaze MMU implementation is based upon PowerPC™ 405. For details, see the *PowerPC Processor Reference Guide* (UG011) document.

The MMU features are summarized as follows:

- Translates effective addresses into physical addresses
- Controls page-level access during address translation
- Provides additional virtual-mode protection control through the use of zones
- Provides independent control over instruction-address and data-address translation and protection
- Supports eight page sizes: 1 kB, 4 kB, 16 kB, 64 kB, 256 kB, 1 MB, 4 MB, and 16 MB. Any combination of page sizes can be used by system software
- Software controls the page-replacement strategy

Real Mode

The processor references memory when it fetches an instruction and when it accesses data with a load or store instruction. Programs reference memory locations using a 32-bit effective address calculated by the processor. When real mode is enabled, the physical address is identical to the effective address and the processor uses it to access physical memory. After a processor reset, the processor operates in real mode. Real mode can also be enabled by clearing the VM bit in the MSR.

Physical-memory data accesses (loads and stores) are performed in real mode using the effective address. Real mode does not provide system software with virtual address translation, but the full memory access-protection is available, implemented when C_USE_MMU > 1 (User Mode) and C_AREA_OPTIMIZED = 0. Implementation of a real-mode memory manager is more straightforward than a virtual-mode memory manager. Real mode is often an appropriate solution for memory management in simple embedded environments, when access-protection is necessary, but virtual address translation is not required.

Virtual Mode

In virtual mode, the processor translates an effective address into a physical address using the process shown in Figure 2-18. Virtual mode can be enabled by setting the VM bit in the MSR.

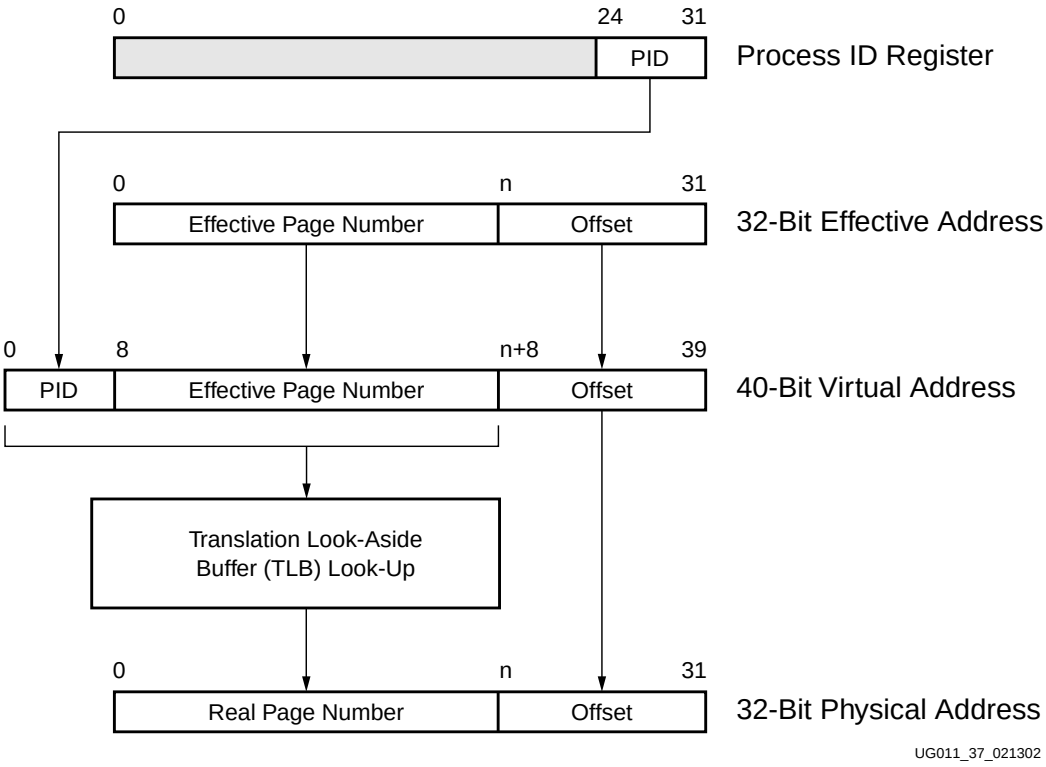

Figure 2-18: **Virtual-Mode Address Translation**

Each address shown in Figure 2-18 contains a page-number field and an offset field. The page number represents the portion of the address translated by the MMU. The offset represents the byte offset into a page and is not translated by the MMU. The virtual address consists of an additional field, called the process ID (PID), which is taken from the PID register (see Process-ID Register, page 33). The combination of PID and effective page number (EPN) is referred to as the virtual page number (VPN). The value n is determined by the page size, as shown in Table 2-37.

System software maintains a page-translation table that contains entries used to translate each virtual page into a physical page. The page size defined by a page translation entry determines the size of the page number and offset fields. For example, when a 4 kB page size is used, the page-number field is 20 bits and the offset field is 12 bits. The VPN in this case is 28 bits.

Then the most frequently used page translations are stored in the translation look-aside buffer (TLB). When translating a virtual address, the MMU examines the page-translation entries for a matching VPN (PID and EPN). Rather than examining all entries in the table, only entries contained in the processor TLB are examined. When a page-translation entry is found with a matching VPN, the corresponding physical-page number is read from the entry and combined with the offset to form the 32-bit physical address. This physical address is used by the processor to reference memory.

System software can use the PID to uniquely identify software processes (tasks, subroutines, threads) running on the processor. Independently compiled processes can operate in effective-address regions that overlap each other. This overlap must be resolved by system software if multitasking is supported. Assigning a PID to each process enables system software to resolve the overlap by relocating each process into a unique region of virtual-address space. The virtual-address space mappings enable independent translation of each process into the physical-address space.

Page-Translation Table

The page-translation table is a software-defined and software-managed data structure containing page translations. The requirement for software-managed page translation represents an architectural trade-off targeted at embedded-system applications. Embedded systems tend to have a tightly controlled operating environment and a well-defined set of application software. That environment enables virtual-memory management to be optimized for each embedded system in the following ways:

- The page-translation table can be organized to maximize page-table search performance (also called table walking) so that a given page-translation entry is located quickly. Most general-purpose processors implement either an indexed page table (simple search method, large page-table size) or a hashed page table (complex search method, small page-table size). With software table walking, any hybrid organization can be employed that suits the particular embedded system. Both the page-table size and access time can be optimized.

- Independent page sizes can be used for application modules, device drivers, system service routines, and data. Independent page-size selection enables system software to more efficiently use memory by reducing fragmentation (unused memory). For example, a large data structure can be allocated to a 16 MB page and a small I/O device-driver can be allocated to a 1 KB page.
- Page replacement can be tuned to minimize the occurrence of missing page translations. As described in the following section, the most-frequently used page translations are stored in the translation look-aside buffer (TLB). Software is responsible for deciding which translations are stored in the TLB and which translations are replaced when a new translation is required. The replacement strategy can be tuned to avoid thrashing, whereby page-translation entries are constantly being moved in and out of the TLB. The replacement strategy can also be tuned to prevent replacement of critical-page translations, a process sometimes referred to as page locking.

The unified 64-entry TLB, managed by software, caches a subset of instruction and data page-translation entries accessible by the MMU. Software is responsible for reading entries from the page-translation table in system memory and storing them in the TLB. The following section describes the unified TLB in more detail. Internally, the MMU also contains shadow TLBs for instructions and data, with sizes configurable by `C_MMU_ITLB_SIZE` and `C_MMU_DTLB_SIZE` respectively.

These shadow TLBs are managed entirely by the processor (transparent to software) and are used to minimize access conflicts with the unified TLB.

Translation Look-Aside Buffer

The translation look-aside buffer (TLB) is used by the MicroBlaze MMU for address translation when the processor is running in virtual mode, memory protection, and storage control. Each entry within the TLB contains the information necessary to identify a virtual page (PID and effective page number), specify its translation into a physical page, determine the protection characteristics of the page, and specify the storage attributes associated with the page.

The MicroBlaze TLB is physically implemented as three separate TLBs:

- Unified TLB—The UTLB contains 64 entries and is pseudo-associative. Instruction-page and data-page translation can be stored in any UTLB entry. The initialization and management of the UTLB is controlled completely by software.
- Instruction Shadow TLB—The ITLB contains instruction page-translation entries and is fully associative. The page-translation entries stored in the ITLB represent the most-recently accessed instruction-page translations from the UTLB. The ITLB is used to minimize contention between instruction translation and UTLB-update operations. The initialization and management of the ITLB is controlled completely by hardware and is transparent to software.

- Data Shadow TLB—The DTLB contains data page-translation entries and is fully associative. The page-translation entries stored in the DTLB represent the most-recently accessed data-page translations from the UTLB. The DTLB is used to minimize contention between data translation and UTLB-update operations. The initialization and management of the DTLB is controlled completely by hardware and is transparent to software.

Figure 2-19 provides the translation flow for TLB.

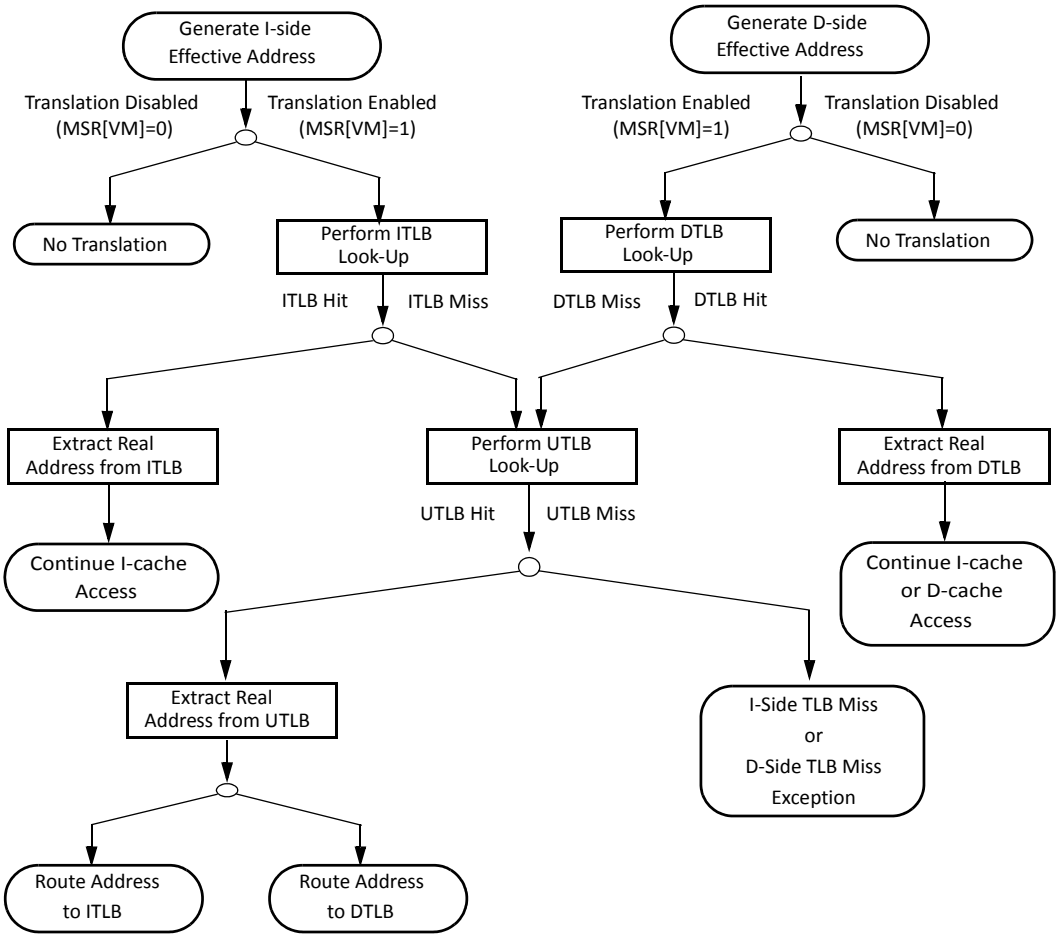

Figure 2-19: **TLB Address Translation Flow**

TLB Entry Format

Figure 2-20 shows the format of a TLB entry. Each TLB entry is 68 bits and is composed of two portions: TLBLO (also referred to as the data entry), and TLBHI (also referred to as the tag entry).

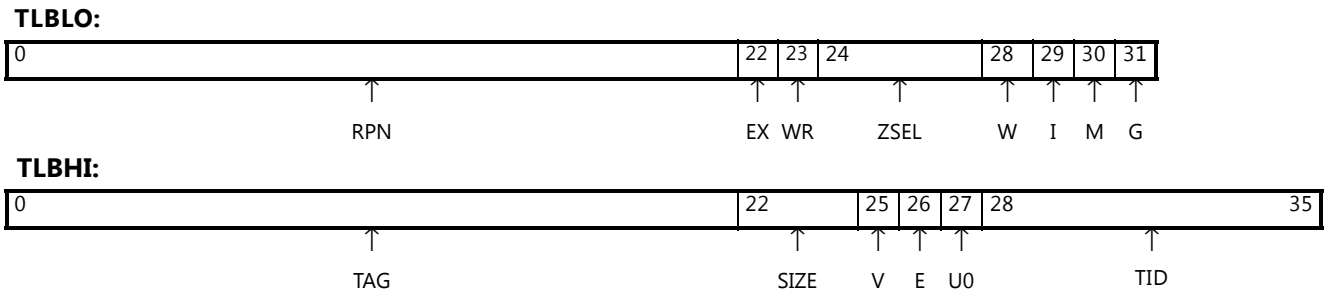

Figure 2-20: **TLB Entry Format**

The TLB entry contents are described in Table 2-20, page 35 and Table 2-21, page 37.

The fields within a TLB entry are categorized as follows:

- Virtual-page identification (TAG, SIZE, V, TID)—These fields identify the page-translation entry. They are compared with the virtual-page number during the translation process.
- Physical-page identification (RPN, SIZE)—These fields identify the translated page in physical memory.
- Access control (EX, WR, ZSEL)—These fields specify the type of access allowed in the page and are used to protect pages from improper accesses.
- Storage attributes (W, I, M, G, E, U0)—These fields specify the storage-control attributes, such as caching policy for the data cache (write-back or write-through), whether a page is cacheable, and how bytes are ordered (endianness).

Table 2-37 shows the relationship between the TLB-entry `SIZE` field and the translated page size. This table also shows how the page size determines which address bits are involved in a tag comparison, which address bits are used as a page offset, and which bits in the physical page number are used in the physical address.

Table 2-37: **Page-Translation Bit Ranges by Page Size**

Page Size	SIZE (TLBHI Field)	Tag Comparison Bit Range	Page Offset	Physical Page Number	RPN Bits Clear to 0
1 KB	000	TAG[0:21] - Address[0:21]	Address[22:31]	RPN[0:21]	-
4 KB	001	TAG[0:19] - Address[0:19]	Address[20:31]	RPN[0:19]	20:21
16 KB	010	TAG[0:17] - Address[0:17]	Address[18:31]	RPN[0:17]	18:21
64 KB	011	TAG[0:15] - Address[0:15]	Address[16:31]	RPN[0:15]	16:21
256 KB	100	TAG[0:13] - Address[0:13]	Address[14:31]	RPN[0:13]	14:21
1 MB	101	TAG[0:11] - Address[0:11]	Address[12:31]	RPN[0:11]	12:21
4 MB	110	TAG[0:9] - Address[0:9]	Address[10:31]	RPN[0:9]	10:21
16 MB	111	TAG[0:7] - Address[0:7]	Address[8:31]	RPN[0:7]	8:21

TLB Access

When the MMU translates a virtual address (the combination of PID and effective address) into a physical address, it first examines the appropriate shadow TLB for the page translation entry. If an entry is found, it is used to access physical memory. If an entry is not found, the MMU examines the UTLB for the entry. A delay occurs each time the UTLB must be accessed due to a shadow TLB miss. The miss latency ranges from 2-32 cycles. The DTLB has priority over the ITLB if both simultaneously access the UTLB.

Figure 2-21, page 60 shows the logical process the MMU follows when examining a page-translation entry in one of the shadow TLBs or the UTLB. All valid entries in the TLB are checked.

A TLB hit occurs when all of the following conditions are met by a TLB entry:

- The entry is valid
- The TAG field in the entry matches the effective address EPN under the control of the SIZE field in the entry
- The TID field in the entry matches the PID

If any of the above conditions are not met, a TLB miss occurs. A TLB miss causes an exception, described as follows:

A TID value of 0x00 causes the MMU to ignore the comparison between the TID and PID. Only the TAG and EA[EPN] are compared. A TLB entry with TID=0x00 represents a process-independent translation. Pages that are accessed globally by all processes should be assigned a TID value of 0x00. A PID value of 0x00 does not identify a process that can access any page. When PID=0x00, a page-translation hit only occurs when TID=0x00. It is possible for software to load the TLB with multiple entries that match an EA[EPN] and PID combination. However, this is considered a programming error and results in undefined behavior.

When a hit occurs, the MMU reads the RPN field from the corresponding TLB entry. Some or all of the bits in this field are used, depending on the value of the SIZE field (see Table 2-37). For example, if the SIZE field specifies a 256 kB page size, RPN[0:13] represents the physical page number and is used to form the physical address. RPN[14:21] is not used, and software must clear those bits to 0 when initializing the TLB entry. The remainder of the physical address is taken from the page-offset portion of the EA. If the page size is 256 kB, the 32-bit physical address is formed by concatenating RPN[0:13] with bits14:31 of the effective address.

Prior to accessing physical memory, the MMU examines the TLB-entry access-control fields. These fields indicate whether the currently executing program is allowed to perform the requested memory access.

If access is allowed, the MMU checks the storage-attribute fields to determine how to access the page. The storage-attribute fields specify the caching policy for memory accesses.

TLB Access Failures

A TLB-access failure causes an exception to occur. This interrupts execution of the instruction that caused the failure and transfers control to an interrupt handler to resolve the failure. A TLB access can fail for two reasons:

- A matching TLB entry was not found, resulting in a TLB miss
- A matching TLB entry was found, but access to the page was prevented by either the storage attributes or zone protection

When an interrupt occurs, the processor enters real mode by clearing MSR[VM] to 0. In real mode, all address translation and memory-protection checks performed by the MMU are disabled. After system software initializes the UTLB with page-translation entries, management of the MicroBlaze UTLB is usually performed using interrupt handlers running in real mode.

Figure 2-21 diagrams the general process for examining a TLB entry.

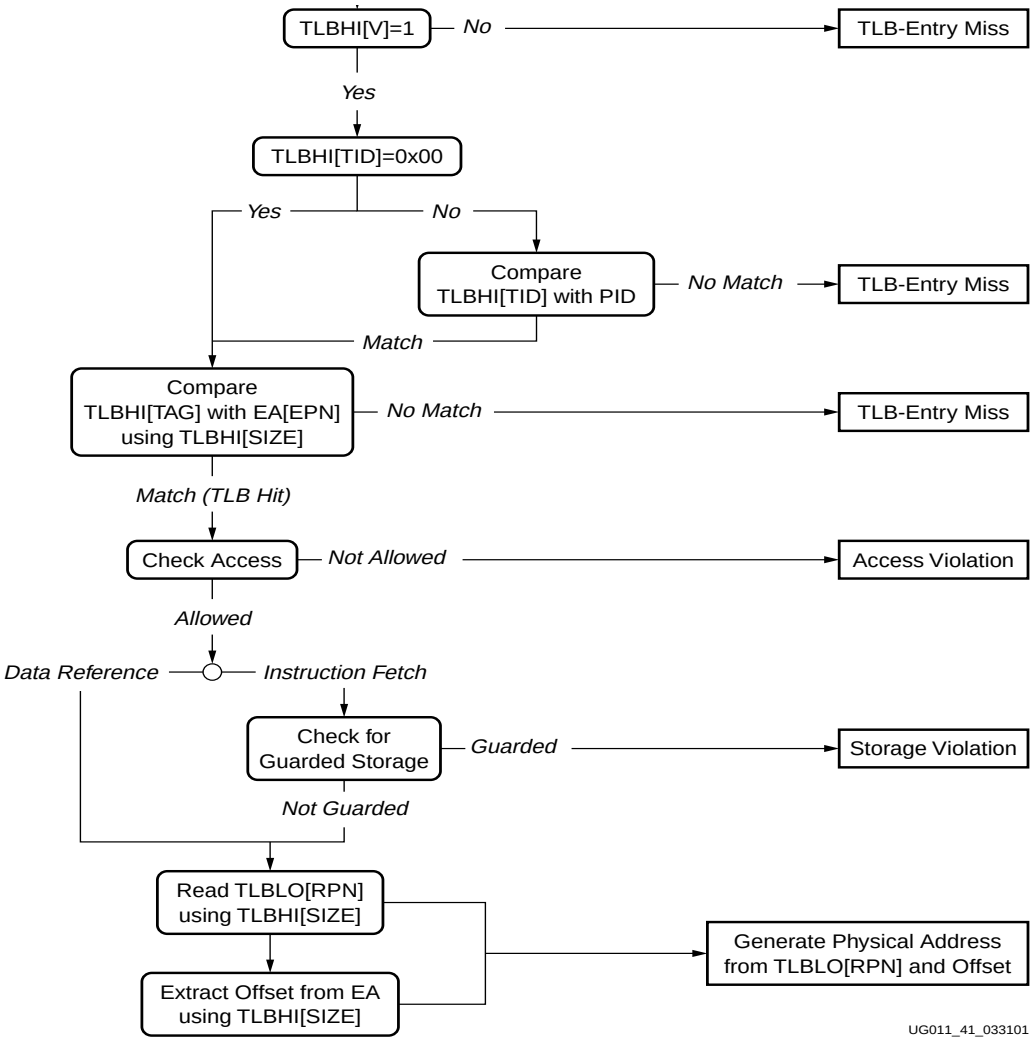

Figure 2-21: **General Process for Examining a TLB Entry**

The following sections describe the conditions under which exceptions occur due to TLB access failures.

Data-Storage Exception

When virtual mode is enabled, (MSR[VM]=1), a data-storage exception occurs when access to a page is not permitted for any of the following reasons:

- From user mode:
  - The TLB entry specifies a zone field that prevents access to the page (ZPR[Zn]=00). This applies to load and store instructions.
  - The TLB entry specifies a read-only page (TLBLO[WR]=0) that is not otherwise overridden by the zone field (ZPR[Zn], 11). This applies to store instructions.

- From privileged mode:
  - The TLB entry specifies a read-only page (TLBLO[WR]=0) that is not otherwise overridden by the zone field (ZPR[Zn], 10 and ZPR[Zn], 11). This applies to store instructions.

Instruction-Storage Exception

When virtual mode is enabled, (MSR[VM]=1), an instruction-storage exception occurs when access to a page is not permitted for any of the following reasons:

- From user mode:
  - The TLB entry specifies a zone field that prevents access to the page (ZPR[Zn]=00).
  - The TLB entry specifies a non-executable page (TLBLO[EX]=0) that is not otherwise overridden by the zone field (ZPR[Zn], 11).
  - The TLB entry specifies a guarded-storage page (TLBLO[G]=1).
- From privileged mode:
  - The TLB entry specifies a non-executable page (TLBLO[EX]=0) that is not otherwise overridden by the zone field (ZPR[Zn], 10 and ZPR[Zn], 11).
  - The TLB entry specifies a guarded-storage page (TLBLO[G]=1).

Data TLB-Miss Exception

When virtual mode is enabled (MSR[VM]=1) a data TLB-miss exception occurs if a valid, matching TLB entry was not found in the TLB (shadow and UTLB). Any load or store instruction can cause a data TLB-miss exception.

Instruction TLB-Miss Exception

When virtual mode is enabled (MSR[VM]=1) an instruction TLB-miss exception occurs if a valid, matching TLB entry was not found in the TLB (shadow and UTLB). Any instruction fetch can cause an instruction TLB-miss exception.

Access Protection

System software uses access protection to protect sensitive memory locations from improper access. System software can restrict memory accesses for both user-mode and privileged-mode software. Restrictions can be placed on reads, writes, and instruction fetches. Access protection is available when virtual protected mode is enabled.

Access control applies to instruction fetches, data loads, and data stores. The TLB entry for a virtual page specifies the type of access allowed to the page. The TLB entry also specifies a zone-protection field in the zone-protection register that is used to override the access controls specified by the TLB entry.

TLB Access-Protection Controls

Each TLB entry controls three types of access:

- Process—Processes are protected from unauthorized access by assigning a unique process ID (PID) to each process. When system software starts a user-mode application, it loads the PID for that application into the PID register. As the application executes, memory addresses are translated using only TLB entries with a TID field in Translation Look-Aside Buffer High (TLBHI) that matches the PID. This enables system software to restrict accesses for an application to a specific area in virtual memory.
  A TLB entry with TID=0x00 represents a process-independent translation. Pages that are accessed globally by all processes should be assigned a TID value of 0x00.
- Execution—The processor executes instructions only if they are fetched from a virtual page marked as executable (TLBLO[EX]=1). Clearing TLBLO[EX] to 0 prevents execution of instructions fetched from a page, instead causing an instruction-storage interrupt (ISI) to occur. The ISI does not occur when the instruction is fetched, but instead occurs when the instruction is executed. This prevents speculatively fetched instructions that are later discarded (rather than executed) from causing an ISI.

  The zone-protection register can override execution protection.
- Read/Write—Data is written only to virtual pages marked as writable (TLBLO[WR]=1). Clearing TLBLO[WR] to 0 marks a page as read-only. An attempt to write to a read-only page causes a data-storage interrupt (DSI) to occur.

  The zone-protection register can override write protection.

TLB entries cannot be used to prevent programs from reading pages. In virtual mode, zone protection is used to read-protect pages. This is done by defining a no-access-allowed zone (ZPR[Zn] = 00) and using it to override the TLB-entry access protection. Only programs running in user mode can be prevented from reading a page. Privileged programs always have read access to a page.

Zone Protection

Zone protection is used to override the access protection specified in a TLB entry. Zones are an arbitrary grouping of virtual pages with common access protection. Zones can contain any number of pages specifying any combination of page sizes. There is no requirement for a zone to contain adjacent pages.

The zone-protection register (ZPR) is a 32-bit register used to specify the type of protection override applied to each of 16 possible zones. The protection override for a zone is encoded in the ZPR as a 2-bit field. The 4-bit zone-select field in a TLB entry (TLBLO[ZSEL]) selects one of the 16 zone fields from the ZPR (Z0–Z15). For example, zone Z5 is selected when ZSEL = 0101.

Changing a zone field in the ZPR applies a protection override across all pages in that zone. Without the ZPR, protection changes require individual alterations to each page translation entry within the zone.

Unimplemented zones (when C_MMU_ZONES < 16) are treated as if they contained 11.

UTLB Management

The UTLB serves as the interface between the processor MMU and memory-management software. System software manages the UTLB to tell the MMU how to translate virtual addresses into physical addresses. When a problem occurs due to a missing translation or an access violation, the MMU communicates the problem to system software using the exception mechanism. System software is responsible for providing interrupt handlers to correct these problems so that the MMU can proceed with memory translation.

Software reads and writes UTLB entries using the MFS and MTS instructions, respectively. These instructions use the TLBX register index (numbered 0 to 63) corresponding to one of the 64 entries in the UTLB. The tag and data portions are read and written separately, so software must execute two MFS or MTS instructions to completely access an entry. The UTLB is searched for a specific translation using the TLBSX register. TLBSX locates a translation using an effective address and loads the corresponding UTLB index into the TLBX register.

Individual UTLB entries are invalidated using the MTS instruction to clear the valid bit in the tag portion of a TLB entry (TLBHI[V]).

When C_FAULT_TOLERANT is set to 1, the UTLB block RAM is protected by parity. In case of a parity error, a TLB miss exception occurs. To avoid accumulating errors in this case, each entry in the UTLB should be periodically invalidated.

Recording Page Access and Page Modification

Software management of virtual-memory poses several challenges:

- In a virtual-memory environment, software and data often consume more memory than is physically available. Some of the software and data pages must be stored outside physical memory, such as on a hard drive, when they are not used. Ideally, the most-frequently used pages stay in physical memory and infrequently used pages are stored elsewhere.
- When pages in physical-memory are replaced to make room for new pages, it is important to know whether the replaced (old) pages were modified. If they were modified, they must be saved prior to loading the replacement (new) pages. If the old pages were not modified, the new pages can be loaded without saving the old pages.
- A limited number of page translations are kept in the UTLB. The remaining translations must be stored in the page-translation table. When a translation is not found in the UTLB (due to a miss), system software must decide which UTLB entry to discard so that

the missing translation can be loaded. It is desirable for system software to replace infrequently used translations rather than frequently used translations.

Solving the above problems in an efficient manner requires keeping track of page accesses and page modifications. MicroBlaze does not track page access and page modification in hardware. Instead, system software can use the TLB-miss exceptions and the data-storage exception to collect this information. As the information is collected, it can be stored in a data structure associated with the page-translation table.

Page-access information is used to determine which pages should be kept in physical memory and which are replaced when physical-memory space is required. System software can use the valid bit in the TLB entry (TLBHI[V]) to monitor page accesses. This requires page translations be initialized as not valid (TLBHI[V]=0) to indicate they have not been accessed. The first attempt to access a page causes a TLB-miss exception, either because the UTLB entry is marked not valid or because the page translation is not present in the UTLB. The TLB-miss handler updates the UTLB with a valid translation (TLBHI[V]=1). The set valid bit serves as a record that the page and its translation have been accessed. The TLB-miss handler can also record the information in a separate data structure associated with the page-translation entry.

Page-modification information is used to indicate whether an old page can be overwritten with a new page or the old page must first be stored to a hard disk. System software can use the write-protection bit in the TLB entry (TLBLO[WR]) to monitor page modification. This requires page translations be initialized as read-only (TLBLO[WR]=0) to indicate they have not been modified. The first attempt to write data into a page causes a data-storage exception, assuming the page has already been accessed and marked valid as described above. If software has permission to write into the page, the data-storage handler marks the page as writable (TLBLO[WR]=1) and returns. The set write-protection bit serves as a record that a page has been modified. The data-storage handler can also record this information in a separate data structure associated with the page-translation entry.

Tracking page modification is useful when virtual mode is first entered and when a new process is started.

Reset, Interrupts, Exceptions, and Break

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

The relative priority starting with the highest is:

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

Table 2-38 defines the memory address locations of the associated vectors and the hardware enforced register file locations for return addresses. Each vector allocates two addresses to allow full address range branching (requires an `IMM` followed by a `BRAI` instruction). Normally the vectors start at address 0x00000000, but the parameter `C_BASE_VECTORS` can be used to locate them anywhere in memory.

The address range 0x28 to 0x4F is reserved for future software support by Xilinx. Allocating these addresses for user applications is likely to conflict with future releases of SDK support software.

Table 2-38: **Vectors and Return Address Register File Location**

Event	Vector Address	Register File Return Address
Reset	C_BASE_VECTORS + 0x00000000 - C_BASE_VECTORS + 0x00000004	-
User Vector (Exception)	C_BASE_VECTORS + 0x00000008 - C_BASE_VECTORS + 0x0000000C	Rx
Interrupt[1]	C_BASE_VECTORS + 0x00000010 - C_BASE_VECTORS + 0x00000014	R14
Break: Non-maskable hardware	C_BASE_VECTORS + 0x00000018 - C_BASE_VECTORS + 0x0000001C	R16
Break: Hardware		
Break: Software		
Hardware Exception	C_BASE_VECTORS + 0x00000020 - C_BASE_VECTORS + 0x00000024	R17 or BTR
Reserved by Xilinx for future use	C_BASE_VECTORS + 0x00000028 - C_BASE_VECTORS + 0x0000004F	-

1. With low-latency interrupt mode, the vector address is supplied by the Interrupt Controller.

All of these events will clear the reservation bit, used together with the LWX and SWX instructions to implement mutual exclusion, such as semaphores and spinlocks.

Reset

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

Equivalent Pseudocode

```
PC ← C_BASE_VECTORS + 0x00000000
MSR ← C_RESET_MSR (see "MicroBlaze Core Configurability" in Chapter 3)
EAR ← 0; ESR ← 0; FSR ← 0
PID ← 0; ZPR ← 0; TLBX ← 0
Reservation ← 0
```

Hardware Exceptions

MicroBlaze can be configured to trap the following internal error conditions: illegal instruction, instruction and data bus error, and unaligned access. The divide exception can only be enabled if the processor is configured with a hardware divider (C_USE_DIV=1). When configured with a hardware floating point unit (C_USE_FPU>0), it can also trap the following floating point specific exceptions: underflow, overflow, float division-by-zero, invalid operation, and denormalized operand error.

When configured with a hardware Memory Management Unit, it can also trap the following memory management specific exceptions: Illegal Instruction Exception, Data Storage Exception, Instruction Storage Exception, Data TLB Miss Exception, and Instruction TLB Miss Exception.

A hardware exception causes MicroBlaze to flush the pipeline and branch to the hardware exception vector (address C_BASE_VECTORS + 0x20). The execution stage instruction in the exception cycle is not executed.

The exception also updates the general purpose register R17 in the following manner:

- For the MMU exceptions (Data Storage Exception, Instruction Storage Exception, Data TLB Miss Exception, Instruction TLB Miss Exception) the register R17 is loaded with the appropriate program counter value to re-execute the instruction causing the exception upon return. The value is adjusted to return to a preceding IMM instruction, if any. If the exception is caused by an instruction in a branch delay slot, the value is adjusted to return to the branch instruction, including adjustment for a preceding IMM instruction, if any.
- For all other exceptions the register R17 is loaded with the program counter value of the subsequent instruction, unless the exception is caused by an instruction in a branch delay slot. If the exception is caused by an instruction in a branch delay slot, the

1. Reset input controlled by the debugger via MDM.

ESR[DS] bit is set. In this case the exception handler should resume execution from the branch target address stored in BTR.

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

The VM and UM bits in MSR are automatically reverted from VMS and UMS when executing the `RTED`, `RTBD`, and `RTID` instructions.

Exception Priority

When two or more exceptions occur simultaneously, they are handled in the following order, from the highest priority to the lowest:

- Instruction Bus Exception
- Instruction TLB Miss Exception
- Instruction Storage Exception
- Illegal Opcode Exception
- Privileged Instruction Exception or Stack Protection Violation Exception
- Data TLB Miss Exception
- Data Storage Exception
- Unaligned Exception
- Data Bus Exception
- Divide Exception
- FPU Exception
- Stream Exception

Exception Causes

- Stream Exception

  The AXI4-Stream exception is caused by executing a `get` or `getd` instruction with the 'e' bit set to '1' when there is a control bit mismatch.

- Instruction Bus Exception

  The instruction bus exception is caused by errors when reading data from memory.

  - The instruction peripheral AXI4 interface (M_AXI_IP) exception is caused by an error response on `M_AXI_IP_RRESP`.
  - The instruction cache AXI4 interface (M_AXI_IC) is caused by an error response on `M_AXI_IC_RRESP`. The exception can only occur when `C_ICACHE_ALWAYS_USED` is set

to 1 and the cache is turned off, or if the MMU Inhibit Caching bit is set for the address. In all other cases the response is ignored.

- The instructions side local memory (ILMB) can only cause instruction bus exception when either an uncorrectable error occurs in the LMB memory, as indicated by the IUE signal, or C_ECC_USE_CE_EXCEPTION is set to 1 and a correctable error occurs in the LMB memory, as indicated by the ICE signal.

- Illegal Opcode Exception

  The illegal opcode exception is caused by an instruction with an invalid major opcode (bits 0 through 5 of instruction). Bits 6 through 31 of the instruction are not checked. Optional processor instructions are detected as illegal if not enabled. If the optional feature C_OPCODE_0x0_ILLEGAL is enabled, an illegal opcode exception is also caused if the instruction is equal to 0x00000000.

- Data Bus Exception

  The data bus exception is caused by errors when reading data from memory or writing data to memory.

  - The data peripheral AXI4 interface (M_AXI_DP) exception is caused by an error response on M_AXI_DP_RRESP or M_AXI_DP_BRESP.
  - The data cache AXI4 interface (M_AXI_DC) exception is caused by:
    - An error response on M_AXI_DC_RRESP or M_AXI_DC_BRESP,
    - OKAY response on M_AXI_DC_RRESP in case of an exclusive access using LWX.

    The exception can only occur when C_DCACHE_ALWAYS_USED is set to 1 and the cache is turned off, when an exclusive access using LWX or SWX is performed, or if the MMU Inhibit Caching bit is set for the address. In all other cases the response is ignored.

  - The data side local memory (DLMB) can only cause instruction bus exception when either an uncorrectable error occurs in the LMB memory, as indicated by the DUE signal, or C_ECC_USE_CE_EXCEPTION is set to 1 and a correctable error occurs in the LMB memory, as indicated by the DCE signal. An error can occur for all read accesses, and for byte and halfword write accesses.

- Unaligned Exception

  The unaligned exception is caused by a word access where the address to the data bus has bits 30 or 31 set, or a half-word access with bit 31 set.

- Divide Exception

  The divide exception is caused by an integer division (idiv or idivu) where the divisor is zero, or by a signed integer division (idiv) where overflow occurs (-2147483648 / -1).

- FPU Exception

  An FPU exception is caused by an underflow, overflow, divide-by-zero, illegal operation, or denormalized operand occurring with a floating point instruction.

  - Underflow occurs when the result is denormalized.
  - Overflow occurs when the result is not-a-number (NaN).
  - The divide-by-zero FPU exception is caused by the rA operand to fdiv being zero when rB is not infinite.
  - Illegal operation is caused by a signaling NaN operand or by illegal infinite or zero operand combinations.

- Privileged Instruction Exception

  The Privileged Instruction exception is caused by an attempt to execute a privileged instruction in User Mode.

- Stack Protection Violation Exception

  A Stack Protection Violation exception is caused by executing a load or store instruction using the stack pointer (register R1) as rA with an address outside the stack boundaries defined by the special Stack Low and Stack High registers, causing a stack overflow or a stack underflow.

- Data Storage Exception

  The Data Storage exception is caused by an attempt to access data in memory that results in a memory-protection violation.

- Instruction Storage Exception

  The Instruction Storage exception is caused by an attempt to access instructions in memory that results in a memory-protection violation.

- Data TLB Miss Exception

  The Data TLB Miss exception is caused by an attempt to access data in memory, when a valid Translation Look-Aside Buffer entry is not present, and virtual protected mode is enabled.

- Instruction TLB Miss Exception

  The Instruction TLB Miss exception is caused by an attempt to access instructions in memory, when a valid Translation Look-Aside Buffer entry is not present, and virtual protected mode is enabled.

Should an Instruction Bus Exception, Illegal Opcode Exception or Data Bus Exception occur when C_FAULT_TOLERANT is set to 1, and an exception is in progress (i.e. MSR[EIP] set and MSR[EE] cleared), the pipeline is halted, and the external signal MB_Error is set.

Imprecise Exceptions

Normally all exceptions in MicroBlaze are precise, meaning that any instructions in the pipeline after the instruction causing an exception are invalidated, and have no effect.

When C_IMPRECISE_EXCEPTIONS is set to 1 (ECC) an Instruction Bus Exception or Data Bus Exception caused by ECC errors in LMB memory is not precise, meaning that a subsequent memory access instruction in the pipeline may be executed. If this behavior is acceptable, the maximum frequency can be improved by setting this parameter to 1.

Equivalent Pseudocode

```
ESR[DS] ← exception in delay slot
if ESR[DS] then
   BTR ← branch target PC
   if MMU exception then
      if branch preceded by IMM then
         r17 ← PC - 8
      else
         r17 ← PC - 4
   else
      r17 ← invalid value
else if MMU exception then
   if instruction preceded by IMM then
      r17 ← PC - 4
   else
      r17 ← PC
else
   r17 ← PC + 4
PC ← C_BASE_VECTORS + 0x00000020
MSR[EE] ← 0, MSR[EIP]← 1
MSR[UMS] ← MSR[UM], MSR[UM] ← 0, MSR[VMS] ← MSR[VM], MSR[VM] ← 0
ESR[EC] ← exception specific value
ESR[ESS]← exception specific value
EAR ← exception specific value
FSR ← exception specific value
Reservation ← 0
```

Breaks

There are two kinds of breaks:

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

Hardware Breaks

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

A normal hardware break (that is, the Ext_BRK input port) is only handled when MSR[BIP] and MSR[EIP] are set to 0 (that is, there is no break or exception in progress). The Break In Progress flag disables interrupts. A non-maskable break (that is, the Ext_NM_BRK input port) is always handled immediately.

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

The Ext_BRK signal must be kept asserted until the break has occurred, and deasserted before the RTBD instruction is executed. The Ext_NM_BRK signal must only be asserted one clock cycle.

Software Breaks

To perform a software break, use the brk and brki instructions. Refer to Chapter 5, MicroBlaze Instruction Set Architecture for detailed information on software breaks.

As a special case, when C_USE_DEBUG is set, and "brki rD, 0x18" is executed, a software breakpoint is signaled to the debugger, e.g. the Xilinx Microprocessor Debugger (XMD) tool, irrespective of the value of C_BASE_VECTORS.

Latency

The time it takes MicroBlaze to enter a break service routine from the time the break occurs depends on the instruction currently in the execution stage and the latency to the memory storing the break vector.

Equivalent Pseudocode

```
r16 ← PC
PC ← C_BASE_VECTORS + 0x00000018
MSR[BIP] ← 1
MSR[UMS] ← MSR[UM], MSR[UM] ← 0, MSR[VMS] ← MSR[VM], MSR[VM] ← 0
Reservation ← 0
```

Interrupt

MicroBlaze supports one external interrupt source (connected to the Interrupt input port). The processor only reacts to interrupts if the Interrupt Enable (IE) bit in the Machine Status

Register (MSR) is set to 1. On an interrupt, the instruction in the execution stage completes while the instruction in the decode stage is replaced by a branch to the interrupt vector. This is either address C_BASE_VECTORS + 0x10, or with low-latency interrupt mode, the address supplied by the Interrupt Controller.

The interrupt return address (the PC associated with the instruction in the decode stage at the time of the interrupt) is automatically loaded into general purpose register R14. In addition, the processor also disables future interrupts by clearing the IE bit in the MSR. The IE bit is automatically set again when executing the RTID instruction.

Interrupts are ignored by the processor if either of the break in progress (BIP) or exception in progress (EIP) bits in the MSR are set to 1.

By using the parameter C_INTERRUPT_IS_EDGE, the external interrupt can either be set to level-sensitive or edge-sensitive:

- When using level-sensitive interrupts, the Interrupt input must remain set until MicroBlaze has taken the interrupt, and jumped to the interrupt vector. Software must clear the interrupt before returning from the interrupt handler. If not, the interrupt is taken again, as soon as interrupts are enabled when returning from the interrupt handler.
- When using edge-sensitive interrupts, MicroBlaze detects and latches the Interrupt input edge, which means that the input only needs to be asserted one clock cycle. The interrupt input can remain asserted, but must be deasserted at least one clock cycle before a new interrupt can be detected. The latching of an edge sensitive interrupt is independent of the IE bit in MSR. Should an interrupt occur while the IE bit is 0, it will immediately be serviced when the IE bit is set to 1.

Low-latency Interrupt Mode

A low-latency interrupt mode is available, which allows the Interrupt Controller to directly supply the interrupt vector for each individual interrupt (via the Interrupt_Address input port).

The address of each fast interrupt handler must be passed to the Interrupt Controller when initializing the interrupt system. When a particular interrupt occurs, this address is supplied by the Interrupt Controller, which allows MicroBlaze to directly jump to the handler code.

With this mode, MicroBlaze also directly sends the appropriate interrupt acknowledge to the Interrupt Controller (via the Interrupt_Ack output port), although it is still the responsibility of the Interrupt Service Routine to acknowledge level sensitive interrupts at the source.

This information allows the Interrupt Controller to acknowledge interrupts appropriately, both for level-sensitive and edge-triggered interrupt.

To inform the Interrupt Controller of the interrupt handling events, `Interrupt_Ack` is set to:

- 01 - when MicroBlaze jumps to the interrupt handler code,
- 10 - when the RTID instruction is executed to return from interrupt,
- 11 - when MSR[IE] is changed from 0 to 1, which enables interrupts again.

The `Interrupt_Ack` output port is active during one clock cycle, and is then reset to 00.

Latency

The time it takes MicroBlaze to enter an Interrupt Service Routine (ISR) from the time an interrupt occurs, depends on the configuration of the processor and the latency of the memory controller storing the interrupt vectors. If MicroBlaze is configured to have a hardware divider, the largest latency happens when an interrupt occurs during the execution of a division instruction.

With low-latency interrupt mode, the time to enter the ISR is significantly reduced, since the interrupt vector for each individual interrupt is directly supplied by the Interrupt Controller. With compiler support for fast interrupts, there is no need for a common ISR at all. Instead, the ISR for each individual interrupt will be directly called, and the compiler takes care of saving and restoring registers used by the ISR.

Equivalent Pseudocode

```
r14 ← PC
if C_USE_INTERRUPT = 2
  PC ← Interrupt_Address
  Interrupt_Ack ← 01
else
  PC ← C_BASE_VECTORS + 0x00000010
MSR[IE] ← 0
MSR[UMS] ← MSR[UM], MSR[UM] ← 0, MSR[VMS] ← MSR[VM], MSR[VM] ← 0
Reservation ← 0
```

User Vector (Exception)

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

Pseudocode

```
rx ← PC
PC ← C_BASE_VECTORS + 0x00000008
MSR[UMS] ← MSR[UM], MSR[UM] ← 0, MSR[VMS] ← MSR[VM], MSR[VM] ← 0
Reservation ← 0
```

Instruction Cache

Overview

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

The instruction cache has the following features:

- Direct mapped (1-way associative)
- User selectable cacheable memory address range
- Configurable cache and tag size
- Caching over AXI4 interface (M_AXI_IC)
- Option to use 4, 8 or 16 word cache-line
- Cache on and off controlled using a bit in the MSR
- Optional WIC instruction to invalidate instruction cache lines
- Optional stream buffers to improve performance by speculatively prefetching instructions
- Optional victim cache to improve performance by saving evicted cache lines
- Optional parity protection that invalidates cache lines if a Block RAM bit error is detected
- Optional data width selection to either use 32 bits, an entire cache line, or 512 bits

General Instruction Cache Functionality

When the instruction cache is used, the memory address space is split into two segments: a cacheable segment and a non-cacheable segment. The cacheable segment is determined by two parameters: `C_ICACHE_BASEADDR` and `C_ICACHE_HIGHADDR`. All addresses within this range correspond to the cacheable address segment. All other addresses are non-cacheable.

The cacheable segment size must be 2^N, where N is a positive integer. The range specified by `C_ICACHE_BASEADDR` and `C_ICACHE_HIGHADDR` must comprise a complete power-of-two range, such that range = 2^N and the N least significant bits of `C_ICACHE_BASEADDR` must be zero.

The cacheable instruction address consists of two parts: the cache address, and the tag address. The MicroBlaze instruction cache can be configured from 64 bytes to 64 kB. This corresponds to a cache address of between 6 and 16 bits. The tag address together with the cache address should match the full address of cacheable memory. When selecting cache

sizes below 2 kB, distributed RAM is used to implement the Tag RAM and Instruction RAM. Distributed RAM is always used to implement the Tag RAM, when setting the parameter C_ICACHE_FORCE_TAG_LUTRAM to 1. This parameter is only available with cache size 8 kB and less for 4 word cache-lines, with 16 kB and less for 8 word cache-lines, and with 32 kB and less for 16 word cache-lines.

For example: in a MicroBlaze configured with C_ICACHE_BASEADDR= 0x00300000, C_ICACHE_HIGHADDR=0x0030ffff, C_CACHE_BYTE_SIZE=4096, C_ICACHE_LINE_LEN=8, and C_ICACHE_FORCE_TAG_LUTRAM=0; the cacheable memory of 64 kB uses 16 bits of byte address, and the 4 kB cache uses 12 bits of byte address, thus the required address tag width is: 16-12=4 bits. The total number of block RAM primitives required in this configuration is: 2 RAMB16 for storing the 1024 instruction words, and 1 RAMB16 for 128 cache line entries, each consisting of: 4 bits of tag, 8 word-valid bits, 1 line-valid bit. In total 3 RAMB16 primitives.

Figure 2-22, page 75 shows the organization of Instruction Cache.

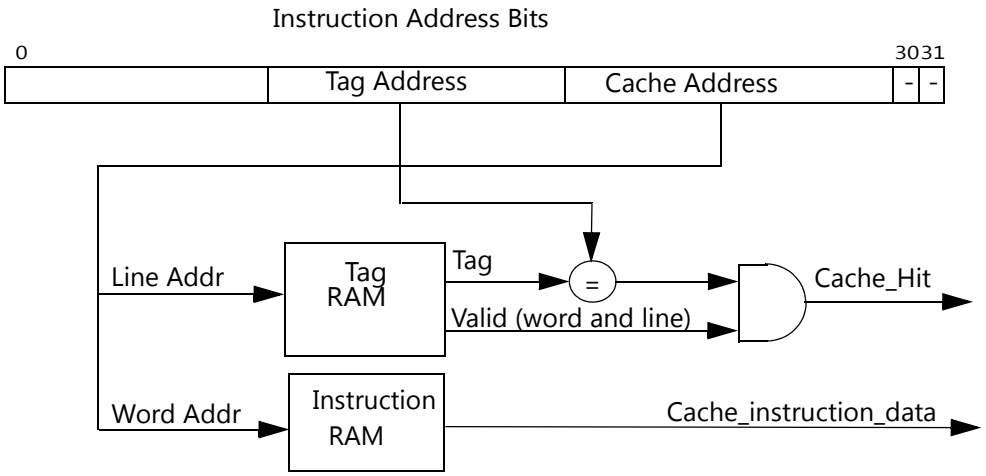

Figure 2-22: **Instruction Cache Organization**

Instruction Cache Operation

For every instruction fetched, the instruction cache detects if the instruction address belongs to the cacheable segment. If the address is non-cacheable, the cache controller ignores the instruction and lets the M_AXI_IP or LMB complete the request. If the address is cacheable, a lookup is performed on the tag memory to check if the requested address is currently cached. The lookup is successful if: the word and line valid bits are set, and the tag address matches the instruction address tag segment. On a cache miss, the cache controller requests the new instruction over the instruction AXI4 interface (M_AXI_IC), and waits for the memory controller to return the associated cache line.

C_ICACHE_DATA_WIDTH determines the bus data width, either 32 bits, an entire cache line (128, 256 or 512 bits), or 512 bits.

When C_FAULT_TOLERANT is set to 1, a cache miss also occurs if a parity error is detected in a tag or instruction Block RAM.

The instruction cache issues burst accesses for the AXI4 interface when 32-bit data width is used, otherwise single accesses are used.

Stream Buffers

When stream buffers are enabled, by setting the parameter C_ICACHE_STREAMS to 1, the cache will speculatively fetch cache lines in advance in sequence following the last requested address, until the stream buffer is full. The stream buffer can hold up to two cache lines. Should the processor subsequently request instructions from a cache line prefetched by the stream buffer, which occurs in linear code, they are immediately available.

The stream buffer often improves performance, since the processor generally has to spend less time waiting for instructions to be fetched from memory.

C_ICACHE_DATA_WIDTH determines the amount of data transferred from the stream buffer each clock cycle, either 32 bits or an entire cache line.

To be able to use instruction cache stream buffers, area optimization must not be enabled.

Victim Cache

The victim cache is enabled by setting the parameter C_ICACHE_VICTIMS to 2, 4 or 8. This defines the number of cache lines that can be stored in the victim cache. Whenever a cache line is evicted from the cache, it is saved in the victim cache. By saving the most recent lines they can be fetched much faster, should the processor request them, thereby improving performance. If the victim cache is not used, all evicted cache lines must be read from memory again when they are needed.

C_ICACHE_DATA_WIDTH determines the amount of data transferred from/to the victim cache each clock cycle, either 32 bits or an entire cache line.

Note that to be able to use the victim cache, area optimization must not be enabled.

Instruction Cache Software Support

MSR Bit

The ICE bit in the MSR provides software control to enable and disable caches.

The contents of the cache are preserved by default when the cache is disabled. You can invalidate cache lines using the WIC instruction or using the hardware debug logic of MicroBlaze.

WIC Instruction

The optional WIC instruction (`C_ALLOW_ICACHE_WR=1`) is used to invalidate cache lines in the instruction cache from an application. For a detailed description, refer to Chapter 5, MicroBlaze Instruction Set Architecture.

The WIC instruction can also be used together with parity protection to periodically invalidate entries the cache, to avoid accumulating errors.

Data Cache

Overview

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

- Direct mapped (1-way associative)
- Write-through or Write-back
- User selectable cacheable memory address range
- Configurable cache size and tag size
- Caching over AXI4 interface (M_AXI_DC)
- Option to use 4, 8 or 16 word cache-lines
- Cache on and off controlled using a bit in the MSR
- Optional WDC instruction to invalidate or flush data cache lines
- Optional victim cache with write-back to improve performance by saving evicted cache lines
- Optional parity protection for write-through cache that invalidates cache lines if a Block RAM bit error is detected
- Optional data width selection to either use 32 bits, an entire cache line, or 512 bits

General Data Cache Functionality

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

The cacheable segment size must be 2^N, where N is a positive integer. The range specified by `C_DCACHE_BASEADDR` and `C_DCACHE_HIGHADDR` must comprise a complete power-of-two range, such that range = 2^N and the N least significant bits of `C_DCACHE_BASEADDR` must be zero.

Figure 2-23 shows the Data Cache Organization.

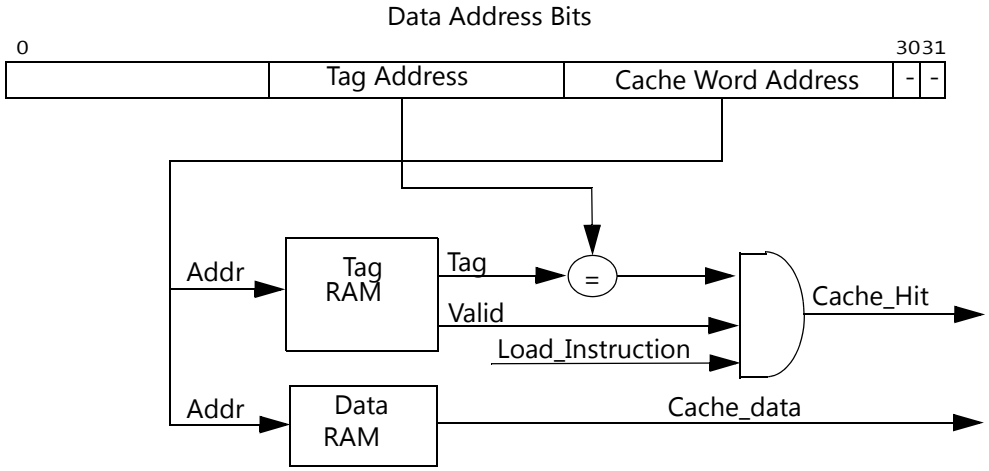

Figure 2-23: **Data Cache Organization**

The cacheable data address consists of two parts: the cache address, and the tag address. The MicroBlaze data cache can be configured from 64 bytes to 64 kB. This corresponds to a cache address of between 6 and 16 bits. The tag address together with the cache address should match the full address of cacheable memory. When selecting cache sizes below 2 kB, distributed RAM is used to implement the Tag RAM and Data RAM, except that block RAM is always used for the Data RAM when C_AREA_OPTIMIZED is set and C_DCACHE_USE_WRITEBACK is not set. Distributed RAM is always used to implement the Tag RAM, when setting the parameter C_DCACHE_FORCE_TAG_LUTRAM to 1. This parameter is only available with cache size 8 kB and less for 4 word cache-lines, with 16 kB and less for 8 word cache-lines, and with 32 kB and less for 16 word cache-lines.

For example, in a MicroBlaze configured with C_DCACHE_BASEADDR=0x00400000, C_DCACHE_HIGHADDR=0x00403fff, C_DCACHE_BYTE_SIZE=2048, C_DCACHE_LINE_LEN=4, and C_DCACHE_FORCE_TAG_LUTRAM=0; the cacheable memory of 16 kB uses 14 bits of byte address, and the 2 kB cache uses 11 bits of byte address, thus the required address tag width is 14-11=3 bits. The total number of block RAM primitives required in this configuration is 1 RAMB16 for storing the 512 data words, and 1 RAMB16 for 128 cache line entries, each consisting of 3 bits of tag, 4 word-valid bits, 1 line-valid bit. In total, 2 RAMB16 primitives.

Data Cache Operation

The caching policy used by the MicroBlaze data cache, write-back or write-through, is determined by the parameter C_DCACHE_USE_WRITEBACK. When this parameter is set, a write-back protocol is implemented, otherwise write-through is implemented. However, when configured with an MMU (C_USE_MMU > 1, C_AREA_OPTIMIZED = 0, C_DCACHE_USE_WRITEBACK = 1), the caching policy in virtual mode is determined by the W storage attribute in the TLB entry, whereas write-back is used in real mode.

With the write-back protocol, a store to an address within the cacheable range always updates the cached data. If the target address word is not in the cache (that is, the access is a cache miss), and the location in the cache contains data that has not yet been written to memory (the cache location is dirty), the old data is written over the data AXI4 interface (M_AXI_DC) to external memory before updating the cache with the new data. If only a single word needs to be written, a single word write is used, otherwise a burst write is used. For byte or halfword stores, in case of a cache miss, the address is first requested over the data AXI4 interface, while a word store only updates the cache.

With the write-through protocol, a store to an address within the cacheable range generates an equivalent byte, halfword, or word write over the data AXI4 interface to external memory. The write also updates the cached data if the target address word is in the cache (that is, the write is a cache hit). A write cache-miss does not load the associated cache line into the cache.

Provided that the cache is enabled a load from an address within the cacheable range triggers a check to determine if the requested data is currently cached. If it is (that is, on a cache hit) the requested data is retrieved from the cache. If not (that is, on a cache miss) the address is requested over the data AXI4 interface using a burst read, and the processor pipeline stalls until the cache line associated to the requested address is returned from the external memory controller.

The parameter `C_DCACHE_DATA_WIDTH` determines the bus data width, either 32 bits, an entire cache line (128, 256 or 512 bits), or 512 bits.

When `C_FAULT_TOLERANT` is set to 1 and write-through protocol is used, a cache miss also occurs if a parity error is detected in the tag or data Block RAM.

All types of accesses issued by the data cache AXI4 interface are summarized in Table 2-39.

Table 2-39: **Data Cache Interface Accesses**

Policy	State	Direction	Access Type
Write-through	Cache Enabled	Read	Burst for 32-bit interface non-exclusive access and exclusive access with ACE enabled, single access otherwise
		Write	Single access
	Cache Disabled	Read	Burst for 32-bit interface exclusive access with ACE enabled, single access otherwise
		Write	Single access
Write-back	Cache Enabled	Read	Burst for 32-bit interface, single access otherwise
		Write	Burst for 32-bit interface cache lines with more than one valid word, a single access otherwise
	Cache Disabled	Read	Burst for 32-bit interface non-exclusive access, discarding all but the desired data, a single access otherwise
		Write	Single access

Victim Cache

The victim cache is enabled by setting the parameter C_DCACHE_VICTIMS to 2, 4 or 8. This defines the number of cache lines that can be stored in the victim cache. Whenever a complete cache line is evicted from the cache, it is saved in the victim cache. By saving the most recent lines they can be fetched much faster, should the processor request them, thereby improving performance. If the victim cache is not used, all evicted cache lines must be read from memory again when they are needed.

With the AXI4 interface, C_DCACHE_DATA_WIDTH determines the amount of data transferred from/to the victim cache each clock cycle, either 32 bits or an entire cache line.

Note that to be able to use the victim cache, write-back must be enabled and area optimization must not be enabled.

Data Cache Software Support

MSR Bit

The DCE bit in the MSR controls whether or not the cache is enabled. When disabling caches the user must ensure that all the prior writes within the cacheable range have been completed in external memory before reading back over M_AXI_DP. This can be done by writing to a semaphore immediately before turning off caches, and then in a loop poll until it has been written.

The contents of the cache are preserved when the cache is disabled.

WDC Instruction

The optional WDC instruction (C_ALLOW_DCACHE_WR=1) is used to invalidate or flush cache lines in the data cache from an application. For a detailed description, please refer to Chapter 5, MicroBlaze Instruction Set Architecture.

The WDC instruction can also be used together with parity protection to periodically invalidate entries the cache, to avoid accumulating errors.

Floating Point Unit (FPU)

Overview

The MicroBlaze floating point unit is based on the IEEE 754-1985 standard:

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

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

- Denormalized(1) operands are not supported. A hardware floating point operation on a denormalized number returns a quiet NaN and sets the sticky denormalized operand error bit in FSR; see "Floating Point Status Register (FSR)" on page 30
- A denormalized result is stored as a signed 0 with the underflow bit set in FSR. This method is commonly referred to as Flush-to-Zero (FTZ)
- An operation on a quiet NaN returns the fixed NaN: 0xFFC00000, rather than one of the NaN operands
- Overflow as a result of a floating point operation always returns signed ∞

Format

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

1. 1-bit *sign*
2. 8-bit biased *exponent*
3. 23-bit *fraction* (a.k.a. mantissa or significand)

1. Numbers that are so close to 0, that they cannot be represented with full precision, that is, any number *n* that falls in the following ranges: ($1.17549*10^{-38} > n > 0$), or ($0 > n > -1.17549 * 10^{-38}$)

The fields are stored in a 32 bit word as defined in Figure 2-24:

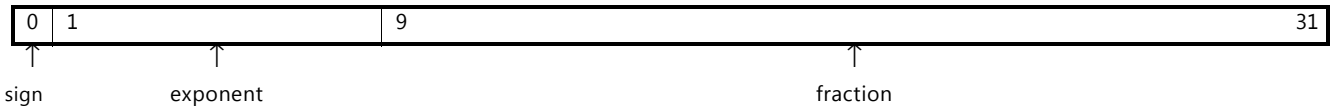

Figure 2-24: **IEEE 754 Single Precision Format**

The value of a floating point number *v* in MicroBlaze has the following interpretation:

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

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

Rounding

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

Operations

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

Arithmetic

The FPU implements the following floating point operations:

- addition, fadd
- subtraction, fsub
- multiplication, fmul
- division, fdiv
- square root, fsqrt (available if `C_USE_FPU = 2, EXTENDED`)

Comparison

The FPU implements the following floating point comparisons:

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

Conversion

The FPU implements the following conversions (available if `C_USE_FPU = 2, EXTENDED`):

- convert from signed integer to floating point, flt
- convert from floating point to signed integer, fint

Exceptions

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

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

Software Support

The SDK compiler system, based on GCC, provides support for the Floating Point Unit compliant with the MicroBlaze API. Compiler flags are automatically added to the GCC command line based on the type of FPU present in the system, when using SDK.

All double-precision operations are emulated in software. Be aware that the xil_printf() function does not support floating-point output. The standard C library printf() and related functions do support floating-point output, but will increase the program code size.

Libraries and Binary Compatibility

The SDK compiler system only includes software floating point C runtime libraries. To take advantage of the hardware FPU, the libraries must be recompiled with the appropriate compiler switches.

For all cases where separate compilation is used, it is very important that you ensure the consistency of FPU compiler flags throughout the build.

Operator Latencies

The latencies of the various operations supported by the FPU are listed in Chapter 5, "MicroBlaze Instruction Set Architecture." The FPU instructions are not pipelined, so only one operation can be ongoing at any time.

C Language Programming

To gain maximum benefit from the FPU without low-level assembly-language programming, it is important to consider how the C compiler will interpret your source code. Very often the same algorithm can be expressed in many different ways, and some are more efficient than others.

Immediate Constants

Floating-point constants in C are double-precision by default. When using a single-precision FPU, careless coding may result in double-precision software emulation routines being used instead of the native single-precision instructions. To avoid this, explicitly specify (by cast or suffix) that immediate constants in your arithmetic expressions are single-precision values.

For example:

```
float x = 0.0;
...
x += (float)1.0; /* float addition */
x += 1.0F;       /* alternative to above */
x += 1.0;        /* warning - uses double addition! */
```

Note that the GNU C compiler can be instructed to treat all floating-point constants as single-precision (contrary to the ANSI C standard) by supplying the compiler flag -fsingle-precision-constants.

Avoid unnecessary casting

While conversions between floating-point and integer formats are supported in hardware by the FPU, when C_USE_FPU is set to 2 (Extended), it is still best to avoid them when possible.

The following "bad" example calculates the sum of squares of the integers from 1 to 10 using floating-point representation:

```
float sum, t;
int i;
sum = 0.0f;
for (i = 1; i <= 10; i++) {
  t = (float)i;
  sum += t * t;
}
```

The above code requires a cast from an integer to a float on each loop iteration. This can be rewritten as:

```
float sum, t;
int i;
t = sum = 0.0f;
for(i = 1; i <= 10; i++) {
  t += 1.0f;
  sum += t * t;
}
```

Note that the compiler is not at liberty to perform this optimization in general, as the two code fragments above may give different results in some cases (for example, very large t).

Square root runtime library function

The standard C runtime math library functions operate using double-precision arithmetic. When using a single-precision FPU, calls to the square root functions (sqrt()) result in inefficient emulation routines being used instead of FPU instructions:

```
#include <math.h>
...
float x=-1.0F;
...
x = sqrt(x); /* uses double precision */
```

Here the math.h header is included to avoid a warning message from the compiler.

When used with single-precision data types, the result is a cast to double, a runtime library call is made (which does not use the FPU) and then a truncation back to float is performed.

The solution is to use the non-ANSI function sqrtf() instead, which operates using single precision and can be carried out using the FPU. For example:

```
#include <math.h>
...
float x=-1.0F;
...
x = sqrtf(x); /* uses single precision */
```

Note that when compiling this code, the compiler flag -fno-math-errno (in addition to -mhard-float and -mxl-float-sqrt) must be used, to ensure that the compiler does not generate unnecessary code to handle error conditions by updating the errno variable.

Stream Link Interfaces

MicroBlaze can be configured with up to 16 AXI4-Stream interfaces, each consisting of one input and one output port. The channels are dedicated uni-directional point-to-point data streaming interfaces.

For detailed information on the AXI4-Stream interface, please refer to the *AMBA 4 AXI4-Stream Protocol Specification, Version 1.0* (ARM IHI 0051A) document.

The interfaces on MicroBlaze are 32 bits wide. A separate bit indicates whether the sent/received word is of control or data type. The get instruction in the MicroBlaze ISA is used to transfer information from a port to a general purpose register. The put instruction is used to transfer data in the opposite direction. Both instructions come in 4 flavors: blocking data, non-blocking data, blocking control, and non-blocking control. For a detailed description of the get and put instructions, please refer to Chapter 5, MicroBlaze Instruction Set Architecture.

Hardware Acceleration

Each link provides a low latency dedicated interface to the processor pipeline. Thus they are ideal for extending the processors execution unit with custom hardware accelerators. A simple example is illustrated in Figure 2-25. The code uses RFSLx to indicate the used link.

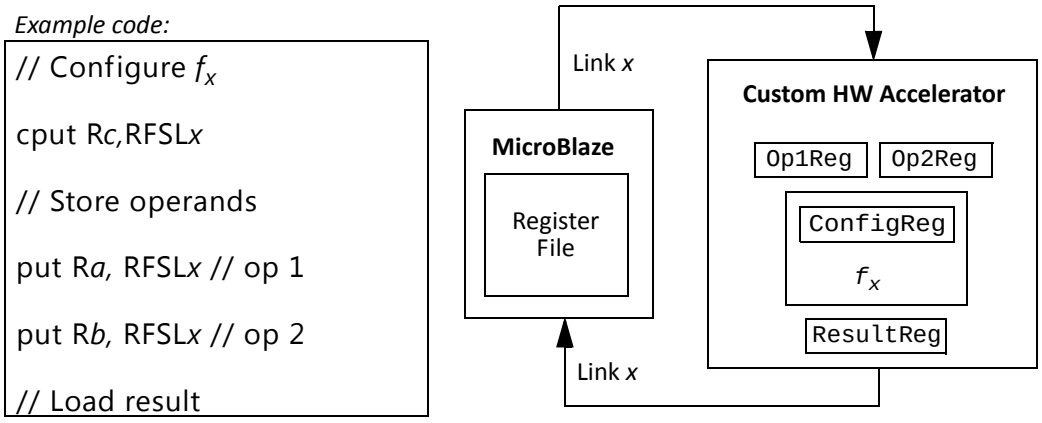

Figure 2-25: **Stream Link Used with HW Accelerated Function f_x**

This method is similar to extending the ISA with custom instructions, but has the benefit of not making the overall speed of the processor pipeline dependent on the custom function. Also, there are no additional requirements on the software tool chain associated with this type of functional extension.

Debug and Trace

Debug Overview

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

Debug registers are accessed via the JTAG debug interface, and are not directly visible to software running on the processor, unless the MDM is configured to enable software access to user-accessible debug registers.

See the *MicroBlaze Debug Module (MDM) Product Guide* (PG115) for a detailed description of the MDM features.

The basic debugging features enabled by setting `C_DEBUG_ENABLED` to 1 (Basic) include:

- Configurable number of hardware breakpoints and watchpoints and unlimited software breakpoints
- External processor control enables debug tools to stop, reset, and single step MicroBlaze
- Read from and write to: memory, general purpose registers, and special purpose register, except EAR, EDR, ESR, BTR and PVR0 - PVR12, which can only be read
- Support for multiple processors

The extended debugging features enabled by setting `C_DEBUG_ENABLED` to 2 (Extended) include:

- Configurable number of performance monitoring event and latency counters
- Program Trace:
  - Embedded program trace with configurable trace buffer size
  - External program trace for multiple processors, provided by the MDM
- Non-intrusive profiling support with configurable profiling buffer size
- Cross trigger support between multiple processors, and external cross trigger inputs and outputs, provided by the MDM

Performance Monitoring

With extended debugging, MicroBlaze provides performance monitoring counters to count various events and to measure latency during program execution. The number of event counters and latency counters can be configured with `C_DEBUG_EVENT_COUNTERS` and `C_DEBUG_LATENCY_COUNTERS` respectively, and the counter width can be set to 32, 48 or 64 bits with `C_DEBUG_COUNTER_WIDTH`. With the default configuration, the counter width is set to 32 bits and there are five event counters and one latency counter.

An event counter simply counts the number of times a certain event has occurred, whereas a latency counter provides the following information:

- Number of times the event has occurred (N)
- The sum of each event latency measured by counting clock cycles from the event starts until it finishes (ΣL), used to calculate the mean latency
- The sum of each event latency squared (ΣL^2), used to calculate the latency standard deviation
- The minimum, shortest, measured latency for all events (L_{min})
- The maximum, longest, measured latency for all events (L_{max})

The mean latency (μ) is calculated by the formula:

$$\mu = \frac{\Sigma L}{N}$$

The standard deviation (σ) of the latency is calculated by the formula:

$$\sigma = \frac{\sqrt{N\Sigma L^2 - (\Sigma L)^2}}{N}$$

Counting can be started or stopped via the Performance Counter Command Register or by cross trigger events (see Table 2-61).

When configuring, reading or writing counters, they are accessed sequentially through the performance counter registers. After every access the selected counter item is incremented.

All counters are sampled simultaneously for reading via the Performance Counter Command Register. This can be done while counting, or after counting has been stopped.

When an event counter reaches its maximum value, the overflow status bit is set, and the external interrupt signal `Dbg_Intr` is set to one. The interrupt signal is reset to zero by clearing the counters via the Performance Counter Command Register.

By using one of the event counters to count number of clock cycles, and initializing this counter to overflow after a predetermined sampling interval, the external interrupt can be used to periodically sample the performance counters.

The available events are described in Table 2-40, listed in numerical order.

A typical procedure to follow when initializing and using the performance monitoring counters is delineated in the steps below.

- Initialize the events to be monitored:
  - Use the Performance Command Register (Table 2-43) to reset the selected counter to the first counter, by setting the Reset bit.
  - Write the desired event numbers for all counters in order, using the Performance Control Register (Table 2-42). With the default configuration this means writing the register five times for the event counters and then once for the latency counter.
- Clear all counters and start monitoring using the Performance Command Register, by setting the Clear and Start bits.
- Run the program or function to be monitored.
- Sample counters and stop monitoring using the Performance Command Register, by setting the Sample and Stop bits.
- Read the results from all counters:
  - Use the Performance Command Register to reset the selected counter to the first counter, by setting the Reset bit.
  - Read the status for all counters in order, using the Performance Counter Status Register (Table 2-44). With the default configuration this means reading the register five times for the event counters and then once for the latency counter. Ensure that the result is valid by checking that the overflow and full bits are not set.
  - Use the Performance Command Register to reset the selected counter to the first counter, by setting the Reset bit.
  - Read the counter items for all counters in order, using the Performance Counter Data Read Register (Table 2-52). With the default configuration this means reading the register five times for the event counters and then four times for the latency counter as described in Table 2-53.
- Calculate the final results, depending on the measured events, for example:
  - Use the formulas above to determine the mean latency and standard deviation for any measured latency.
  - The clock cycles per instruction (CPI) can be calculated by E_{30} / E_0.
  - The instruction and data cache hit rates can be calculated by E_{11} / E_{10} and E_{47} / E_{46}.
  - The instruction cache miss latency is determined by ($E_{60}(\Sigma L)$ - $E_{60}(N)$) / (E_{10} - E_{11}), and equivalent formulas can be used to determine the data cache read and write miss latencies.
  - The ratio of floating point instructions in a program is E_{29}/E_0.

Table 2-40: **MicroBlaze Performance Monitoring Events**

Event	Description	Event	Description
Event Counter events			
0	Any valid instruction executed	29	Floating point (*fadd, ..., fsqrt*)
1	Load word (*lw, lwi, lwx*) executed	30	Number of clock cycles
2	Load halfword (*lhu, lhui*) executed	31	Immediate (*imm*) executed
3	Load byte (*lbu, lbui*) executed	32	Pattern compare (*pcmpbf, pcmpeq, pcmpne*)
4	Store word (*sw, swi, swx*) executed	33	Sign extend instructions (*sext8, sext16*) executed
5	Store halfword (*sh, shi*) executed	34	Instruction cache invalidate (*wic*) executed
6	Store byte (*sb, sbi*) executed	35	Data cache invalidate or flush (*wdc*) executed
7	Unconditional branch (*br, bri, brk, brki*) executed	36	Machine status instructions (*msrset, msrclr*)
8	Taken conditional branch (*beq, ..., bnei*) executed	37	Unconditional branch with delay slot executed
9	Not taken conditional branch (*beq, ..., bnei*) executed	38	Taken conditional branch with delay slot executed
10	Data request from instruction cache	39	Not taken conditional branch with delay slot
11	Hit in instruction cache	40	Delay slot with no operation instruction executed
12	Read data requested from data cache	41	Load instruction (*lbu, ..., lwx*) executed
13	Read data hit in data cache	42	Store instruction (*sb, ..., swx*) executed
14	Write data request to data cache	43	MMU data access request
15	Write data hit in data cache	44	Conditional branch (*beq, ..., bnei*) executed
16	Load (*lbu, ..., lwx*) with r1 as operand executed	45	Branch (*br, bri, brk, brki, beq, ..., bnei*) executed
17	Store (*sb, ..., swx*) with r1 as operand executed	46	Read or write data request from/to data cache
18	Logical operation (*and, andn, or, xor*) executed	47	Read or write data cache hit
19	Arithmetic operation (*add, idiv, mul, rsub*) executed	48	MMU exception taken
20	Multiply operation (*mul, mulh, mulhu, mulhsu, muli*)	49	MMU instruction side exception taken
21	Barrel shifter operation (*bsrl, bsra, bsll*) executed	50	MMU data side exception taken
22	Shift operation (*sra, src, srl*) executed	51	Pipeline stalled
23	Exception taken	52	Branch target cache hit for a branch or return
24	Interrupt occurred	53	MMU instruction side access request
25	Pipeline stalled due to operand fetch stage (OF)	54	MMU instruction TLB (ITLB) hit
26	Pipeline stalled due to execute stage (EX)	55	MMU data TLB (DTLB) hit
27	Pipeline stalled due to memory stage (MEM)	56	MMU unified TLB (UTLB) hit
28	Integer divide (*idiv, idivu*) executed		
Latency and Event Counter events			
57	Interrupt latency from input to interrupt vector	61	MMU address lookup latency
58	Data cache latency for memory read	62	Peripheral AXI interface data read latency
59	Data cache latency for memory write	63	Peripheral AXI interface data write latency
60	Instruction cache latency for memory read		

The debug registers used to configure and control performance monitoring, and to read or write the event and latency counters, are listed in Table 2-41. All of these registers except the Performance Counter Command register are accessed repeatedly to read or write information, first for all of the event counters followed by all of the latency counters.

The DBG_CTRL Value indicates the value to use in the MDM Debug Register Access Control Register to access the register, used with MDM software access to debug registers.

Table 2-41: **MicroBlaze Performance Monitoring Debug Registers**

Register Name	Size (bits)	MDM Command	DBG_CTRL Value	R/W	Description
Performance Counter Control	8	0101 0001	4A207	W	Select event for each configured counter, according to Table 2-40
Performance Counter Command	5	0101 0010	4A404	W	Command to clear counters, start or stop counting, or sample counters
Performance Counter Status	2	0101 0011	4A601	R	Read the sampled status for each configured performance counter
Performance Counter Data Read	32	0101 0110	4AC1F	R	Read the sampled values for each configured performance counter
Performance Counter Data Write	32	0101 0111	4AE1F	W	Write initial values for each configured performance counter

Performance Counter Control Register

The Performance Counter Control Register (PCCTRLR) is used to define the events that are counted by the configured performance counters. To define the events for all configured counters, the register should be written repeatedly for each of the counters. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

Every time the register is written, the selected counter is incremented. By using the Performance Counter Command Register, the selected counter can be reset to the first counter again.

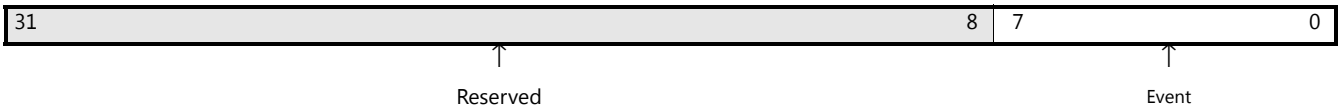

Figure 2-26: **Performance Counter Control Register**

Table 2-42: **Performance Counter Control Register (PCCTRLR)**

Bits	Name	Description	Reset Value
7:0	Event	Performance counter event, according to Table 2-40.	0

Performance Counter Command Register

The Performance Counter Command Register (PCCMDR) is used to issue commands to clear, start, stop, or sample all counters. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

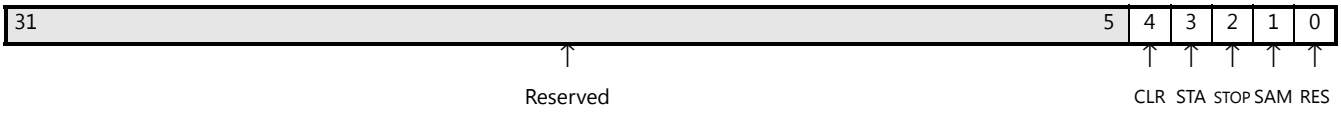

Figure 2-27: **Performance Counter Command Register**

Table 2-43: **Performance Counter Command Register (PCCMDR)**

Bits	Name	Description	Reset Value
4	Clear	Clear all counters to zero	0
3	Start	Start counting configured events for all counters simultaneously	0
2	Stop	Stop counting all counters simultaneously	0
1	Sample	Sample status and values in all counters simultaneously for reading	0
0	Reset	Reset accessed counter to the first event counter for access using the Performance Counter Command, Status, Read Data and Write Data	0

Performance Counter Status Register

The Performance Counter Status Register (PCSR) reads the sampled status of the counters. To read the status for all configured counters, the register should be read repeatedly for each of the counters. This register is a read-only register. Issuing a write request to the register does nothing.

Every time the register is read, the selected counter is incremented. By using the Performance Counter Command Register, the selected counter can be reset to the first counter again.

Figure 2-28: **Performance Counter Status Register**

Table 2-44: **Performance Counter Status Register (PCSR)**

Bits	Name	Description	Reset Value
1	Overflow	This bit is set when the counter has counted past its maximum value	0
0	Full	This bit is set when a new latency counter event is started before the previous event has finished. This indicates that the accuracy of the measured values is reduced.	0

Performance Counter Data Read Register

The Performance Counter Data Read Register (PCDRR) reads the sampled values of the counters. To read the values of all configured counters, the register should be read repeatedly. This register is a read-only register. Issuing a write request to the register does nothing.

Since a counter can have more than 32 bits, depending on the configuration, the register may need to be read repeatedly to retrieve all information for a particular counter. This is detailed in Table 2-46.

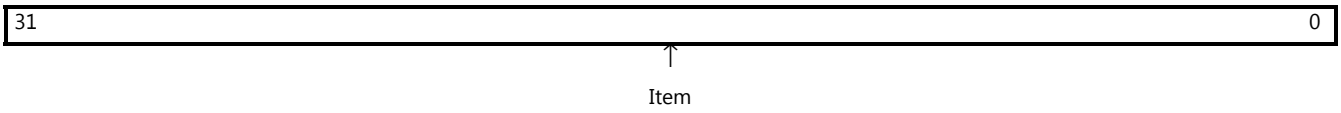

Figure 2-29: **Performance Counter Data Read Register**

Table 2-45: **Performance Counter Data Read Register (PCDRR)**

Bits	Name	Description	Reset Value
31:0	Item	Sampled counter value item	0

Table 2-46: **Performance Counter Data Items**

Counter Type	Item	Description	
C_DEBUG_COUNTER_WIDTH = 32			
Event Counter	1	The number of times the event occurred	
Latency Counter	1	The number of times the event occurred	
	2	The sum of each event latency	
	3	The sum of each event latency squared	
	4	31:16 15:0	Minimum measured latency, 16 bits Maximum measured latency, 16 bits
C_DEBUG_COUNTER_WIDTH = 48			
Event Counter	1	31:16 15:0	0x0000 The number of times the event occurred, 16 most significant bits
	2	The number of times the event occurred, 32 least significant bits	

Table 2-46: **Performance Counter Data Items *(Cont'd)***

Counter Type	Item	Description	
Latency Counter	1	The number of times the event occurred	
	2	31:16 15:0	0x0000 The sum of each event latency, 16 most significant bits
	3	The sum of each event latency, 32 least significant bits	
	4	31:16 15:0	0x0000 The sum of each event latency squared, 16 most significant bits
	5	The sum of each event latency squared, 32 least significant bits	
	6	Minimum measured latency, 32 bits	
	7	Maximum measured latency, 32 bits	
C_DEBUG_COUNTER_WIDTH = 64			
Event Counter	1	The number of times the event occurred, 32 most significant bits	
	2	The number of times the event occurred, 32 least significant bits	
Latency Counter	1	The number of times the event occurred, 32 bits	
	2	The sum of each event latency, 32 most significant bits	
	3	The sum of each event latency, 32 least significant bits	
	4	The sum of each event latency squared, 32 most significant bits	
	5	The sum of each event latency squared, 32 least significant bits	
	6	Minimum measured latency, 32 bits	
	7	Maximum measured latency, 32 bits	

Performance Counter Data Write Register

The Performance Counter Data Read Register (PCDWR) writes initial values to the counters. To write all configured counters, the register should be written repeatedly. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

Since a counter can have more than 32 bits, depending on the configuration, the register may need to be written repeatedly to update all information for a particular counter, as described in Table 2-46.

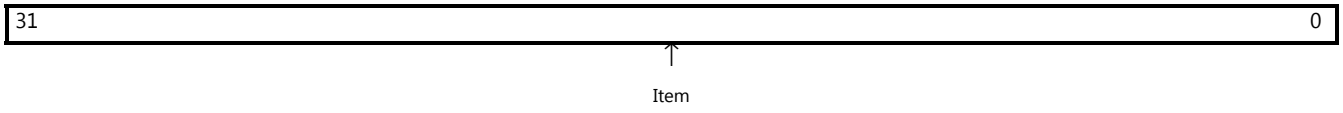

Figure 2-30: **Performance Counter Data Write Register**

Table 2-47: **Performance Counter Data Write Register (PCDWR)**

Bits	Name	Description	Reset Value
31:0	Item	Counter value item to write into a counter	0

Program Trace

With extended debugging, MicroBlaze provides program trace, either storing information in the Embedded Trace Buffer or transmitting it to the MDM, to enable program execution tracing. The MDM is used when the parameter C_DEBUG_EXTERNAL_TRACE is set, allowing output of program trace from multiple processors via external interfaces.

The size of the Embedded Trace Buffer can be configured from 8KB to 128KB using the parameter C_DEBUG_TRACE_SIZE. By setting C_DEBUG_TRACE_SIZE to 0 (None), program trace is disabled.

Program trace uses compression to reduce the amount of trace data, while still allowing reconstruction of the program execution flow or the entire processor software state. There are three main compression levels:

- Complete trace

  Stores complete trace information including cycle count for each executed instruction using 144 bits, ranging from 512 to 8192 items depending on the configured Embedded Trace Buffer size. Complete trace is not available when C_DEBUG_EXTERNAL_TRACE is set.

- Program flow

  Stores program flow changes, i.e. the sequence of branches taken or not taken, and the new program counter for indirect branches, interrupts, exceptions and hardware breaks.

  The program counter may also optionally be stored for return instructions to simplify program flow reconstruction, or for all taken branches to handle self-modifying code.

  Data read from memory or fetched from AXI4-Stream interfaces may optionally be stored to allow reconstructing the entire processor software state, enabling reverse single step functionality.

- Program flow and cycle count

  Stores the cycle count between instructions along with the same information as program flow alone, to also allow reconstruction of the program execution time.

Tracing can be started via the Trace Command Register, by hitting a program breakpoint or watchpoint configured as a tracepoint in the Trace Control Register, or by a cross trigger event (see Table 2-61).

Tracing is automatically stopped when the trace buffer becomes full, and can be stopped via the Trace Command Register or by a cross trigger event (see Table 2-61).

The cycle count can measure up to 32768 clock cycles when using complete trace, and up to 8192 cycles between instructions when using program flow and cycle count. If the cycle count exceeds this value, the Trace Status Register overflow bit is set to one.

It is possible to configure trace to halt the processor when the trace buffer becomes full or when the cycle count overflows. This allows continuous trace of the entire program flow, albeit not in real time due to the time required to read the trace buffer.

The debug registers used to configure and control tracing, and to read the Embedded Trace Buffer, are listed in Table 2-48.

The DBG_CTRL Value indicates the value to use in the MDM Debug Register Access Control Register to access the register, used with MDM software access to debug registers.

Table 2-48: **MicroBlaze Program Trace Debug Registers**

Register Name	Size (bits)	MDM Command	DBG_CTRL Value	R/W	Description
Trace Control	22	0110 0001	4C215	W	Set tracepoints, trace compression level and optionally stored trace information
Trace Command	4	0110 0010	4C403	W	Command to clear trace buffer, start or stop trace, and sample number of current buffer items
Trace Status	18	0110 0011	4C611	R	Read the sampled trace buffer status
Trace Data Read[1]	18	0110 0110	4CC11	R	Read the oldest item from the Embedded Trace Buffer

1. This register is not available when C_DEBUG_EXTERNAL_TRACE is set

Trace Control Register

The Trace Control Register (TCTRLR) is used to define the trace behavior. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

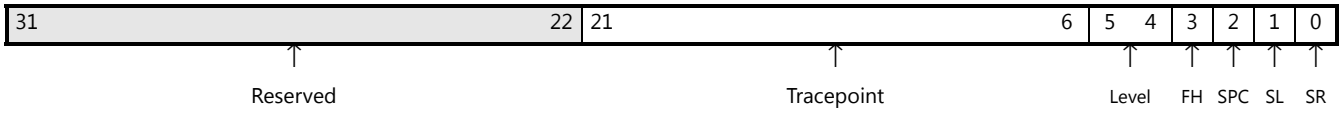

Figure 2-31: **Trace Control Register**

Table 2-49: **Trace Control Register (TCTRLR)**

Bits	Name	Description	Reset Value
21:6	Tracepoint	Change corresponding breakpoint or watchpoint to a tracepoint	0
5:4	Level	Trace compression level: 00 = Complete trace, not available with C_DEBUG_EXTERNAL_TRACE 01 = Program flow 10 = Program flow and cycle count 11 = *Reserved*	00
3	Full Halt	Debug Halt on full trace buffer or cycle count overflow	0

Table 2-49: **Trace Control Register (TCTRLR) *(Cont'd)***

Bits	Name	Description	Reset Value
2	Save PC	Save new program counter for all taken branches	0
1	Save Load	Save load and get instruction new data value	0
0	Save Return	Save new program counter for return instructions	0

Trace Command Register

The Trace Command Register (TCMDR) is used to issue commands to clear, start, or stop trace, as well as sample the number of trace items. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

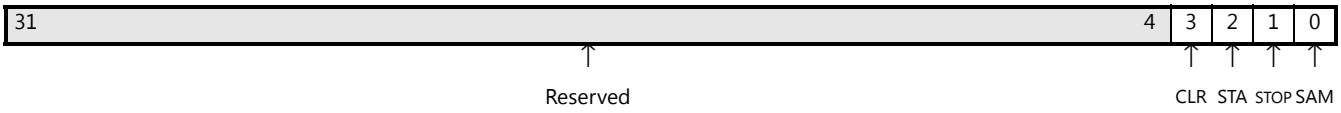

Figure 2-32: **Trace Command Register**

Table 2-50: **Trace Command Register (TCMDR)**

Bits	Name	Description	Reset Value
3	Clear	Clear trace status and empty the trace buffer	0
2	Start	Start trace immediately	0
1	Stop	Stop trace immediately	0
0	Sample	Sample the number of current items in the trace buffer	0

Trace Status Register

The Trace Status Register (TSR) can be used to determine if trace has been started or not, to check for cycle count overflow and to read the sampled number of items in the Embedded Trace Buffer. This register is a read-only register. Issuing a write request to the register does nothing.

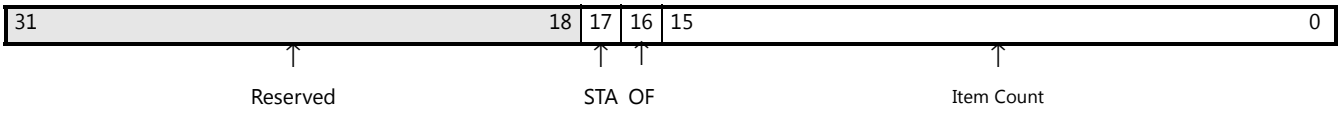

Figure 2-33: **Trace Status Register**

Table 2-51: **Trace Status Register (TSR)**

Bits	Name	Description	Reset Value
17	Started	Trace started, set to one when trace is started and cleared to zero when it is stopped	0
16	Overflow	Cycle count overflow, set to one when the cycle count overflows, and cleared to zero by the Clear command	0
15:0	Item Count	Sampled trace buffer item count	0x0000

Trace Data Read Register

The Trace Data Read Register (TDRR) contains the oldest item read from the Embedded Trace Buffer. When the register has been read, the next item is read from the trace buffer. It is an error to read more items than are available in the trace buffer, as indicated by the item count in the Trace Status Register. This register is a read-only register. Issuing a write request to the register does nothing.

Since a trace data entity can consist of more than 18 bits, depending on the compression level and stored data, the register may need to be read repeatedly to retrieve all information for a particular data entity. This is detailed in Table 2-53.

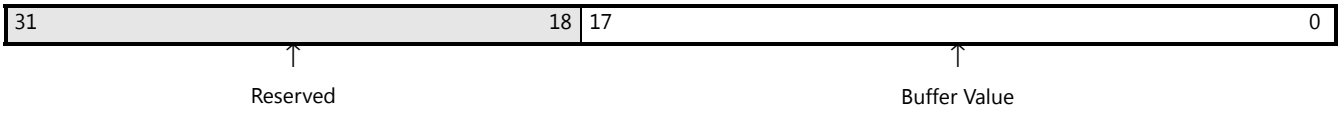

Figure 2-34: **Trace Data Read Register**

Table 2-52: **Trace Data Read Register (TDRR)**

Bits	Name	Description	Reset Value
17:0	Buffer Value	Embedded Trace Buffer item	0x00000

Table 2-53: **Trace Counter Data Entities**

Entity	Item	Bits	Description
Complete Trace	1	17:3 2:0	Cycle count for the executed instruction Machine Status Register [17:19]
	2	17:6 5:1 0	Machine Status Register [20:31] Destination register address (r0 - r31), valid if written Destination register written if set to one

Table 2-53: **Trace Counter Data Entities *(Cont'd)***

Entity	Item	Bits	Description
	3	17:13 12 11 10 9:6 5:0	Exception Kind, valid if exception taken Exception taken if set to one Load instruction reading data if set to one Store instruction writing data if set to one Byte enable, valid for store instruction Write data [0:5] for store instructions, or Destination register data [0:5] for other instructions
	4	17:0	Write data [6:23] or Destination register data [6:23]
	5	17:10 9:0	Write data [24:31] or Destination register data [24:31] Data address [0:9] for load and store instructions, or Executed instruction [0:9] for other instruction
	6	17:0	Data address [10:27] or Executed instruction [10:27]
	7	17:14 13:0	Data address [28:31] or Executed instruction [28:31] Program Counter [0:13]
	8	17:0	Program Counter [14:31]
Program Flow: Branches	1	17:16 15:12 11:0	00 - The item contains program flow branches Number of branches (N) counted in the item (1 - 12) The N leftmost bits represent branches in the program flow. If the bit is set to one the branch is taken, otherwise it is not taken.
Program Flow: Program Counter	1	17:16 15:0	01 - The item contains a Program Counter value Program Counter [0:15]
	2	17:16 15:0	01 - The item contains a Program Counter value Program Counter [16:31]
Program Flow: Read Data	1	17:16 15:0	10 - The item contains read data Data read by load and get instructions [0:15]
	2	17:16 15:0	10 - The item contains read data Data read by load and get instructions [15:31]
Program Flow with Cycle Count: Branches and short cycle count	1	17:16 15:14 13:8 7 6:1 0	00 - The item contains program flow branches 01, 10 - Number of branches (N) counted (1 - 2) Cycle count for previously executed instructions Branch is taken if set to one, otherwise it is not taken Cycle count for previously executed instructions Branch is taken if set to one, otherwise it is not taken
Program Flow with Cycle Count: Branch and long cycle count	1	17:16 15:14 13:1 0	00 - The item contains program flow branches 11 - The item contains branch and long cycle count Cycle count for previously executed instructions Branch is taken if set to one, otherwise it is not taken

Non-Intrusive Profiling

With extended debugging, non-intrusive profiling is provided, which uses a Profiling Buffer to store program execution statistics. The size of the profiling buffer can be configured from 4KB to 128KB using the parameter `C_DEBUG_PROFILE_SIZE`. By setting `C_DEBUG_PROFILE_SIZE` to 0 (None), non-intrusive profiling is disabled.

The Profiling Buffer is divided into a number of bins, each counting the number of executed instructions or clock cycles within a certain address range. Each bin counts up to 2^{36} - 1 = 68719476735 instructions or cycles.

The address range of each bin is determined by the buffer size and the profiled address range defined via the Profiling Low Address Register and Profiling High Address Register.

Profiling can be started or stopped via the Profiling Control Register or by cross trigger events (see Table 2-61).

The debug registers used to configure and control profiling, and to read or write the Profiling Buffer, are listed in Table 2-54.

The DBG_CTRL Value indicates the value to use in the MDM Debug Register Access Control Register to access the register, used with MDM software access to debug registers.

Table 2-54: **MicroBlaze Profiling Debug Registers**

Register Name	Size (bits)	MDM Command	DBG_CTRL Value	R/W	Description
Profiling Control	8	0111 0001	4E207	W	Enable or disable profiling, configure counting method and bin usage
Profiling Low Address	30	0111 0010	4E41D	W	Defines the low address of the profiled address range
Profiling High Address	30	0111 0011	4E61D	W	Defines the high address of the profiled address range
Profiling Buffer Address	9 - 14	0111 0100	9: 4E808 10: 4E809 ... 14: 4E80D	W	Sets the address (bin) in the Profiling Buffer to read or write
Profiling Data Read	36	0111 0110	4EC23	R	Read data from the Profiling Buffer
Profiling Data Write	36	0111 0111	4EE23	W	Write data to the Profiling Buffer

Profiling Control Register

The Profiling Control Register (PCTRLR) is used to enable (start) profiling and disable (stop) profiling. It is also used to configure whether to count the number of executed instructions or the number of executed clock cycles, as well as define the Profiling Buffer bin usage. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

The Bin Control value (B) can be calculated by the formula

$$B = \left\lceil log_2 \frac{H - L + S \cdot 4}{S \cdot 4} \right\rceil$$

where *L* is the Profiling Low Register, *H* is the Profiling High Register, and *S* is the parameter C_DEBUG_PROFILE_SIZE.

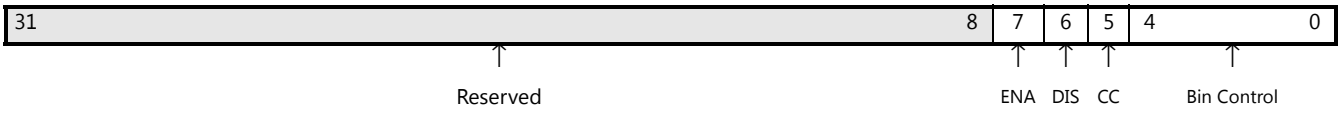

Figure 2-35: **Profiling Control Register**

Table 2-55: **Profiling Control Register (PCTRLR)**

Bits	Name	Description	Reset Value
7	Enable	Enable and start profiling	0
6	Disable	Disable and stop profiling	0
5	Enable Cycle Count	Enable cycle count to count clock cycles of executed instruction 0 = Disabled, number of executed instructions counted 1 = Enabled, clock cycles of executed instructions counted	0
4:0	Bin Control	The number of addresses counted by each bin in the Profiling Buffer	00000

Profiling Low Address Register

The Profiling Low Address Register (PLAR) is used to define the low word address of the profiled area. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

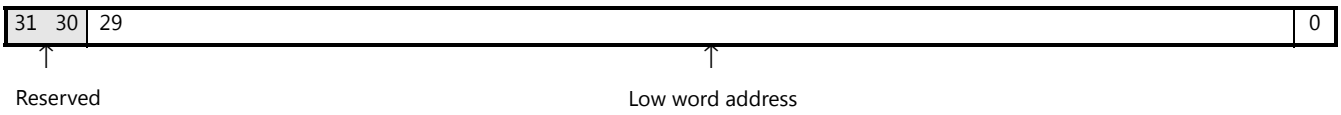

Figure 2-36: **Profiling Low Address Register**

Table 2-56: **Profiling Low Address Register (PLAR)**

Bits	Name	Description	Reset Value
29:0	Low word	Low word address of the profiled area	0

Profiling High Address Register

The Profiling High Address Register (PLAR) is used to define the high word address of the profiled area. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

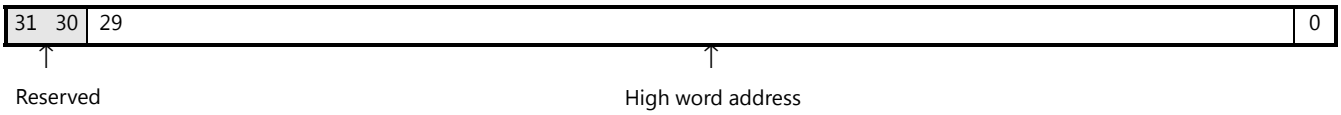

Figure 2-37: **Profiling High Address Register**

Table 2-57: **Profiling High Address Register (PHAR)**

Bits	Name	Description	Reset Value
29:0	High word	High word address of the profiled area	0

Profiling Buffer Address Register

The Profiling Buffer Address Register (PBAR) is used to define the bin in the Profiling Buffer to be read or written. This register has variable number of bits, depending on the parameter C_DEBUG_PROFILE_SIZE. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

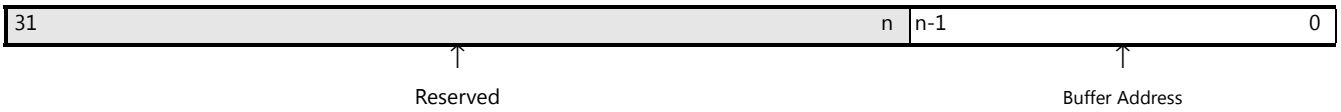

Figure 2-38: **Profiling Buffer Address Register**

Table 2-58: **Profiling Buffer Address Register (PBAR)**

Bits	Name	Description	Reset Value
n-1:0	Buffer Address	Bin in the Profiling Buffer to read or write. The number of bits (n) is 9 for a 4KB buffer, 10 for a 8KB buffer, ..., 14 for a 128KB buffer.	0

Profiling Data Read Register

The Profiling Data Read Register (PDRR) reads the bin value indicated by the Profiling Buffer Address Register and increments the Profiling Buffer Address Register. This register is a read-only register. Issuing a write request to the register does nothing.

When reading this register with MDM software access to debug registers, data is read with two consecutive accesses.

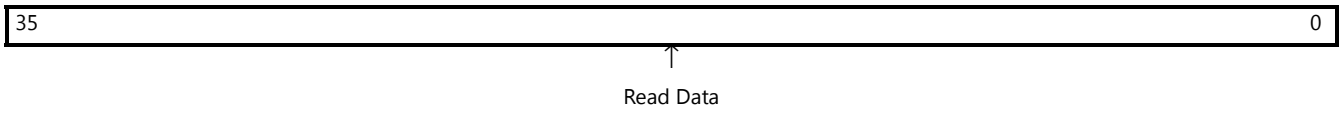

Figure 2-39: **Profiling Data Read Register**

Table 2-59: **Profiling Data Read Register (PDRR)**

Bits	Name	Description	Reset Value
35:0	Read Data	Number of executed instructions or executed clock cycles in the bin	0

Profiling Data Write Register

The Profiling Data Write Register (PDWR) writes a new value to the bin indicated by the Profiling Buffer Address Register and increments the Profiling Buffer Address Register. This register is a write-only register. Issuing a read request has no effect, and undefined data is read.

This register can be used to clear the Profiling Buffer before enabling profiling.

When writing this register with MDM software access to debug registers, data is written with two consecutive accesses.

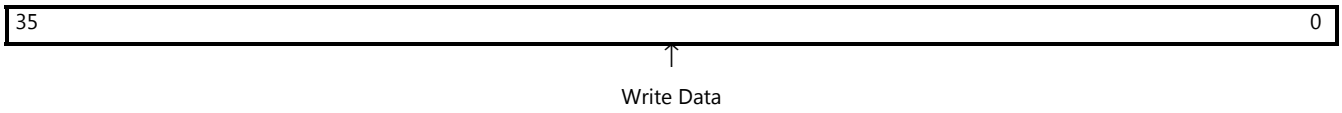

Figure 2-40: **Profiling Data Write Register**

Table 2-60: **Profiling Data Write Register (PDWR)**

Bits	Name	Description	Reset Value
35:0	Write Data	Data to write to a bin	0

Cross Trigger Support

With basic debugging, cross trigger support is provided by two signals, DBG_STOP and MB_Halted.

- When the DBG_STOP input is set to 1, MicroBlaze will halt after a few instructions. XMD will detect that MicroBlaze has halted, and indicate where the halt occurred. The signal can be used to halt MicroBlaze at any external event, for example when a ChipScope™ integrated logic analyzer is triggered.
- Whenever Microblaze is halted the MB_Halted output signal is set to 1, for example after a breakpoint or watchpoint is hit, after a stop XMD command, or when the DBG_STOP input is set. The output is cleared when MicroBlaze execution is resumed by an XMD command.

  The MB_Halted signal may be used to trigger a ChipScope integrated logic analyzer, or halt other MicroBlaze cores in a multiprocessor system by connecting the signal to their DBG_STOP inputs.

With extended debugging, cross trigger support is available in conjunction with the MDM. The MDM provides programmable cross triggering between all connected processors, as well as external trigger inputs and outputs. For details, see the *MicroBlaze Debug Module (MDM) Product Guide* (PG115).

MicroBlaze can handle up to eight cross trigger actions. Cross trigger actions are generated by the corresponding MDM cross trigger outputs, connected via the Debug bus. The effect of each of the cross trigger actions is listed in Table 2-61.

MicroBlaze can generate up to eight cross trigger events. Cross trigger events affect the corresponding MDM cross trigger inputs, connected via the Debug bus. The cross trigger events are described in Table 2-62.

Table 2-61: **MicroBlaze Cross Trigger Actions**

Number	Action	Description
0	Debug stop	Stop MicroBlaze if the processor is executing, and set the MB_Halted output. The same effect is achieved by setting the Dbg_Stop input.
1	Continue execution	Continue execution if the processor is stopped, and clear the MB_Halted output.
2	Stop program trace	Stop program trace if tracing is in progress.
3	Start program trace	Start program trace if trace is stopped.
4	Stop performance monitoring	Stop performance monitoring if it is in progress.
5	Start performance monitoring	Start performance monitoring if it is stopped.
6	Disable profiling	Disable profiling if it is in progress.
7	Enable profiling	Enable profiling if it is disabled.

Table 2-62: **MicroBlaze Cross Trigger Events**

Number	Event	Description
0	MicroBlaze halted	Generate an event when MicroBlaze is halted. The same event is signalled when the `MB_Halted` output is set.
1	Execution resumed	Generate an event when the processor resumes execution from debug halt. The same event is signalled when the `MB_Halted` output is cleared.
2	Program trace stopped	Generate an event when program trace is stopped by writing a command to the Program Trace Command Register, when the trace buffer is full, or by a cross trigger action.
3	Program trace started	Generate an event when program trace is started by writing a command to the Program Trace Command Register, by hitting a tracepoint, or by a cross trigger action.
4	Performance monitoring stopped	Generate an event when performance monitoring is stopped by writing a command to the Performance Counter Command Register or by a cross trigger action.
5	Performance monitoring started	Generate an event when performance monitoring is started by writing a command to the Performance Counter Command Register, or by a cross trigger action.
6	Profiling disabled	Generate an event when profiling is enabled by writing a command to the Profiling Control Register or by a cross trigger action.
7	Profiling enabled	Generate an event when profiling is disabled by writing a command to the Profiling Control Register or by a cross trigger action.

Trace Interface Overview

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

See Table 3-14 for a list of exported signals.

Fault Tolerance

The fault tolerance features included in MicroBlaze, enabled with C_FAULT_TOLERANT, provide Error Detection for internal block RAMs, and support for Error Detection and Correction (ECC) in LMB block RAMs. When fault tolerance is enabled, all soft errors in block RAMs are detected and corrected, which significantly reduces overall failure intensity.

In addition to protecting block RAM, the FPGA configuration memory also generally needs to be protected. A detailed explanation of this topic, and further references, can be found in the document *LogiCore IP Soft Error Mitigation Controller* (PG036).

Configuration

Using MicroBlaze Configuration

Fault tolerance can be enabled in the MicroBlaze configuration dialog, on the General page.

After enabling fault tolerance in MicroBlaze, ECC is automatically enabled in the connected LMB BRAM Interface Controllers by the tools, when the system is generated. This means that nothing else needs to be configured to enable fault tolerance and minimal ECC support.

It is possible (albeit not recommended) to manually override ECC support, leaving the LMB BRAM unprotected, by disabling C_ECC in the configuration dialogs of all connected LMB BRAM Interface Controllers. In this case, the internal MicroBlaze block RAM protection is still enabled, since fault tolerance is enabled.

Using LMB BRAM Interface Controller Configuration

As an alternative to the method described above, it is also possible to enable ECC in the configuration dialogs of all connected LMB BRAM Interface Controllers. In this case, fault tolerance is automatically enabled in MicroBlaze by the tools, when the system is generated. This means that nothing else needs to be configured to enable ECC support and MicroBlaze fault tolerance.

ECC must either be enabled or disabled in all Controllers, which is enforced by a DRC.

It is possible to manually override fault tolerance support in MicroBlaze, by explicitly disabling C_FAULT_TOLERANT in the MicroBlaze configuration dialog. This is not recommended, unless no block RAM is used in MicroBlaze, and there is no need to handle bus exceptions from uncorrectable ECC errors.

Features

An overview of all MicroBlaze fault tolerance features is given here. Further details on each feature can be found in the following sections:

- "Instruction Cache Operation"
- "Data Cache Operation"
- "UTLB Management"
- "Branch Target Cache"
- "Instruction Bus Exception"
- "Data Bus Exception"
- "Exception Causes"

The LMB BRAM Interface Controller v4.0 or later provides the LMB ECC implementation. For details, including performance and resource utilization, see the *LogiCORE IP LMB BRAM Interface Controller* (PG112) product guide, in the Xilinx IP Documentation.

Instruction and Data Cache Protection

To protect the block RAM in the Instruction and Data Cache, parity is used. When a parity error is detected, the corresponding cache line is invalidated. This forces the cache to reload the correct value from external memory. Parity is checked whenever a cache hit occurs.

Note that this scheme only works for write-through, and thus write-back data cache is not available when fault tolerance is enabled. This is enforced by a DRC.

When new values are written to a block RAM in the cache, parity is also calculated and written. One parity bit is used for the tag, one parity bit for the instruction cache data, and one parity bit for each word in a data cache line.

In many cases, enabling fault tolerance does not increase the required number of cache block RAMs, since spare bits can be used for the parity. Any increase in resource utilization, in particular number of block RAMs, can easily be seen in the MicroBlaze configuration dialog, when enabling fault tolerance.

Memory Management Unit Protection

To protect the block RAM in the MMU Unified Translation Look-Aside Buffer (UTLB), parity is used. When a parity error is detected during an address translation, a TLB miss exception occurs, forcing software to reload the entry.

When a new TLB entry is written using the TLBHI and TLBLO registers, parity is calculated. One parity bit is used for each entry.

Parity is also checked when a UTLB entry is read using the TLBHI and TLBLO registers. When a parity error is detected in this case, the entry is marked invalid by clearing the valid bit.

Enabling fault tolerance does not increase the MMU block RAM size, since a spare bit is available for the parity.

Branch Target Cache Protection

To protect block RAM in the Branch Target Cache, parity is used. When a parity error is detected when looking up a branch target address, the address is ignored, forcing a normal branch.

When a new branch address is written to the Branch Target Cache, parity is calculated. One parity bit is used for each address.

Enabling fault tolerance does not increase the Branch Target Cache block RAM size, since a spare bit is available for the parity.

Exception Handling

With fault tolerance enabled, if an error occurs in LMB block RAM, the LMB BRAM Interface Controller generates error signals on the LMB interface.

If exceptions are enabled in MicroBlaze, by setting the EE bit in the Machine Status Register, the uncorrectable error signal either generates an instruction bus exception or a data bus exception, depending on the affected interface.

Should a bus exception occur when an exception is in progress, MicroBlaze is halted, and the external error signal `MB_Error` is set. This behavior ensures that it is impossible to execute an instruction corrupted by an uncorrectable error.

Software Support

Scrubbing

To ensure that bit errors are not accumulated in block RAMs, they must be periodically scrubbed.

The standalone BSP provides the function *`microblaze_scrub()`* to perform scrubbing of the entire LMB block RAM and all MicroBlaze internal block RAMs used in a particular configuration. This function is intended to be called periodically from a timer interrupt routine.

The following example code illustrates how this can be done.

```
#include "xparameters.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "mb_interface.h"

#define SCRUB_PERIOD ...

XIntc InterruptController; /* The Interrupt Controller instance */
XTmrCtr TimerCounterInst;/* The Timer Counter instance */

void MicroBlazeScrubHandler(void *CallBackRef, u8 TmrCtrNumber)
{
  /* Perform other timer interrupt processing here */
  microblaze_scrub();
}

int main (void)
{
  int Status;

  /*
   * Initialize the timer counter so that it's ready to use,
   * specify the device ID that is generated in xparameters.h
   */
  Status = XTmrCtr_Initialize(&TimerCounterInst, TMRCTR_DEVICE_ID);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  /*
   * Connect the timer counter to the interrupt subsystem such that
   * interrupts can occur.
   */
  Status = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  /*
   * Connect a device driver handler that will be called when an
   * interrupt for the device occurs, the device driver handler performs
   * the specific interrupt processing for the device
   */
  Status = XIntc_Connect(&InterruptController, TMRCTR_DEVICE_ID,
       (XInterruptHandler)XTmrCtr_InterruptHandler,
       (void *) &TimerCounterInst);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }
```

```
    /*
     * Start the interrupt controller such that interrupts are enabled for
     * all devices that cause interrupts, specifying real mode so that the
     * timer counter can cause interrupts thru the interrupt controller.
     */
    Status = XIntc_Start(&InterruptController, XIN_REAL_MODE);
    if (Status != XST_SUCCESS) {
      return XST_FAILURE;
    }

    /*
     * Setup the handler for the timer counter that will be called from the
     * interrupt context when the timer expires, specify a pointer to the
     * timer counter driver instance as the callback reference so the
     * handler is able to access the instance data
     */
    XTmrCtr_SetHandler(&TimerCounterInst, MicroBlazeScrubHandler,
              &TimerCounterInst);

    /*
     * Enable the interrupt of the timer counter so interrupts will occur
     * and use auto reload mode such that the timer counter will reload
     * itself automatically and continue repeatedly, without this option
     * it would expire once only
     */
    XTmrCtr_SetOptions(&TimerCounterInst, TIMER_CNTR_0,
         XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

    /*
     * Set a reset value for the timer counter such that it will expire
     * earlier than letting it roll over from 0, the reset value is loaded
     * into the timer counter when it is started
     */
    XTmrCtr_SetResetValue(TmrCtrInstancePtr,TmrCtrNumber,SCRUB_PERIOD);

    /*
     * Start the timer counter such that it's incrementing by default,
     * then wait for it to timeout a number of times
     */
    XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_0);

    ...
}
```

See the section "Scrubbing" below for further details on how scrubbing is implemented, including how to calculate the scrubbing rate.

BRAM Driver

The standalone BSP BRAM driver is used to access the ECC registers in the LMB BRAM Interface Controller, and also provides a comprehensive self test.

By implementing the SDK Xilinx C Project "Peripheral Tests", a self-test example including the BRAM self test for each LMB BRAM Interface Controller in the system is generated. Depending on the ECC features enabled in the LMB BRAM Interface Controller, this code will perform all possible tests of the ECC function.

The self-test example can be found in the standalone BSP BRAM driver source code, typically in the subdirectory `microblaze_0/libsrc/bram_v3_03_a/src/xbram_selftest.c`.

Scrubbing

Scrubbing Methods

Scrubbing is performed using specific methods for the different block RAMs:

- Instruction and data caches: All lines in the caches are cyclically invalidated using the WIC and WDC instructions respectively. This forces the cache to reload the cache line from external memory.
- Memory Management Unit UTLB: All entries in the UTLB are cyclically invalidated by writing the TLBHI register with the valid bit cleared.
- Branch Target Cache: The entire BTC is invalided by doing a synchronizing branch, BRI 4.
- LMB block RAM: All addresses in the memory are cyclically read and written, thus correcting any single bit errors on each address.

It is also possible to add interrupts for correctable errors from the LMB BRAM Interface Controllers, and immediately scrub this address in the interrupt handler, although in most cases it only improves reliability slightly.

The failing address can be determined by reading the Correctable Error First Failing Address Register in each of the LMB BRAM Interface Controllers. To be able to generate an interrupt `C_ECC_STATUS_REGISTERS` must be set to 1 in the connected LMB BRAM Interface Controllers, and to read the failing address `C_CE_FAILING_REGISTERS` must be set to 1.

Calculating Scrubbing Rate

The scrubbing rate depends on failure intensity and desired reliability.

The approximate equation to determine the LMB memory scrubbing rate is in our case given by

$$P_W \approx 760\left(\frac{BER^2}{SR^2}\right)$$

where P_W is the probability of an uncorrectable error in a memory word, *BER* is the soft error rate for a single memory bit, and *SR* is the Scrubbing Rate.

The soft error rates affecting block RAM for each product family can be found in the *Device Reliability Report* (UG116).

Use Cases

Several common use cases are described here. These use cases are derived from the *LogiCore IP Processor LMB BRAM Interface Controller* (PG112) product guide.

Minimal

This system is obtained when enabling fault tolerance in MicroBlaze, without doing any other configuration.

The system is suitable when area constraints are high, and there is no need for testing of the ECC function, or analysis of error frequency and location. No ECC registers are implemented. Single bit errors are corrected by the ECC logic before being passed to MicroBlaze. Uncorrectable errors set an error signal, which generates an exception in MicroBlaze.

Small

This system should be used when it is necessary to monitor error frequency, but there is no need for testing of the ECC function. It is a minimal system with Correctable Error Counter Register added to monitor single bit error rates. If the error rate is too high, the scrubbing rate should be increased to minimize the risk of a single bit error becoming an uncorrectable double bit error. Parameters set are `C_ECC` = 1 and `C_CE_COUNTER_WIDTH` = 10.

Typical

This system represents a typical use case, where it is required to monitor error frequency, as well as generating an interrupt to immediately correct a single bit error through software. It does not provide support for testing of the ECC function. It is a small system with Correctable Error First Failing registers and Status register added. A single bit error will latch the address for the access into the Correctable Error First Failing Address Register and set the CE_STATUS bit in the ECC Status Register. An interrupt will be generated triggering MicroBlaze to read the failing address and then perform a read followed by a write on the failing address. This will remove the single bit error from the BRAM, thus reducing the risk of the single bit error becoming a uncorrectable double bit error. Parameters set are `C_ECC` = 1, `C_CE_COUNTER_WIDTH` = 10, `C_ECC_STATUS_REGISTER` = 1 and `C_CE_FAILING_REGISTERS` = 1.

Full

This system uses all of the features provided by the LMB BRAM Interface Controller, to enable full error injection capability, as well as error monitoring and interrupt generation. It is a typical system with Uncorrectable Error First Failing registers and Fault Injection registers added. All features are switched on for full control of ECC functionality for system debug or systems with high fault tolerance requirements. Parameters set are `C_ECC` = 1, `C_CE_COUNTER_WIDTH` = 10, `C_ECC_STATUS_REGISTER` = 1 and `C_CE_FAILING_REGISTERS` = 1, `C_UE_FAILING_REGISTERS` = 1 and `C_FAULT_INJECT` = 1.

Lockstep Operation

MicroBlaze is able to operate in a lockstep configuration, where two or more identical MicroBlaze cores execute the same program. By comparing the outputs of the cores, any tampering attempts, transient faults or permanent hardware faults can be detected.

System Configuration

The parameter `C_LOCKSTEP_SLAVE` is set to one on all slave MicroBlaze cores in the system, except the master (or primary) core. The master core drives all the output signals, and handles the debug functionality. The port `Lockstep_Master_Out` on the master is connected to the port `Lockstep_Slave_In` on the slaves, in order to handle debugging.

The slave cores should not drive any output signals, only receive input signals. This must be ensured by only connecting signals to the input ports of the slaves. For buses this means that each individual input port must be explicitly connected.

The port `Lockstep_Out` on the master and slave cores provide all output signals for comparison. Unless an error occurs, individual signals from each of the cores are identical every clock cycle.

To ensure that lockstep operation works properly, all input signals to the cores must be synchronous. Input signals that may require external synchronization are `Interrupt`, `Reset`, `Ext_Brk`, and `Ext_Nm_Brk`.

Use Cases

Two common use cases are described here. In addition, lockstep operation provides the basis for implementing triple modular redundancy on MicroBlaze core level.

Tamper Protection

This application represents a high assurance use case, where it is required that the system is tamper-proof. A typically example is a cryptographic application.

The approach involves having two redundant MicroBlaze processors with dedicated local memory and redundant comparators, each in a protected area. The outputs from each processor feed two comparators and each processor receive copies of every input signal.

The redundant MicroBlaze processors are functionally identical and completely independent of each other, without any connecting signals. The only exception is debug logic and associated signals, since it is assumed that debugging is disabled before any productization and certification of the system.

The outputs from the master MicroBlaze core drive the peripherals in the system. All data leaving the protected area pass through inhibitors. Each inhibitor is controlled from its associated comparator.

Each protected area of the design must be implemented in its own partition, using a hierarchical Single Chip Cryptography (SCC) flow. A detailed explanation of this flow, and further references, can be found in the document *Hierarchical Design Methodology Guide* (UG748).

A block diagram of the system is shown in Figure 2-41.

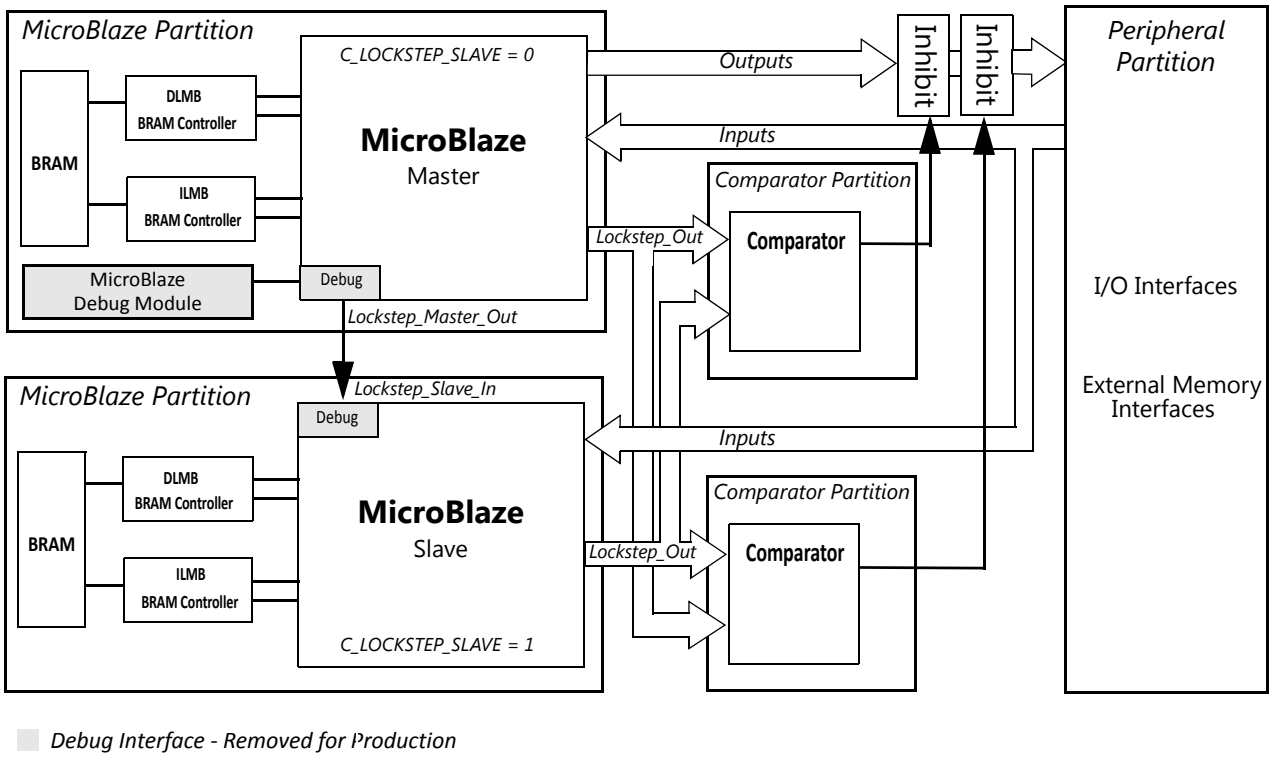

Figure 2-41: **Lockstep Tamper Protection Application**

Error Detection

The error detection use case requires that all transient and permanent faults are detected. This is essential in fail safe and fault tolerant applications, where redundancy is utilized to improve system availability.

In this system two redundant MicroBlaze processors run in lockstep. A comparator is used to signal an error when a mis-match is detected on the outputs of the two processors. Any error immediately causes both processors to halt, preventing further error propagation.

The redundant MicroBlaze processors are functionally identical, except for debug logic and associated signals.The outputs from the master MicroBlaze core drive the peripherals in the system. The slave MicroBlaze core only has inputs connected; all outputs are left open.

The system contains the basic building block for designing a complete fault tolerant application, where one or more additional blocks must be added to provide redundancy.

This use case is illustrated in Figure 2-42.

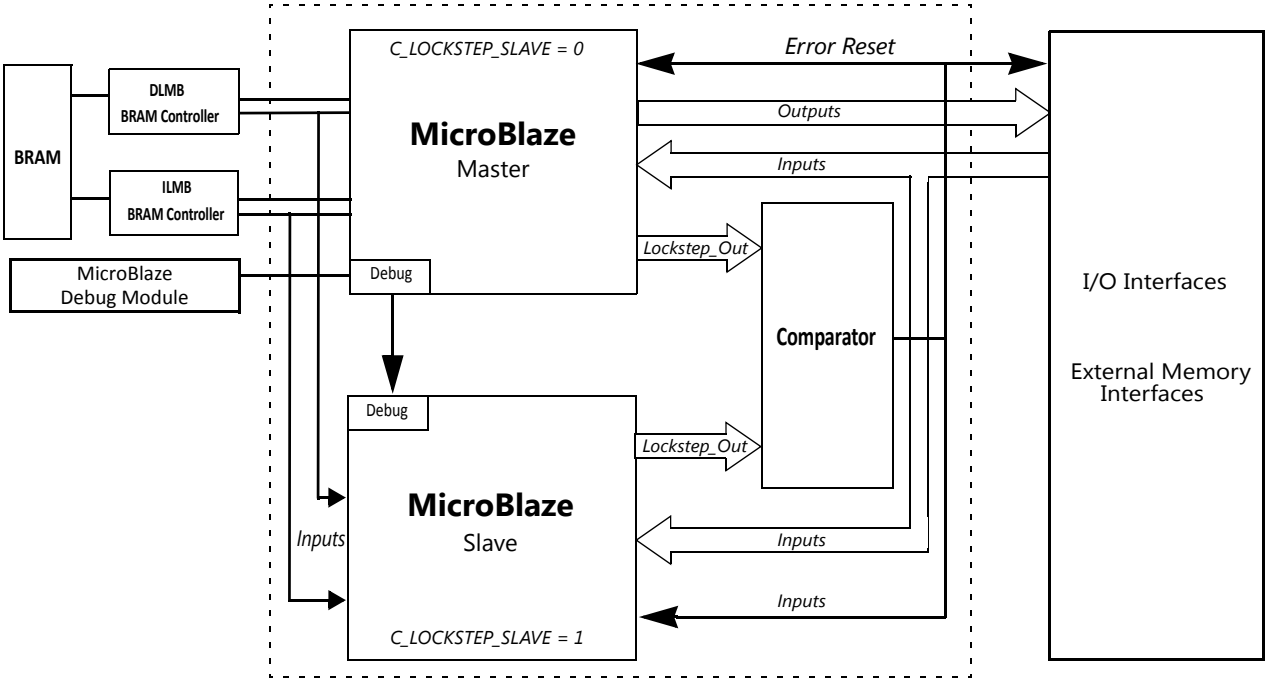

Figure 2-42: **Lockstep Error Detection Application**

Coherency

MicroBlaze supports cache coherency, as well as invalidation of caches and translation look-aside buffers, using the AXI Coherency Extension (ACE) defined in *AMBA® AXI and ACE Protocol Specification* (ARM IHI 0022E). The coherency support is enabled when the parameter C_INTERCONNECT is set to 3 (ACE).

Using ACE ensures coherency between the caches of all MicroBlaze processors in the coherency domain. The peripheral ports (AXI_IP, AXI_DP) and local memory (ILMB, DLMB) are outside the coherency domain.

Coherency is not supported with write-back data cache, wide cache interfaces (more than 32-bit data), instruction cache streams, instruction cache victims or when area optimization is enabled. In addition both C_ICACHE_ALWAYS_USED and C_DCACHE_ALWAYS_USED must be set to 1.

Invalidation

The coherency hardware handles invalidation in the following cases:

- Data Cache invalidation:
  When a MicroBlaze core in the coherency domain invalidates a data cache line with an external cache invalidation instruction (WDC.EXT.CLEAR or WDC.EXT.FLUSH), hardware messages ensure that all other cores in the coherency domain will do the same. The physical address is always used.
- Instruction Cache invalidation:
  When a MicroBlaze core in the coherency domain invalidates an instruction cache line, hardware messages ensure that all other cores in the coherency domain will do the same. When the MMU is in virtual mode the virtual address is used, otherwise the physical address is used.
- MMU TLB invalidation:
  When a MicroBlaze core in the coherency domain invalidates an entry in the UTLB (i.e. writes TLBHI with a zero Valid flag), hardware messages ensure that all other cores in the coherency domain will invalidate all entries in their unified TLBs having a TAG matching the invalidated virtual address, as well as empty their shadow TLBs.

  The TID is not taken into account when matching the entries, which can result in invalidation of entries belonging to other processes. Subsequent accesses to these entries will generate TLB miss exceptions, which must be handled by software.

  Before invalidating an MMU page, it must first be loaded into the UTLB to ensure that the hardware invalidation is propagated within the coherency domain. It is not sufficient to simply invalidate the page in memory, since other processors in the coherency domain may have this particular entry stored in their TLBs.

After a MicroBlaze core has invalidated one or more entries, it must execute a memory barrier instruction (MBAR), to ensure that all peer processors have completed their TLB invalidation.

- Branch Target Cache invalidation:
  When a MicroBlaze core in the coherency domain invalidates the Branch Target Cache, either with a memory barrier instruction or with a synchronizing branch, hardware messages ensure that all other cores in the coherency domain will do the same.

In particular, this means that self-modifying code can be used transparently within the coherency domain in a multi-processor system, provided that the guidelines in"Self-modifying Code" are followed.

Protocol Compliance

The MicroBlaze instruction cache interface issues the following subset of the possible ACE transactions:

- ReadClean
  Issued when a cache line is allocated.
- ReadOnce
  Issued when the cache is off, or if the MMU Inhibit Caching bit is set for the cache line.

The MicroBlaze data cache interface issues the following subset of the possible ACE transactions:

- ReadClean
  Issued when a cache line is allocated.
- CleanUnique
  Issued when an SWX instruction is executed as part of an exclusive access sequence.
- ReadOnce
  Issued when the cache is off, or if the MMU Inhibit Caching bit is set for the cache line.
- WriteUnique
  Issued whenever a store instruction performs a write.
- CleanInvalid
  Issued when a WDC.EXT.FLUSH instruction is executed.
- MakeInvalid
  Issued when a WDC.EXT.CLEAR instruction is executed.

Both interfaces issue the following subset of the possible Distributed Virtual Memory (DVM) transactions:

- DVM Operation
  - TLB Invalidate – Hypervisor TLB Invalidate by VA
  - Branch Predictor Invalidate – Branch Predictor Invalidate all
  - Physical Instruction Cache Invalidate – Non-secure Physical Instruction Cache Invalidate by PA without Virtual Index
  - Virtual Instruction Cache Invalidate – Hypervisor Invalidate by VA
- DVM Sync
  - Synchronization
- DVM Complete
  - In addition to the DVM transactions above, the interfaces only accept the CleanInvalid and MakeInvalid transactions. These transactions have no effect in the instruction cache, and invalidate the indicated data cache lines. If any other transactions are received, the behavior is undefined.
  - Only a subset of AXI4 transactions are utilized by the interfaces, as described in "Cache Interfaces".

MicroBlaze Signal Interface Description

This chapter describes the types of signal interfaces that can be used to connect MicroBlaze™.

Overview

The MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data and instruction accesses. The following two memory interfaces are supported: Local Memory Bus (LMB), and the AMBA® AXI4 interface (AXI4) and ACE interface (ACE). The LMB provides single-cycle access to on-chip dual-port block RAM. The AXI4 interfaces provide a connection to both on-chip and off-chip peripherals and memory. The ACE interfaces provide cache coherent connections to memory. MicroBlaze also supports up to 16 AXI4-Stream interface ports, each with one master and one slave interface.

Features

MicroBlaze can be configured with the following bus interfaces:

- The AMBA AXI4 Interface for peripheral interfaces, and the AMBA AXI4 or AXI Coherency Extension (ACE) Interface for cache interfaces (see ARM® *AMBA® AXI and ACE Protocol Specification,* ARM IHI 0022E).
- LMB provides simple synchronous protocol for efficient block RAM transfers
- AXI4-Stream provides a fast non-arbitrated streaming communication mechanism
- Debug interface for use with the Microprocessor Debug Module (MDM) core
- Trace interface for performance analysis

MicroBlaze I/O Overview

The core interfaces shown in Figure 3-1 and the following Table 3-1 are defined as follows:

M_AXI_DP: Peripheral Data Interface, AXI4-Lite or AXI4 interface
DLMB: Data interface, Local Memory Bus (BRAM only)
M_AXI_IP: Peripheral Instruction interface, AXI4-Lite interface
ILMB: Instruction interface, Local Memory Bus (BRAM only)
M0_AXIS..M15_AXIS: AXI4-Stream interface master direct connection interfaces
S0_AXIS..S15_AXIS: AXI4-Stream interface slave direct connection interfaces
M_AXI_DC: Data side cache AXI4 interface
M_ACE_DC: Data side cache AXI Coherency Extension (ACE) interface
M_AXI_IC: Instruction side cache AXI4 interface
M_ACE_IC: Instruction side cache AXI Coherency Extension (ACE) interface
Core: Miscellaneous signals for: clock, reset, interrupt, debug, trace

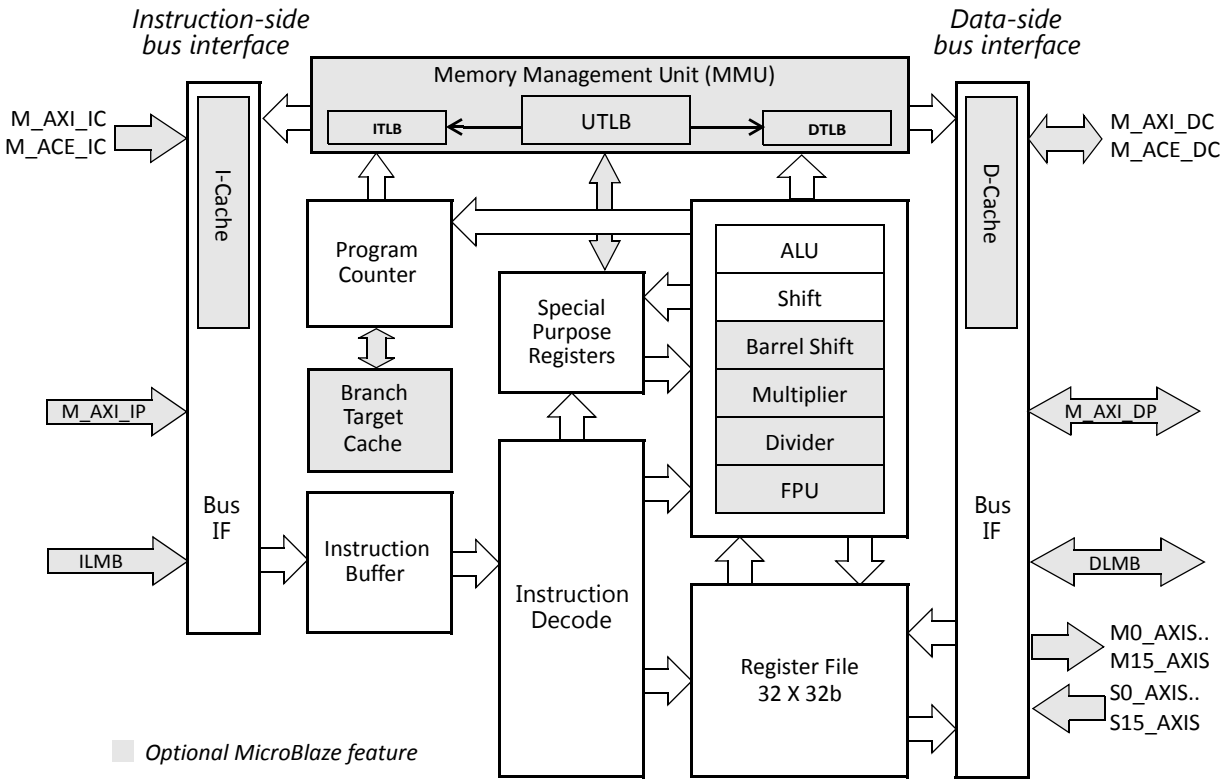

Figure 3-1: **MicroBlaze Core Block Diagram**

Table 3-1: **Summary of MicroBlaze Core I/O**

Signal	Interface	I/O	Description
M_AXI_DP_AWID	M_AXI_DP	O	Master Write address ID
M_AXI_DP_AWADDR	M_AXI_DP	O	Master Write address
M_AXI_DP_AWLEN	M_AXI_DP	O	Master Burst length
M_AXI_DP_AWSIZE	M_AXI_DP	O	Master Burst size
M_AXI_DP_AWBURST	M_AXI_DP	O	Master Burst type
M_AXI_DP_AWLOCK	M_AXI_DP	O	Master Lock type
M_AXI_DP_AWCACHE	M_AXI_DP	O	Master Cache type
M_AXI_DP_AWPROT	M_AXI_DP	O	Master Protection type
M_AXI_DP_AWQOS	M_AXI_DP	O	Master Quality of Service
M_AXI_DP_AWVALID	M_AXI_DP	O	Master Write address valid
M_AXI_DP_AWREADY	M_AXI_DP	I	Slave Write address ready
M_AXI_DP_WDATA	M_AXI_DP	O	Master Write data
M_AXI_DP_WSTRB	M_AXI_DP	O	Master Write strobes
M_AXI_DP_WLAST	M_AXI_DP	O	Master Write last
M_AXI_DP_WVALID	M_AXI_DP	O	Master Write valid
M_AXI_DP_WREADY	M_AXI_DP	I	Slave Write ready
M_AXI_DP_BID	M_AXI_DP	I	Slave Response ID
M_AXI_DP_BRESP	M_AXI_DP	I	Slave Write response
M_AXI_DP_BVALID	M_AXI_DP	I	Slave Write response valid
M_AXI_DP_BREADY	M_AXI_DP	O	Master Response ready
M_AXI_DP_ARID	M_AXI_DP	O	Master Read address ID
M_AXI_DP_ARADDR	M_AXI_DP	O	Master Read address
M_AXI_DP_ARLEN	M_AXI_DP	O	Master Burst length
M_AXI_DP_ARSIZE	M_AXI_DP	O	Master Burst size
M_AXI_DP_ARBURST	M_AXI_DP	O	Master Burst type
M_AXI_DP_ARLOCK	M_AXI_DP	O	Master Lock type
M_AXI_DP_ARCACHE	M_AXI_DP	O	Master Cache type
M_AXI_DP_ARPROT	M_AXI_DP	O	Master Protection type
M_AXI_DP_ARQOS	M_AXI_DP	O	Master Quality of Service
M_AXI_DP_ARVALID	M_AXI_DP	O	Master Read address valid
M_AXI_DP_ARREADY	M_AXI_DP	I	Slave Read address ready

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
M_AXI_DP_RID	M_AXI_DP	I	Slave Read ID tag
M_AXI_DP_RDATA	M_AXI_DP	I	Slave Read data
M_AXI_DP_RRESP	M_AXI_DP	I	Slave Read response
M_AXI_DP_RLAST	M_AXI_DP	I	Slave Read last
M_AXI_DP_RVALID	M_AXI_DP	I	Slave Read valid
M_AXI_DP_RREADY	M_AXI_DP	O	Master Read ready
M_AXI_IP_AWID	M_AXI_IP	O	Master Write address ID
M_AXI_IP_AWADDR	M_AXI_IP	O	Master Write address
M_AXI_IP_AWLEN	M_AXI_IP	O	Master Burst length
M_AXI_IP_AWSIZE	M_AXI_IP	O	Master Burst size
M_AXI_IP_AWBURST	M_AXI_IP	O	Master Burst type
M_AXI_IP_AWLOCK	M_AXI_IP	O	Master Lock type
M_AXI_IP_AWCACHE	M_AXI_IP	O	Master Cache type
M_AXI_IP_AWPROT	M_AXI_IP	O	Master Protection type
M_AXI_IP_AWQOS	M_AXI_IP	O	Master Quality of Service
M_AXI_IP_AWVALID	M_AXI_IP	O	Master Write address valid
M_AXI_IP_AWREADY	M_AXI_IP	I	Slave Write address ready
M_AXI_IP_WDATA	M_AXI_IP	O	Master Write data
M_AXI_IP_WSTRB	M_AXI_IP	O	Master Write strobes
M_AXI_IP_WLAST	M_AXI_IP	O	Master Write last
M_AXI_IP_WVALID	M_AXI_IP	O	Master Write valid
M_AXI_IP_WREADY	M_AXI_IP	I	Slave Write ready
M_AXI_IP_BID	M_AXI_IP	I	Slave Response ID
M_AXI_IP_BRESP	M_AXI_IP	I	Slave Write response
M_AXI_IP_BVALID	M_AXI_IP	I	Slave Write response valid
M_AXI_IP_BREADY	M_AXI_IP	O	Master Response ready
M_AXI_IP_ARID	M_AXI_IP	O	Master Read address ID
M_AXI_IP_ARADDR	M_AXI_IP	O	Master Read address
M_AXI_IP_ARLEN	M_AXI_IP	O	Master Burst length
M_AXI_IP_ARSIZE	M_AXI_IP	O	Master Burst size
M_AXI_IP_ARBURST	M_AXI_IP	O	Master Burst type
M_AXI_IP_ARLOCK	M_AXI_IP	O	Master Lock type

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
M_AXI_IP_ARCACHE	M_AXI_IP	O	Master Cache type
M_AXI_IP_ARPROT	M_AXI_IP	O	Master Protection type
M_AXI_IP_ARQOS	M_AXI_IP	O	Master Quality of Service
M_AXI_IP_ARVALID	M_AXI_IP	O	Master Read address valid
M_AXI_IP_ARREADY	M_AXI_IP	I	Slave Read address ready
M_AXI_IP_RID	M_AXI_IP	I	Slave Read ID tag
M_AXI_IP_RDATA	M_AXI_IP	I	Slave Read data
M_AXI_IP_RRESP	M_AXI_IP	I	Slave Read response
M_AXI_IP_RLAST	M_AXI_IP	I	Slave Read last
M_AXI_IP_RVALID	M_AXI_IP	I	Slave Read valid
M_AXI_IP_RREADY	M_AXI_IP	O	Master Read ready
M_AXI_DC_AWADDR	M_AXI_DC	O	Master Write address
M_AXI_DC_AWLEN	M_AXI_DC	O	Master Burst length
M_AXI_DC_AWSIZE	M_AXI_DC	O	Master Burst size
M_AXI_DC_AWBURST	M_AXI_DC	O	Master Burst type
M_AXI_DC_AWLOCK	M_AXI_DC	O	Master Lock type
M_AXI_DC_AWCACHE	M_AXI_DC	O	Master Cache type
M_AXI_DC_AWPROT	M_AXI_DC	O	Master Protection type
M_AXI_DC_AWQOS	M_AXI_DC	O	Master Quality of Service
M_AXI_DC_AWVALID	M_AXI_DC	O	Master Write address valid
M_AXI_DC_AWREADY	M_AXI_DC	I	Slave Write address ready
M_AXI_DC_AWUSER	M_AXI_DC	O	Master Write address user signals
M_AXI_DC_AWDOMAIN	M_ACE_DC	O	Master Write address domain
M_AXI_DC_AWSNOOP	M_ACE_DC	O	Master Write address snoop
M_AXI_DC_AWBAR	M_ACE_DC	O	Master Write address barrier
M_AXI_DC_WDATA	M_AXI_DC	O	Master Write data
M_AXI_DC_WSTRB	M_AXI_DC	O	Master Write strobes
M_AXI_DC_WLAST	M_AXI_DC	O	Master Write last
M_AXI_DC_WVALID	M_AXI_DC	O	Master Write valid
M_AXI_DC_WREADY	M_AXI_DC	I	Slave Write ready
M_AXI_DC_WUSER	M_AXI_DC	O	Master Write user signals
M_AXI_DC_BRESP	M_AXI_DC	I	Slave Write response

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
M_AXI_DC_BID	M_AXI_DC	I	Slave Response ID
M_AXI_DC_BVALID	M_AXI_DC	I	Slave Write response valid
M_AXI_DC_BREADY	M_AXI_DC	O	Master Response ready
M_AXI_DC_BUSER	M_AXI_DC	I	Slave Write response user signals
M_AXI_DC_WACK	M_ACE_DC	O	Slave Write acknowledge
M_AXI_DC_ARID	M_AXI_DC	O	Master Read address ID
M_AXI_DC_ARADDR	M_AXI_DC	O	Master Read address
M_AXI_DC_ARLEN	M_AXI_DC	O	Master Burst length
M_AXI_DC_ARSIZE	M_AXI_DC	O	Master Burst size
M_AXI_DC_ARBURST	M_AXI_DC	O	Master Burst type
M_AXI_DC_ARLOCK	M_AXI_DC	O	Master Lock type
M_AXI_DC_ARCACHE	M_AXI_DC	O	Master Cache type
M_AXI_DC_ARPROT	M_AXI_DC	O	Master Protection type
M_AXI_DC_ARQOS	M_AXI_DC	O	Master Quality of Service
M_AXI_DC_ARVALID	M_AXI_DC	O	Master Read address valid
M_AXI_DC_ARREADY	M_AXI_DC	I	Slave Read address ready
M_AXI_DC_ARUSER	M_AXI_DC	O	Master Read address user signals
M_AXI_DC_ARDOMAIN	M_ACE_DC	O	Master Read address domain
M_AXI_DC_ARSNOOP	M_ACE_DC	O	Master Read address snoop
M_AXI_DC_ARBAR	M_ACE_DC	O	Master Read address barrier
M_AXI_DC_RID	M_AXI_DC	I	Slave Read ID tag
M_AXI_DC_RDATA	M_AXI_DC	I	Slave Read data
M_AXI_DC_RRESP	M_AXI_DC	I	Slave Read response
M_AXI_DC_RLAST	M_AXI_DC	I	Slave Read last
M_AXI_DC_RVALID	M_AXI_DC	I	Slave Read valid
M_AXI_DC_RREADY	M_AXI_DC	O	Master Read ready
M_AXI_DC_RUSER	M_AXI_DC	I	Slave Read user signals
M_AXI_DC_RACK	M_ACE_DC	O	Master Read acknowledge
M_AXI_DC_ACVALID	M_ACE_DC	I	Slave Snoop address valid
M_AXI_DC_ACADDR	M_ACE_DC	I	Slave Snoop address
M_AXI_DC_ACSNOOP	M_ACE_DC	I	Slave Snoop address snoop
M_AXI_DC_ACPROT	M_ACE_DC	I	Slave Snoop address protection type

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
M_AXI_DC_ACREADY	M_ACE_DC	O	Master Snoop ready
M_AXI_DC_CRREADY	M_ACE_DC	I	Slave Snoop response ready
M_AXI_DC_CRVALID	M_ACE_DC	O	Master Snoop response valid
M_AXI_DC_CRRESP	M_ACE_DC	O	Master Snoop response
M_AXI_DC_CDVALID	M_ACE_DC	O	Master Snoop data valid
M_AXI_DC_CDREADY	M_ACE_DC	I	Slave Snoop data ready
M_AXI_DC_CDDATA	M_ACE_DC	O	Master Snoop data
M_AXI_DC_CDLAST	M_ACE_DC	O	Master Snoop data last
M_AXI_IC_AWID	M_AXI_IC	O	Master Write address ID
M_AXI_IC_AWADDR	M_AXI_IC	O	Master Write address
M_AXI_IC_AWLEN	M_AXI_IC	O	Master Burst length
M_AXI_IC_AWSIZE	M_AXI_IC	O	Master Burst size
M_AXI_IC_AWBURST	M_AXI_IC	O	Master Burst type
M_AXI_IC_AWLOCK	M_AXI_IC	O	Master Lock type
M_AXI_IC_AWCACHE	M_AXI_IC	O	Master Cache type
M_AXI_IC_AWPROT	M_AXI_IC	O	Master Protection type
M_AXI_IC_AWQOS	M_AXI_IC	O	Master Quality of Service
M_AXI_IC_AWVALID	M_AXI_IC	O	Master Write address valid
M_AXI_IC_AWREADY	M_AXI_IC	I	Slave Write address ready
M_AXI_IC_AWUSER	M_AXI_IC	O	Master Write address user signals
M_AXI_IC_AWDOMAIN	M_ACE_IC	O	Master Write address domain
M_AXI_IC_AWSNOOP	M_ACE_IC	O	Master Write address snoop
M_AXI_IC_AWBAR	M_ACE_IC	O	Master Write address barrier
M_AXI_IC_WDATA	M_AXI_IC	O	Master Write data
M_AXI_IC_WSTRB	M_AXI_IC	O	Master Write strobes
M_AXI_IC_WLAST	M_AXI_IC	O	Master Write last
M_AXI_IC_WVALID	M_AXI_IC	O	Master Write valid
M_AXI_IC_WREADY	M_AXI_IC	I	Slave Write ready
M_AXI_IC_WUSER	M_AXI_IC	O	Master Write user signals
M_AXI_IC_BID	M_AXI_IC	I	Slave Response ID
M_AXI_IC_BRESP	M_AXI_IC	I	Slave Write response
M_AXI_IC_BVALID	M_AXI_IC	I	Slave Write response valid

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
M_AXI_IC_BREADY	M_AXI_IC	O	Master Response ready
M_AXI_IC_BUSER	M_AXI_IC	I	Slave Write response user signals
M_AXI_IC_WACK	M_ACE_IC	O	Slave Write acknowledge
M_AXI_IC_ARID	M_AXI_IC	O	Master Read address ID
M_AXI_IC_ARADDR	M_AXI_IC	O	Master Read address
M_AXI_IC_ARLEN	M_AXI_IC	O	Master Burst length
M_AXI_IC_ARSIZE	M_AXI_IC	O	Master Burst size
M_AXI_IC_ARBURST	M_AXI_IC	O	Master Burst type
M_AXI_IC_ARLOCK	M_AXI_IC	O	Master Lock type
M_AXI_IC_ARCACHE	M_AXI_IC	O	Master Cache type
M_AXI_IC_ARPROT	M_AXI_IC	O	Master Protection type
M_AXI_IC_ARQOS	M_AXI_IC	O	Master Quality of Service
M_AXI_IC_ARVALID	M_AXI_IC	O	Master Read address valid
M_AXI_IC_ARREADY	M_AXI_IC	I	Slave Read address ready
M_AXI_IC_ARUSER	M_AXI_IC	O	Master Read address user signals
M_AXI_IC_ARDOMAIN	M_ACE_IC	O	Master Read address domain
M_AXI_IC_ARSNOOP	M_ACE_IC	O	Master Read address snoop
M_AXI_IC_ARBAR	M_ACE_IC	O	Master Read address barrier
M_AXI_IC_RID	M_AXI_IC	I	Slave Read ID tag
M_AXI_IC_RDATA	M_AXI_IC	I	Slave Read data
M_AXI_IC_RRESP	M_AXI_IC	I	Slave Read response
M_AXI_IC_RLAST	M_AXI_IC	I	Slave Read last
M_AXI_IC_RVALID	M_AXI_IC	I	Slave Read valid
M_AXI_IC_RREADY	M_AXI_IC	O	Master Read ready
M_AXI_IC_RUSER	M_AXI_IC	I	Slave Read user signals
M_AXI_IC_RACK	M_ACE_IC	O	Master Read acknowledge
M_AXI_IC_ACVALID	M_ACE_IC	I	Slave Snoop address valid
M_AXI_IC_ACADDR	M_ACE_IC	I	Slave Snoop address
M_AXI_IC_ACSNOOP	M_ACE_IC	I	Slave Snoop address snoop
M_AXI_IC_ACPROT	M_ACE_IC	I	Slave Snoop address protection type
M_AXI_IC_ACREADY	M_ACE_IC	O	Master Snoop ready
M_AXI_IC_CRREADY	M_ACE_IC	I	Slave Snoop response ready

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
M_AXI_IC_CRVALID	M_ACE_IC	O	Master Snoop response valid
M_AXI_IC_CRRESP	M_ACE_IC	O	Master Snoop response
M_AXI_IC_CDVALID	M_ACE_IC	O	Master Snoop data valid
M_AXI_IC_CDREADY	M_ACE_IC	I	Slave Snoop data ready
M_AXI_IC_CDDATA	M_ACE_IC	O	Master Snoop data
M_AXI_IC_CDLAST	M_ACE_IC	O	Master Snoop data last
Data_Addr[0:31]	DLMB	O	Data interface LMB address bus
Byte_Enable[0:3]	DLMB	O	Data interface LMB byte enables
Data_Write[0:31]	DLMB	O	Data interface LMB write data bus
D_AS	DLMB	O	Data interface LMB address strobe
Read_Strobe	DLMB	O	Data interface LMB read strobe
Write_Strobe	DLMB	O	Data interface LMB write strobe
Data_Read[0:31]	DLMB	I	Data interface LMB read data bus
DReady	DLMB	I	Data interface LMB data ready
DWait	DLMB	I	Data interface LMB data wait
DCE	DLMB	I	Data interface LMB correctable error
DUE	DLMB	I	Data interface LMB uncorrectable error
Instr_Addr[0:31]	ILMB	O	Instruction interface LMB address bus
I_AS	ILMB	O	Instruction interface LMB address strobe
IFetch	ILMB	O	Instruction interface LMB instruction fetch
Instr[0:31]	ILMB	I	Instruction interface LMB read data bus
IReady	ILMB	I	Instruction interface LMB data ready
IWait	ILMB	I	Instruction interface LMB data wait
ICE	ILMB	I	Instruction interface LMB correctable error
IUE	ILMB	I	Instruction interface LMB uncorrectable error
Mn_AXIS_TLAST	M0_AXIS.. M15_AXIS	O	Master interface output AXI4 channels write last
Mn_AXIS_TDATA	M0_AXIS.. M15_AXIS	O	Master interface output AXI4 channels write data
Mn_AXIS_TVALID	M0_AXIS.. M15_AXIS	O	Master interface output AXI4 channels write valid
Mn_AXIS_TREADY	M0_AXIS.. M15_AXIS	I	Master interface input AXI4 channels write ready

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
Sn_AXIS_TLAST	S0_AXIS.. S15_AXIS	I	Slave interface input AXI4 channels write last
Sn_AXIS_TDATA	S0_AXIS.. S15_AXIS	I	Slave interface input AXI4 channels write data
Sn_AXIS_TVALID	S0_AXIS.. S15_AXIS	I	Slave interface input AXI4 channels write valid
Sn_AXIS_TREADY	S0_AXIS.. S15_AXIS	O	Slave interface output AXI4 channels write ready
Interrupt	Core	I	Interrupt
Interrupt_Address[1]	Core	I	Interrupt vector address
Interrupt_Ack[1]	Core	O	Interrupt acknowledge
Reset	Core	I	Core reset, active high. Should be held for at least 1 Clk clock cycle.
Reset_Mode[0:1]	Core	I	Reset mode. Sampled when Reset is active. SeeTable 3-2 for details.
Clk	Core	I	Clock[2]
Ext_BRK	Core	I	Break signal from MDM
Ext_NM_BRK	Core	I	Non-maskable break signal from MDM
MB_Halted	Core	O	Pipeline is halted, either via the Debug Interface, by setting Dbg_Stop, or by setting Reset_Mode[0:1] to 10.
Dbg_Stop	Core	I	Unconditionally force pipeline to halt as soon as possible. Rising-edge detected pulse that should be held for at least 1 Clk clock cycle. The signal only has any effect when C_DEBUG_ENABLED is greater than 0.
Dbg_Intr	Core	O	Debug interrupt output, set when a Performance Monitor counter overflows, available when C_DEBUG_ENABLED is set to 2 (Extended).
MB_Error	Core	O	Pipeline is halted due to a missed exception, when C_FAULT_TOLERANT is set to 1.
Sleep	Core	O	MicroBlaze is in sleep mode after executing a SLEEP instruction or by setting Reset_Mode[0:1] to 10, all external accesses are completed, and the pipeline is halted.
Wakeup[0:1]	Core	I	Wake MicroBlaze from sleep mode when either or both bits are set to 1. Ignored if MicroBlaze is not in sleep mode.

Table 3-1: **Summary of MicroBlaze Core I/O *(Cont'd)***

Signal	Interface	I/O	Description
Dbg_Wakeup	Core	O	Debug request that external logic should wake MicroBlaze from sleep mode with the Wakeup signal.
Lockstep_...	Core	IO	Lockstep signals for high integrity applications. See Table 3-11 for details.
Dbg_...	Core	IO	Debug signals from MDM. See Table 3-13 for details.
Trace_...	Core	O	Trace signals for real time HW analysis. See Table 3-14 for details.

1. Only used with C_USE_INTERRUPT = 2, for low-latency interrupt support.
2. MicroBlaze is a synchronous design clocked with the Clk signal, except for hardware debug logic, which is clocked with the Dbg_Clk signal. If hardware debug logic is not used, there is no minimum frequency limit for Clk. However, if hardware debug logic is used, there are signals transferred between the two clock regions. In this case Clk must have a higher frequency than Dbg_Clk.

Table 3-2: **Effect of Reset Mode inputs**

Reset_Mode[0:1]	Description
00	MicroBlaze starts executing at the reset vector, defined by C_BASE_VECTORS. This is the nominal default behavior.
01	MicroBlaze immediately enters sleep mode without performing any bus access, just as if a SLEEP instruction had been executed. The Sleep output is set to 1. When any of the Wakeup[0:1] signals is set, MicroBlaze starts executing at the reset vector, defined by C_BASE_VECTORS. This functionality can be useful in a multiprocessor configuration, allowing secondary processors to be configured without LMB memory.
10	If C_DEBUG_ENABLED is 0, the behavior is the same as if Reset_Mode[0:1] = 00. If C_DEBUG_ENABLED is greater than 0, MicroBlaze immediately enters debug halt without performing any bus access, and the MB_Halted output is set to 1. When execution is continued via the debug interface, MicroBlaze starts executing at the reset vector, defined by C_BASE_VECTORS.
11	*Reserved*

AXI4 and ACE Interface Description

Memory Mapped Interfaces

Peripheral Interfaces

The MicroBlaze AXI4 memory mapped peripheral interfaces are implemented as 32-bit masters. Each of these interfaces only have a single outstanding transaction at any time, and all transactions are completed in order.

- The instruction peripheral interface (M_AXI_IP) only performs single word read accesses, and is always set to use the AXI4-Lite subset.
- The data peripheral interface (M_AXI_DP) performs single word accesses, and is set to use the AXI4-Lite subset as default, but is set to use AXI4 when enabling exclusive access for LWX and SWX instructions. Halfword and byte writes are performed by setting the appropriate byte strobes.

Cache Interfaces

The AXI4 memory mapped cache interfaces are implemented either as 32-bit, 128-bit, 256-bit, or 512-bit masters, depending on cache line length and data width parameters, whereas the AXI Coherency Extension (ACE) interfaces are implemented as 32-bit masters.

- With a 32-bit master, the instruction cache interface (M_AXI_IC or M_ACE_IC) performs 4 word, 8 word or 16 word burst read accesses, depending on cache line length. With 128-bit, 256-bit, or 512-bit masters, only single read accesses are performed.

  With a 32-bit master, this interface can have multiple outstanding transactions, issuing up to 2 transactions or up to 5 transactions when stream cache is enabled. The stream cache can request two cache lines in advance, which means that in some cases 5 outstanding transactions can occur. In this case the number of outstanding reads is set to 8, since this must be a power of two. With 128-bit, 256-bit, or 512-bit masters, the interface only has a single outstanding transaction.

  How memory locations are accessed depend on the parameter `C_ICACHE_ALWAYS_USED`. If the parameter is 1, the cached memory range is always accessed via the AXI4 or ACE cache interface. If the parameter is 0, the cached memory range is accessed over the AXI4 peripheral interface when the caches are software disabled (that is, MSR[ICE]=0).

- With a 32-bit master, the data cache interface (M_AXI_DC or M_ACE_DC) performs single word accesses, as well as 4 word, 8 word or 16 word burst accesses, depending on cache line length. Burst write accesses are only performed when using write-back

cache with AXI4. With 128-bit, 256-bit, or 512-bit AXI4 masters, only single accesses are performed.

This interface can have multiple outstanding transactions, either issuing up to 2 transactions when reading, or up to 32 transactions when writing. MicroBlaze ensures that all outstanding writes are completed before a read is issued, since the processor must maintain an ordered memory model but AXI4 or ACE has separate read/write channels without any ordering. Using up to 32 outstanding write transactions improves performance, since it allows multiple writes to proceed without stalling the pipeline.

Word, halfword and byte writes are performed by setting the appropriate byte strobes.

Exclusive accesses can be enabled for LWX and SWX instructions.

How memory locations are accessed depend on the parameter C_DCACHE_ALWAYS_USED. If the parameter is 1, the cached memory range is always accessed via the AXI4 or ACE cache interface. If the parameter is 0, the cached memory range is accessed over the AXI4 peripheral interface when the caches are software disabled (that is, MSR[DCE]=0).

Interface Parameters and Signals

The relationship between MicroBlaze parameter settings and AXI4 interface behavior for tool-assigned parameters is summarized in Table 3-3.

Table 3-3: **AXI Memory Mapped Interface Parameters**

Interface	Parameter	Description
M_AXI_DP	C_M_AXI_DP_PROTOCOL	**AXI4-Lite:** Default. **AXI4:** Used to allow exclusive access when C_M_AXI_DP_EXCLUSIVE_ACCESS is 1.
M_AXI_IC M_ACE_IC	C_M_AXI_IC_DATA_WIDTH	**32:** Default, single word accesses and burst accesses with C_ICACHE_LINE_LEN word busts used with AXI4 and ACE. **128:** Used when C_ICACHE_DATA_WIDTH is set to 1 and C_ICACHE_LINE_LEN is set to 4 with AXI4. Only single accesses can occur. **256:** Used when C_ICACHE_DATA_WIDTH is set to 1 and C_ICACHE_LINE_LEN is set to 8 with AXI4. Only single accesses can occur. **512:** Used when C_ICACHE_DATA_WIDTH is set to 2, or when it is set to 1 and C_ICACHE_LINE_LEN is set to 16 with AXI4. Only single accesses can occur.

Table 3-3: **AXI Memory Mapped Interface Parameters *(Cont'd)***

Interface	Parameter	Description
M_AXI_DC M_ACE_DC	C_M_AXI_DC_DATA_WIDTH	**32:** Default, single word accesses and burst accesses with C_DCACHE_LINE_LEN word busts used with AXI4 and ACE. Write bursts are only used with AXI4 when C_DCACHE_USE_WRITEBACK is set to 1. **128:** Used when C_DCACHE_DATA_WIDTH is set to 1 and C_DCACHE_LINE_LEN is set to 4 with AXI4. Only single accesses can occur. **256:** Used when C_DCACHE_DATA_WIDTH is set to 1 and C_DCACHE_LINE_LEN is set to 8 with AXI4. Only single accesses can occur. **512:** Used when C_DCACHE_DATA_WIDTH is set to 2, or when it is set to 1 and C_DCACHE_LINE_LEN is set to 16 with AXI4. Only single accesses can occur.
M_AXI_IC M_ACE_IC	NUM_READ_OUTSTANDING	**1:** Default for 128-bit, 256-bit and 512-bit masters, a single outstanding read. **2:** Default for 32-bit masters, 2 simultaneous outstanding reads. **8:** Used for 32-bit masters when C_ICACHE_STREAMS is set to 1, allowing 8 simultaneous outstanding reads. Can be set to 1, 2, or 8.
M_AXI_DC M_ACE_DC	NUM_READ_OUTSTANDING	**1:** Default for 128-bit, 256-bit and 512-bit masters, a single outstanding read. **2:** Default for 32-bit masters, 2 simultaneous outstanding reads. Can be set to 1 or 2.
M_AXI_DC M_ACE_DC	NUM_WRITE_OUTSTANDING	**32:** Default, 32 simultaneous outstanding writes. Can be set to 1, 2, 4, 8, 16, or 32.

Values for access permissions, memory types, quality of service and shareability domain are defined in Table 3-4.

Table 3-4: **AXI Interface Signal Definitions**

Interface	Signal	Description
M_AXI_IP	C_M_AXI_IP_ARPROT	Access Permission: • Unprivileged, secure instruction access (100)
M_AXI_DP	C_M_AXI_DP_ARCACHE C_M_AXI_DP_AWCACHE	Memory Type, AXI4 protocol: • Normal Non-cacheable Bufferable (0011)
	C_M_AXI_DP_ARPROT C_M_AXI_DP_AWPROT	Access Permission, AXI4 and AXI4-Lite protocol: • Unprivileged, secure data access (000)
	C_M_AXI_DP_ARQOS C_M_AXI_DP_AWQOS	Quality of Service, AXI4 protocol: • Priority 8 (1000)
M_AXI_IC	C_M_AXI_IC_ARCACHE	Memory Type: • Write-back Read and Write-allocate (1111)

Table 3-4: **AXI Interface Signal Definitions**

Interface	Signal	Description
M_ACE_IC	C_M_AXI_IC_ARCACHE	Memory Type, normal access: • Write-back Read and Write-allocate (1111) Memory Type, DVM access: • Normal Non-cacheable Non-bufferable (0010)
	C_M_AXI_IC_ARDOMAIN	Shareability Domain: • Inner shareable (01)
M_AXI_IC M_ACE_IC	C_M_AXI_IC_ARPROT	Access Permission: • Unprivileged, secure instruction access (100)
	C_M_AXI_IC_ARQOS	Quality of Service: • Priority 7 (0111)
M_AXI_DC	C_M_AXI_DC_ARCACHE	Memory Type, normal access: • Write-back Read and Write-allocate (1111) Memory Type, exclusive access: • Normal Non-cacheable Non-bufferable (0010)
M_ACE_DC	C_M_AXI_DC_ARCACHE	Memory Type, normal and exclusive access: • Write-back Read and Write-allocate (1111) Memory Type, DVM access: • Normal Non-cacheable Non-bufferable (0010)
	C_M_AXI_DC_ARDOMAIN C_M_AXI_DC_AWDOMAIN	Shareability Domain: • Inner shareable (01)
M_AXI_DC M_ACE_DC	C_M_AXI_DC_AWCACHE	Memory Type, normal access: • Write-back Read and Write-allocate (1111) Memory Type, exclusive access: • Normal Non-cacheable Non-bufferable (0010)
	C_M_AXI_DC_ARPROT C_M_AXI_DC_AWPROT	Access Permission: • Unprivileged, secure data access (000)
	C_M_AXI_DC_ARQOS	Quality of Service, read access: • Priority 12 ((1100)
	C_M_AXI_DC_AWQOS	Quality of Service, write access: • Priority 8 (1000)

Please refer to the *AMBA AXI and ACE Protocol Specification* (ARM IHI 0022E) document for details.

Stream Interfaces

The MicroBlaze AXI4-Stream interfaces (M0_AXIS..M15_AXIS, S0_AXIS..S15_AXIS) are implemented as 32-bit masters and slaves. Please refer to the *AMBA 4 AXI4-Stream Protocol Specification, Version 1.0* (ARM IHI 0051A) document for further details.

Write Operation

A write to the stream interface is performed by MicroBlaze using one of the put or putd instructions. A write operation transfers the register contents to an output AXI4 interface. The transfer is completed in a single clock cycle for blocking mode writes (put and cput instructions) as long as the interface is not busy. If the interface is busy, the processor stalls until it becomes available. The non-blocking instructions (with prefix n), always complete in a single clock cycle even if the interface is busy. If the interface was busy, the write is inhibited and the carry bit is set in the MSR.

The control instructions (with prefix c) set the AXI4-Stream TLAST output, to '1', which is used to indicate the boundary of a packet.

Read Operation

A read from the stream interface is performed by MicroBlaze using one of the get or getd instructions. A read operations transfers the contents of an input AXI4 interface to a general purpose register. The transfer is typically completed in 2 clock cycles for blocking mode reads as long as data is available. If data is not available, the processor stalls at this instruction until it becomes available. In the non-blocking mode (instructions with prefix n), the transfer is completed in one or two clock cycles irrespective of whether or not data was available. In case data was not available, the transfer of data does not take place and the carry bit is set in the MSR.

The data get instructions (without prefix c) expect the AXI4-Stream TLAST input to be cleared to '0', otherwise the instructions will set MSR[FSL] to '1'. Conversely, the control get instructions (with prefix c) expect the TLAST input to be set to '1', otherwise the instructions will set MSR[FSL] to '1'. This can be used to check for the boundary of a packet.

Local Memory Bus (LMB) Interface Description

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

LMB Signal Interface

Table 3-5: **LMB Bus Signals**

Signal	Data Interface	Instruction Interface	Type	Description
Addr[0:31]	Data_Addr[0:31]	Instr_Addr[0:31]	O	Address bus
Byte_Enable[0:3]	Byte_Enable[0:3]	*not used*	O	Byte enables
Data_Write[0:31]	Data_Write[0:31]	*not used*	O	Write data bus
AS	D_AS	I_AS	O	Address strobe
Read_Strobe	Read_Strobe	IFetch	O	Read in progress
Write_Strobe	Write_Strobe	*not used*	O	Write in progress
Data_Read[0:31]	Data_Read[0:31]	Instr[0:31]	I	Read data bus
Ready	DReady	IReady	I	Ready for next transfer
Wait[1]	DWait	IWait	I	Wait until accepted transfer is ready
CE[1]	DCE	ICE	I	Correctable error
UE[1]	DUE	IUE	I	Uncorrectable error
Clk	Clk	Clk	I	Bus clock

1. Added in LMB for MicroBlaze v8.00

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 3-6: **Valid Values for Byte_Enable[0:3]**

Byte_Enable[0:3]	Byte Lanes Used			
	Data[0:7]	Data[8:15]	Data[16:23]	Data[24:31]
0001				●
0010			●	
0100		●		
1000	●			
0011			●	●
1100	●	●		
1111	●	●	●	●

Data_Write[0:31]

The write data bus is an output from the core and contains the data that is written to memory. It is valid only when AS is 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 may remain high until the clock cycle after Ready is sampled high. If a new read transfer is directly started in the next clock cycle, 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 may remain high until the clock cycle after Ready is sampled high. If a new write transfer is directly started in the next clock cycle, 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` 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.

Wait

The Wait signal is an input to the core and indicates that the current transfer has been accepted, but not yet completed. It is sampled on the rising edge of the clock.

CE

The CE signal is an input to the core and indicates that the current transfer had a correctable error. It is valid on the rising edge of the clock when Ready is high. For reads, this signal indicates that an error has been corrected on the Data_Read[0:31] bus, and for byte and halfword writes it indicates that the corresponding data word in local memory has been corrected before writing the new data.

UE

The UE signal is an input to the core and indicates that the current transfer had an uncorrectable error. It is valid on the rising edge of the clock when Ready is high. For reads, this signal indicates that the value of the Data_Read[0:31] bus is erroneous, and for byte and halfword writes it indicates that the corresponding data word in local memory was erroneous before writing the new data.

Clk

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

LMB Transactions

The following diagrams provide examples of LMB bus operations.

Generic Write Operations

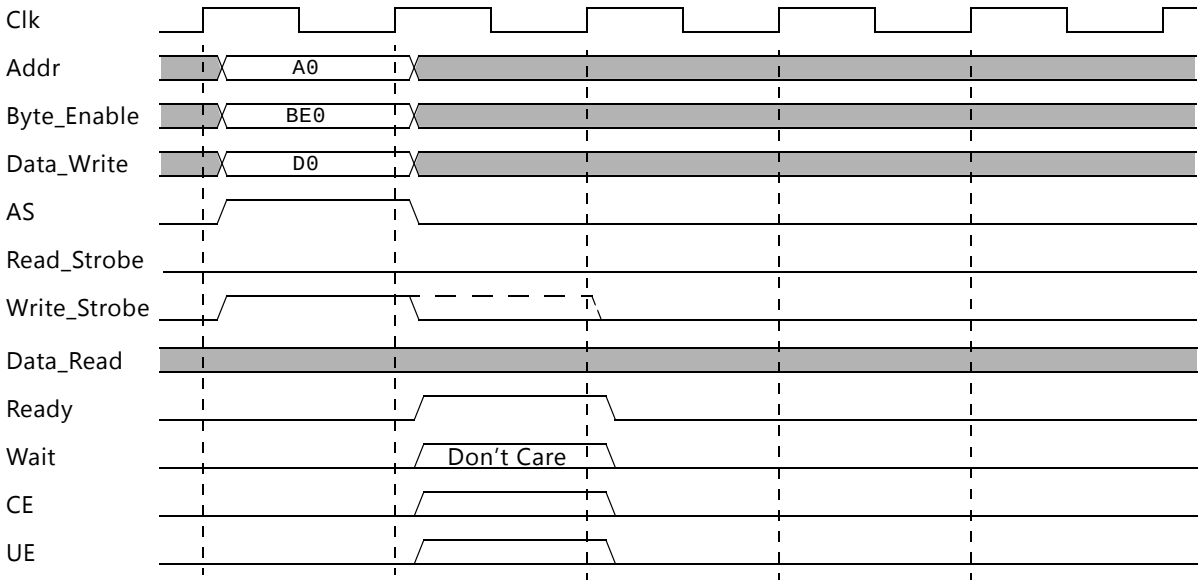

Figure 3-2: **LMB Generic Write Operation, 0 Wait States**

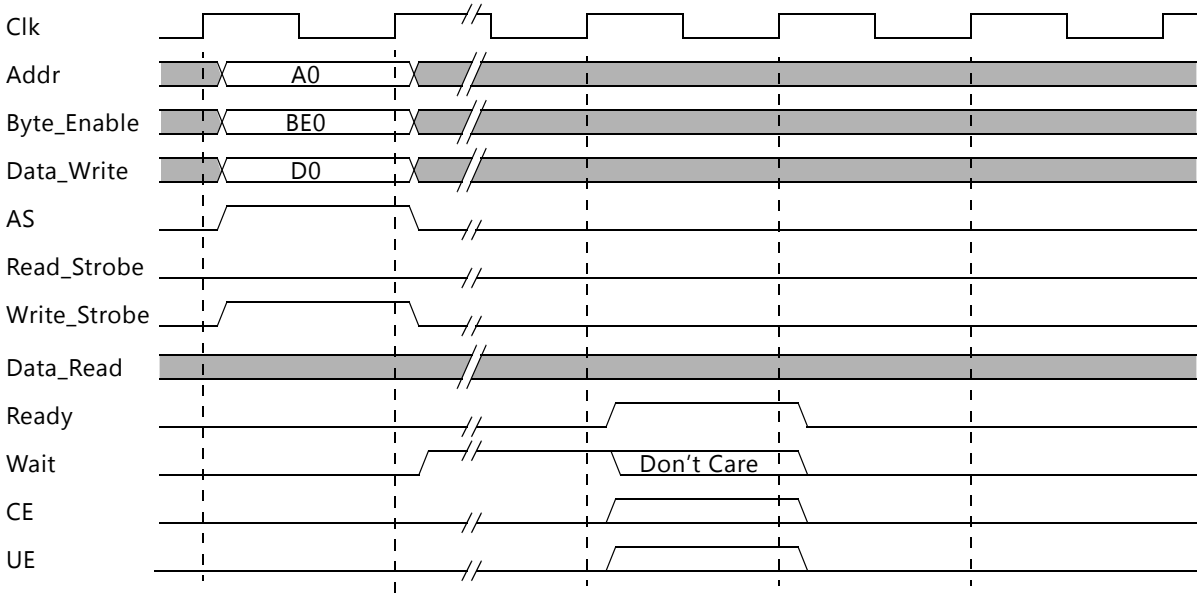

Figure 3-3: **LMB Generic Write Operation, N Wait States**

Generic Read Operations

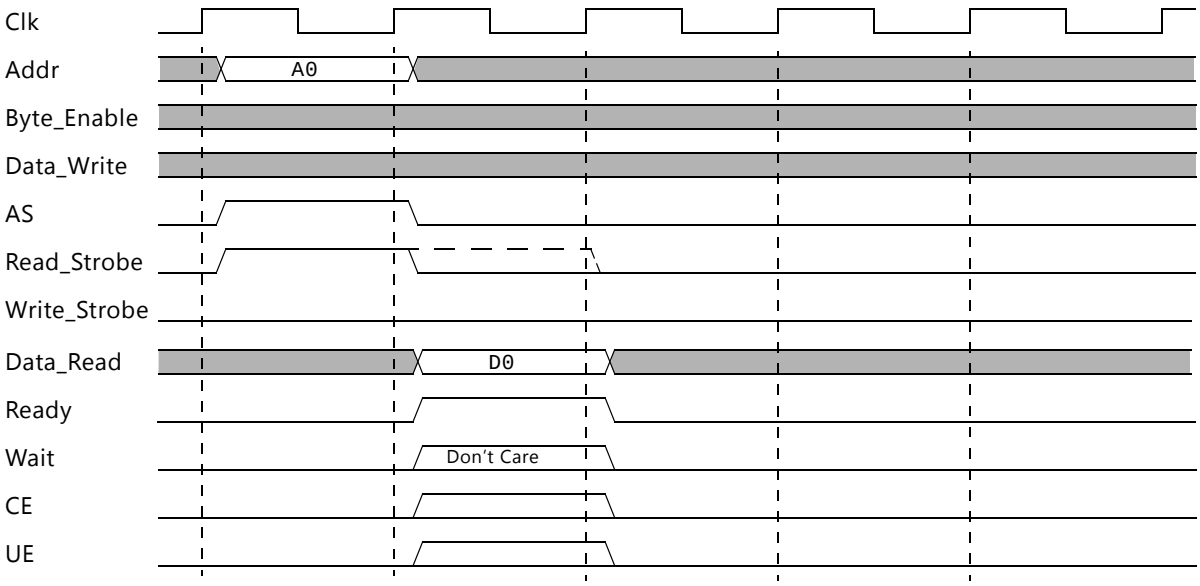

Figure 3-4: **LMB Generic Read Operation, 0 Wait States**

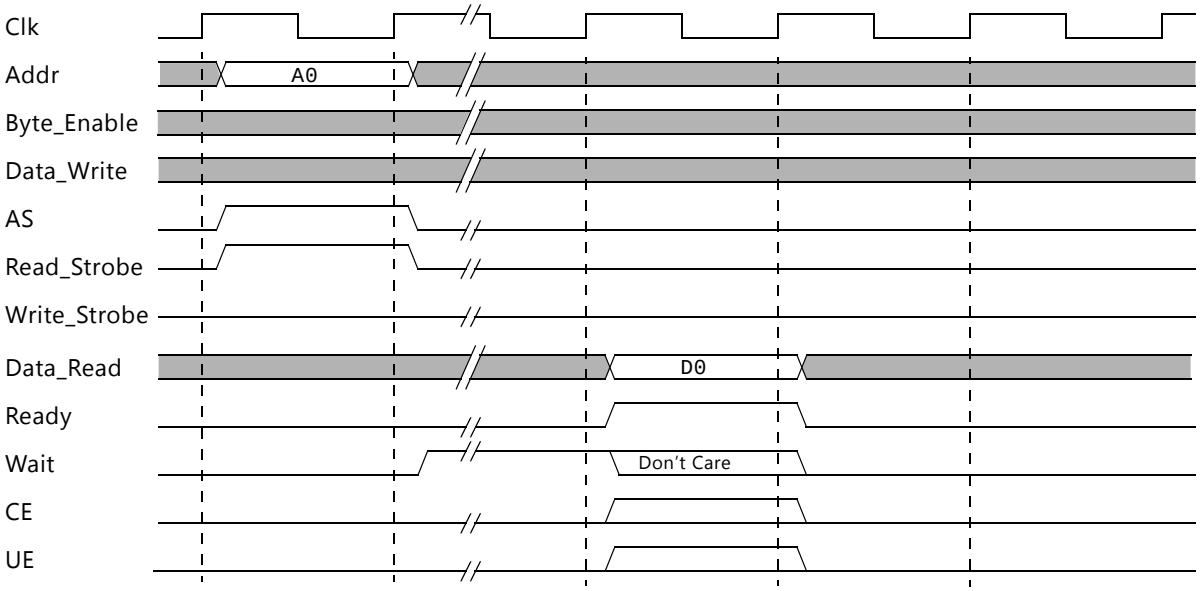

Figure 3-5: **LMB Generic Read Operation, N Wait States**

Back-to-Back Write Operation

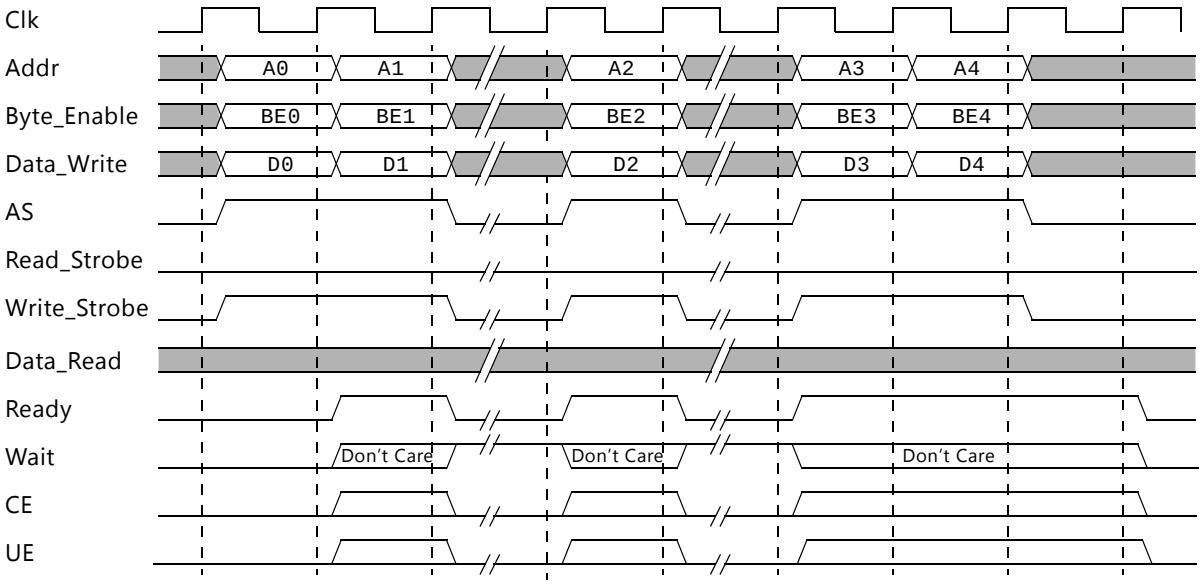

Figure 3-6: **LMB Back-to-Back Write Operation**

Back-to-Back Read Operation

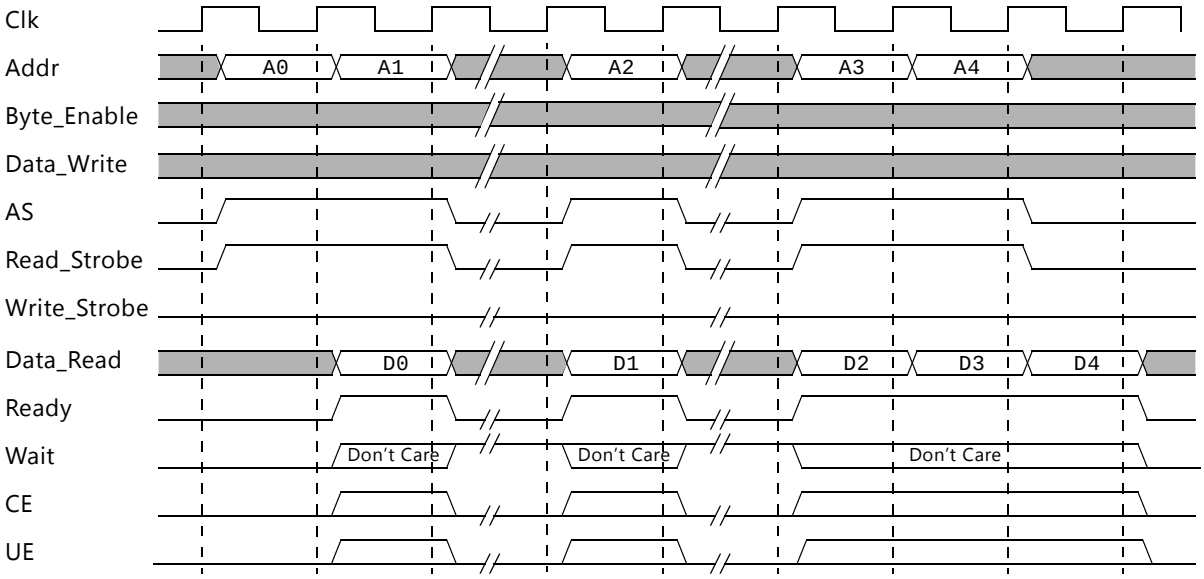

Figure 3-7: **LMB Back-to-Back Read Operation**

Back-to-Back Mixed Write/Read Operation

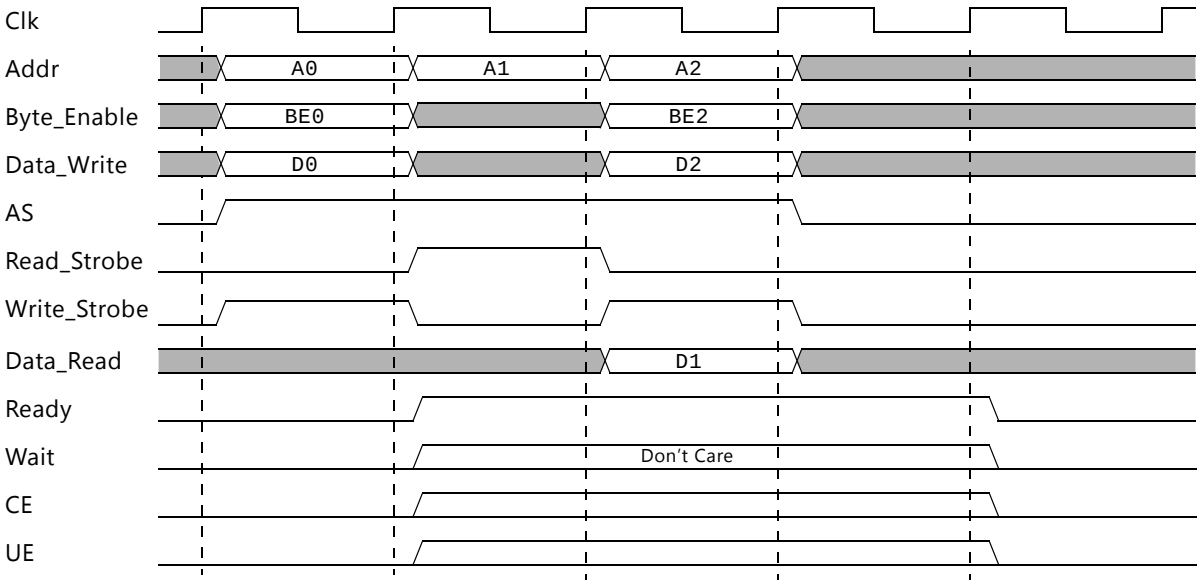

Figure 3-8: **Back-to-Back Mixed Write/Read Operation, 0 Wait States**

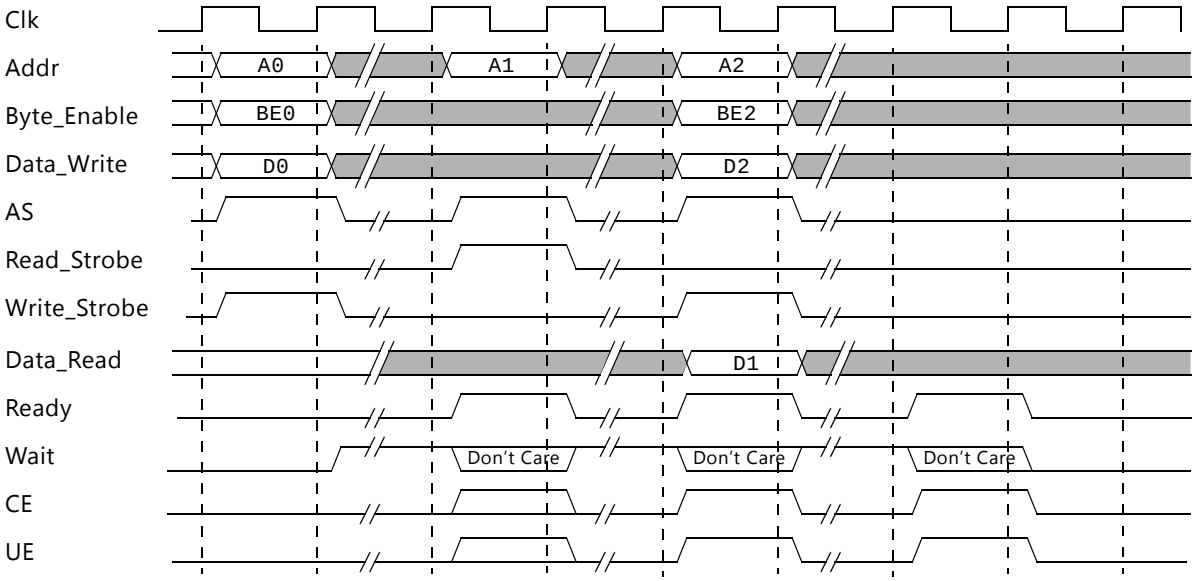

Figure 3-9: **Back-to-Back Mixed Write/Read Operation, N Wait States**

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 are shown in Table 3-7 and Table 3-8, and data steering for write cycles are shown in Table 3-9 and Table 3-10.

Table 3-7: **Big Endian Read Data Steering (Load to Register rD)**

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

Table 3-8: **Little Endian Read Data Steering (Load to Register rD)**

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

Table 3-9: **Big Endian Write Data Steering (Store from Register rD)**

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

Table 3-10: **Little Endian Write Data Steering (Store from Register rD)**

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

Note: Other masters may have more restrictive requirements for byte lane placement than those allowed by MicroBlaze. 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.

Lockstep Interface Description

The lockstep interface on MicroBlaze is designed to connect a master and one or more slave MicroBlaze instances. The lockstep signals on MicroBlaze are listed in Table 3-11.

Table 3-11: **MicroBlaze Lockstep Signals**

Signal Name	Description	VHDL Type	Direction
Lockstep_Master_Out	Output with signals going from master to slave MicroBlaze. Not connected on slaves.	std_logic	output
Lockstep_Slave_In	Input with signals coming from master to slave MicroBlaze. Not connected on master.	std_logic	input
Lockstep_Out	Output with all comparison signals from both master and slaves.	std_logic	output

The comparison signals provided by Lockstep_Out are listed in Table 3-12.

Table 3-12: **MicroBlaze Lockstep Comparison Signals**

Signal Name	Bus Index Range	VHDL Type
MB_Halted	0	std_logic
MB_Error	1	std_logic
IFetch	2	std_logic
I_AS	3	std_logic
Instr_Addr	4 to 35	std_logic_vector
Data_Addr	36 to 67	std_logic_vector
Data_Write	68 to 99	std_logic_vector
D_AS	100	std_logic
Read_Strobe	101	std_logic
Write_Strobe	102	std_logic
Byte_Enable	103 to 106	std_logic_vector
Reserved	107 to 736	
M_AXI_IP_AWID	737	std_logic
M_AXI_IP_AWADDR	738 to 769	std_logic_vector
M_AXI_IP_AWLEN	770 to 777	std_logic_vector
M_AXI_IP_AWSIZE	778 to 780	std_logic_vector
M_AXI_IP_AWBURST	781 to 782	std_logic_vector
M_AXI_IP_AWLOCK	783	std_logic
M_AXI_IP_AWCACHE	784 to 787	std_logic_vector
M_AXI_IP_AWPROT	788 to 790	std_logic_vector
M_AXI_IP_AWQOS	791 to 794	std_logic_vector
M_AXI_IP_AWVALID	795	std_logic
M_AXI_IP_WDATA	796 to 827	std_logic_vector

Table 3-12: **MicroBlaze Lockstep Comparison Signals *(Cont'd)***

Signal Name	Bus Index Range	VHDL Type
M_AXI_IP_WSTRB	828 to 831	std_logic_vector
M_AXI_IP_WLAST	832	std_logic
M_AXI_IP_WVALID	833	std_logic
M_AXI_IP_BREADY	834	std_logic
M_AXI_IP_ARID	835	std_logic
M_AXI_IP_ARADDR	836 to 867	std_logic_vector
M_AXI_IP_ARLEN	868 to 875	std_logic_vector
M_AXI_IP_ARSIZE	876 to 878	std_logic_vector
M_AXI_IP_ARBURST	879 to 880	std_logic_vector
M_AXI_IP_ARLOCK	881	std_logic
M_AXI_IP_ARCACHE	882 to 885	std_logic_vector
M_AXI_IP_ARPROT	886 to 888	std_logic_vector
M_AXI_IP_ARQOS	889 to 892	std_logic_vector
M_AXI_IP_ARVALID	893	std_logic
M_AXI_IP_RREADY	894	std_logic
M_AXI_DP_AWID	895	std_logic
M_AXI_DP_AWADDR	896 to 927	std_logic_vector
M_AXI_DP_AWLEN	928 to 935	std_logic_vector
M_AXI_DP_AWSIZE	936 to 938	std_logic_vector
M_AXI_DP_AWBURST	939 to 940	std_logic_vector
M_AXI_DP_AWLOCK	941	std_logic
M_AXI_DP_AWCACHE	942 to 945	std_logic_vector
M_AXI_DP_AWPROT	946 to 948	std_logic_vector
M_AXI_DP_AWQOS	949 to 952	std_logic_vector
M_AXI_DP_AWVALID	953	std_logic
M_AXI_DP_WDATA	954 to 985	std_logic_vector
M_AXI_DP_WSTRB	986 to 989	std_logic_vector
M_AXI_DP_WLAST	990	std_logic
M_AXI_DP_WVALID	991	std_logic
M_AXI_DP_BREADY	992	std_logic
M_AXI_DP_ARID	993	std_logic
M_AXI_DP_ARADDR	994 to 1025	std_logic_vector
M_AXI_DP_ARLEN	1026 to 1033	std_logic_vector
M_AXI_DP_ARSIZE	1034 to 1036	std_logic_vector
M_AXI_DP_ARBURST	1037 to 1038	std_logic_vector
M_AXI_DP_ARLOCK	1039	std_logic
M_AXI_DP_ARCACHE	1040 to 1043	std_logic_vector
M_AXI_DP_ARPROT	1044 to 1046	std_logic_vector
M_AXI_DP_ARQOS	1047 to 1050	std_logic_vector

Table 3-12: **MicroBlaze Lockstep Comparison Signals *(Cont'd)***

Signal Name	Bus Index Range	VHDL Type
M_AXI_DP_ARVALID	1051	std_logic
M_AXI_DP_RREADY	1052	std_logic
Reserved	1053 to 1644	
M*n*_AXIS_TLAST	1645 + *n* * 35	std_logic
M*n*_AXIS_TDATA	1646 + *n* * 35 to 1677 + n * 35	std_logic_vector
M*n*_AXIS_TVALID	1678 + *n* * 35	std_logic
S*n*_AXIS_TREADY	1679 + *n* * 35	std_logic
M_AXI_IC_AWID	2205	std_logic
M_AXI_IC_AWADDR	2206 to 2237	std_logic_vector
M_AXI_IC_AWLEN	2238 to 2245	std_logic_vector
M_AXI_IC_AWSIZE	2246 to 2248	std_logic_vector
M_AXI_IC_AWBURST	2249 to 2250	std_logic_vector
M_AXI_IC_AWLOCK	2251	std_logic
M_AXI_IC_AWCACHE	2252 to 2255	std_logic_vector
M_AXI_IC_AWPROT	2256 to 2258	std_logic_vector
M_AXI_IC_AWQOS	2259 to 2262	std_logic_vector
M_AXI_IC_AWVALID	2263	std_logic
M_AXI_IC_AWUSER	2264 to 2268	std_logic_vector
M_AXI_IC_AWDOMAIN[1]	2269 to 2270	std_logic_vector
M_AXI_IC_AWSNOOP[1]	2271 to 2273	std_logic_vector
M_AXI_IC_AWBAR[1]	2274 to 2275	std_logic_vector
M_AXI_IC_WDATA	2276 to 2787	std_logic_vector
M_AXI_IC_WSTRB	2788 to 2851	std_logic_vector
M_AXI_IC_WLAST	2852	std_logic
M_AXI_IC_WVALID	2853	std_logic
M_AXI_IC_WUSER	2854	std_logic
M_AXI_IC_BREADY	2855	std_logic
M_AXI_IC_WACK	2856	std_logic
M_AXI_IC_ARID	2857	std_logic_vector
M_AXI_IC_ARADDR	2858 to 2889	std_logic_vector
M_AXI_IC_ARLEN	2890 to 2897	std_logic_vector
M_AXI_IC_ARSIZE	2898 to 2900	std_logic_vector
M_AXI_IC_ARBURST	2901 to 2902	std_logic_vector
M_AXI_IC_ARLOCK	2903	std_logic
M_AXI_IC_ARCACHE	2904 to 2907	std_logic_vector
M_AXI_IC_ARPROT	2908 to 2910	std_logic_vector
M_AXI_IC_ARQOS	2911 to 2914	std_logic_vector
M_AXI_IC_ARVALID	2915	std_logic
M_AXI_IC_ARUSER	2916 to 2920	std_logic_vector

Table 3-12: **MicroBlaze Lockstep Comparison Signals *(Cont'd)***

Signal Name	Bus Index Range	VHDL Type
M_AXI_IC_ARDOMAIN[1]	2921 to 2922	std_logic_vector
M_AXI_IC_ARSNOOP[1]	2923 to 2926	std_logic_vector
M_AXI_IC_ARBAR[1]	2927 to 2928	std_logic_vector
M_AXI_IC_RREADY	2929	std_logic
M_AXI_IC_RACK[1]	2930	std_logic
M_AXI_IC_ACREADY[1]	2931	std_logic
M_AXI_IC_CRVALID[1]	2932	std_logic
M_AXI_IC_CRRESP[1]	2933 to 2937	std_logic_vector
M_AXI_IC_CDVALID[1]	2938	std_logic
M_AXI_IC_CDLAST[1]	2939	std_logic
M_AXI_DC_AWID	2940	std_logic
M_AXI_DC_AWADDR	2941 to 2972	std_logic_vector
M_AXI_DC_AWLEN	2973 to 2980	std_logic_vector
M_AXI_DC_AWSIZE	2981 to 2983	std_logic_vector
M_AXI_DC_AWBURST	2984 to 2985	std_logic_vector
M_AXI_DC_AWLOCK	2986	std_logic
M_AXI_DC_AWCACHE	2987 to 2990	std_logic_vector
M_AXI_DC_AWPROT	2991 to 2993	std_logic_vector
M_AXI_DC_AWQOS	2994 to 2997	std_logic_vector
M_AXI_DC_AWVALID	2998	std_logic
M_AXI_DC_AWUSER	2999 to 3003	std_logic_vector
M_AXI_DC_AWDOMAIN[1]	3004 to 3005	std_logic_vector
M_AXI_DC_AWSNOOP[1]	3006 to 3008	std_logic_vector
M_AXI_DC_AWBAR[1]	3009 to 3010	std_logic_vector
M_AXI_DC_WDATA	3011 to 3522	std_logic_vector
M_AXI_DC_WSTRB[1]	3523 to 3586	std_logic_vector
M_AXI_DC_WLAST	3587	std_logic
M_AXI_DC_WVALID	3588	std_logic
M_AXI_DC_WUSER	3589	std_logic
M_AXI_DC_BREADY	3590	std_logic
M_AXI_DC_WACK[1]	3591	std_logic
M_AXI_DC_ARID	3592	std_logic
M_AXI_DC_ARADDR	3593 to 3624	std_logic_vector
M_AXI_DC_ARLEN	3625 to 3632	std_logic_vector
M_AXI_DC_ARSIZE	3633 to 3635	std_logic_vector
M_AXI_DC_ARBURST	3636 to 3637	std_logic_vector
M_AXI_DC_ARLOCK	3638	std_logic
M_AXI_DC_ARCACHE	3639 to 3642	std_logic_vector
M_AXI_DC_ARPROT	3643 to 3645	std_logic_vector

Table 3-12: **MicroBlaze Lockstep Comparison Signals *(Cont'd)***

Signal Name	Bus Index Range	VHDL Type
M_AXI_DC_ARQOS	3646 to 3649	std_logic_vector
M_AXI_DC_ARVALID	3650	std_logic
M_AXI_DC_ARUSER	3651 to 3655	std_logic_vector
M_AXI_DC_ARDOMAIN[1]	3656 to 3657	std_logic_vector
M_AXI_DC_ARSNOOP[1]	3658 to 3661	std_logic_vector
M_AXI_DC_ARBAR[1]	3662 to 3663	std_logic_vector
M_AXI_DC_RREADY	3664	std_logic
M_AXI_DC_RACK[1]	3665	std_logic
M_AXI_DC_ACREADY[1]	3666	std_logic
M_AXI_DC_CRVALID[1]	3667	std_logic
M_AXI_DC_CRRESP[1]	3668 to 3672	std_logic_vector
M_AXI_DC_CDVALID[1]	3673	std_logic
M_AXI_DC_CDLAST[1]	3674	std_logic
Trace_Instruction	3675 to 3706	std_logic_vector
Trace_Valid_Instr	3707	std_logic
Trace_PC	3708 to 3739	std_logic_vector
Trace_Reg_Write	3740	std_logic
Trace_Reg_Addr	3741 to 3745	std_logic_vector
Trace_MSR_Reg	3746 to 3760	std_logic_vector
Trace_PID_Reg	3761 to 3768	std_logic_vector
Trace_New_Reg_Value	3769 to 3800	std_logic_vector
Trace_Exception_Taken	3801	std_logic
Trace_Exception_Kind	3802 to 3806	std_logic_vector
Trace_Jump_Taken	3807	std_logic
Trace_Delay_Slot	3808	std_logic
Trace_Data_Address	3809 to 3840	std_logic_vector
Trace_Data_Write_Value	3841 to 3872	std_logic_vector
Trace_Data_Byte_Enable	3873 to 3876	std_logic_vector
Trace_Data_Access	3877	std_logic
Trace_Data_Read	3878	std_logic
Trace_Data_Write	3879	std_logic
Trace_DCache_Req	3880	std_logic
Trace_DCache_Hit	3881	std_logic
Trace_DCache_Rdy	3882	std_logic
Trace_DCache_Read	3883	std_logic
Trace_ICache_Req	3884	std_logic
Trace_ICache_Hit	3885	std_logic
Trace_ICache_Rdy	3886	std_logic
Trace_OF_PipeRun	3887	std_logic

Table 3-12: **MicroBlaze Lockstep Comparison Signals *(Cont'd)***

Signal Name	Bus Index Range	VHDL Type
Trace_EX_PipeRun	3888	std_logic
Trace_MEM_PipeRun	3889	std_logic
Trace_MB_Halted	3890	std_logic
Trace_Jump_Hit	3891	std_logic
Reserved for future use	3892 to 4095	

1. This signal is only used when C_INTERCONNECT = 3 (ACE).

Debug Interface Description

The debug interface on MicroBlaze is designed to work with the Xilinx Microprocessor Debug Module (MDM) IP core. The MDM is controlled by the Xilinx Microprocessor Debugger (XMD) through the JTAG port of the FPGA. The MDM can control multiple MicroBlaze processors at the same time. The debug signals are grouped in the DEBUG bus. The debug signals on MicroBlaze are listed in Table 3-13.

Table 3-13: **MicroBlaze Debug Signals**

Signal Name	Description	VHDL Type	Direction
Dbg_Clk	JTAG clock from MDM	std_logic	input
Dbg_TDI	JTAG TDI from MDM	std_logic	input
Dbg_TDO	JTAG TDO to MDM	std_logic	output
Dbg_Reg_En	Debug register enable from MDM	std_logic_vector	input
Dbg_Shift[1]	JTAG BSCAN shift signal from MDM	std_logic	input
Dbg_Capture	JTAG BSCAN capture signal from MDM	std_logic	input
Dbg_Update	JTAG BSCAN update signal from MDM	std_logic	input
Debug_Rst[1]	Reset signal from MDM, active high. Should be held for at least 1 Clk clock cycle.	std_logic	input
Dbg_Trig_In[2]	Cross trigger event input to MDM	std_logic_vector	output
Dbg_Trig_Ack_In[2]	Cross trigger event input acknowledge from MDM	std_logic_vector	input
Dbg_Trig_Out[2]	Cross trigger action output from MDM	std_logic_vector	input
Dbg_Trig_Ack_Out[2]	Cross trigger action output acknowledge to MDM	std_logic_vector	output
Dbg_Trace_Data[3]	External Program Trace data output to MDM	std_logic_vector	output
Dbg_Trace_Valid[3]	External Program Trace valid to MDM	std_logic	output
Dbg_Trace_Ready[3]	External Program Trace ready from MDM	std_logic	input
Dbg_Trace_Clk[3]	External Program Trace clock from MDM	std_logic	input

1. Updated for MicroBlaze v7.00: Dbg_Shift added and Debug_Rst included in DEBUG bus
2. Updated for MicroBlaze v9.3: Dbg_Trig signals added to DEBUG bus
3. Updated for MicroBlaze v9.4: External Program Trace signal added to DEBUG bus

Trace Interface Description

The MicroBlaze core exports a number of internal signals for trace purposes. This signal interface is not standardized and new revisions of the processor may not be backward compatible for signal selection or functionality. It is recommended that you not design custom logic for these signals, but rather to use them via Xilinx provided analysis IP. The trace signals are grouped in the TRACE bus. The current set of trace signals were last updated for MicroBlaze v7.30 and are listed in Table 3-14.

The mapping of the MSR bits is shown in Table 3-15. For a complete description of the Machine Status Register, see "Special Purpose Registers".

The Trace exception types are listed in Table 3-16. All unused Trace exception types are reserved.

Table 3-14: **MicroBlaze Trace Signals**

Signal Name	Description	VHDL Type	Direction
Trace_Valid_Instr	Valid instruction on trace port.	std_logic	output
Trace_Instruction [1]	Instruction code	std_logic_vector (0 to 31)	output
Trace_PC [1]	Program counter	std_logic_vector (0 to 31)	output
Trace_Reg_Write [1]	Instruction writes to the register file	std_logic	output
Trace_Reg_Addr [1]	Destination register address	std_logic_vector (0 to 4)	output
Trace_MSR_Reg[1]	Machine status register. The mapping of the register bits is documented below.	std_logic_vector (0 to 14)[2]	output
Trace_PID_Reg[1]	Process identifier register	std_logic_vector (0 to 7)	output
Trace_New_Reg_Value[1]	Destination register update value	std_logic_vector (0 to 31)	output
Trace_Exception_Taken[1,2]	Instruction result in taken exception	std_logic	output
Trace_Exception_Kind[1]	Exception type. The description for the exception type is documented below.	std_logic_vector (0 to 4)[2]	output
Trace_Jump_Taken[1]	Branch instruction evaluated true, i.e taken	std_logic	output
Trace_Jump_Hit[1,3]	Branch Target Cache hit	std_logic	output
Trace_Delay_Slot[1]	Instruction is in delay slot of a taken branch	std_logic	output
Trace_Data_Access[1]	Valid D-side memory access	std_logic	output
Trace_Data_Address[1]	Address for D-side memory access	std_logic_vector (0 to 31)	output
Trace_Data_Write_Value[1]	Value for D-side memory write access	std_logic_vector (0 to 31)	output
Trace_Data_Byte_Enable[1]	Byte enables for D-side memory access	std_logic_vector (0 to 3)	output
Trace_Data_Read[1]	D-side memory access is a read	std_logic	output
Trace_Data_Write[1]	D-side memory access is a write	std_logic	output

Table 3-14: **MicroBlaze Trace Signals *(Cont'd)***

Signal Name	Description	VHDL Type	Direction
Trace_DCache_Req	Data memory address is within D-Cache range	std_logic	output
Trace_DCache_Hit	Data memory address is present in D-Cache	std_logic	output
Trace_DCache_Rdy	Data memory address is within D-Cache range and the access is completed	std_logic	output
Trace_DCache_Read[4]	The D-Cache request is a read	std_logic	output
Trace_ICache_Req	Instruction memory address is within I-Cache range	std_logic	output
Trace_ICache_Hit	Instruction memory address is present in I-Cache	std_logic	output
Trace_ICache_Rdy	Instruction memory address is within I-Cache range and the access is completed	std_logic	output
Trace_OF_PipeRun	Pipeline advance for Decode stage	std_logic	output
Trace_EX_PipeRun[3]	Pipeline advance for Execution stage	std_logic	output
Trace_MEM_PipeRun[3]	Pipeline advance for Memory stage	std_logic	output
Trace_MB_Halted	Pipeline is halted by debug	std_logic	output

1. Valid only when Trace_Valid_Instr = 1
2. Valid only when Trace_Exception_Taken = 1
3. Not used with area optimization feature
4. Valid only when Trace_DCache_Req = 1

Table 3-15: **Mapping of Trace MSR**

Trace_MSR_Reg	Machine Status Register		
Bit	Bit	Name	Description
0	17	VMS	Virtual Protected Mode Save
1	18	VM	Virtual Protected Mode
2	19	UMS	User Mode Save
3	20	UM	User Mode
4	21	PVR	Processor Version Register exists
5	22	EIP	Exception In Progress
6	23	EE	Exception Enable
7	24	DCE	Data Cache Enable
8	25	DZO	Division by Zero or Division Overflow
9	26	ICE	Instruction Cache Enable

Table 3-15: **Mapping of Trace MSR**

Trace_MSR_Reg	Machine Status Register		
Bit	**Bit**	**Name**	**Description**
10	27	FSL	AXI4-Stream Error
11	28	BIP	Break in Progress
12	29	C	Arithmetic Carry
13	30	IE	Interrupt Enable
14	31	Reserved	Reserved

Table 3-16: **Type of Trace Exception**

Trace_Exception_Kind [0:4]	Description
00000	Stream exception
00001	Unaligned exception
00010	Illegal Opcode exception
00011	Instruction Bus exception
00100	Data Bus exception
00101	Divide exception
00110	FPU exception
00111	Privileged instruction exception
01010	Interrupt
01011	External non maskable break
01100	External maskable break
10000	Data storage exception
10001	Instruction storage exception
10010	Data TLB miss exception
10011	Instruction TLB miss exception

MicroBlaze Core Configurability

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

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

Parameters valid for MicroBlaze v9.2 are listed in Table 3-17. Not all of these are recognized by older versions of MicroBlaze; however, the configurability is fully backward compatibility.

Note: Shaded rows indicate that the parameter has a fixed value and *cannot* be modified.

Table 3-17: **MPD Parameters**

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_FAMILY	Target Family	Listed in Table 3-18	virtex7	yes	string
C_DATA_SIZE	Data Size	32	32	NA	integer
C_DYNAMIC_BUS_SIZING	Legacy	1	1	NA	integer
C_SCO	Xilinx internal	0	0	NA	integer
C_AREA_OPTIMIZED	Select implementation to optimize area with lower instruction throughput	0, 1	0		integer
C_OPTIMIZATION	Reserved for future use	0	0	NA	integer
C_INTERCONNECT	Select interconnect 2 = AXI4 only 3 = AXI4 and ACE	2, 3	2		integer
C_ENDIANNESS	Select endianness 1 = Little Endian	1	1	yes	integer
C_BASE_VECTORS[1]	Configurable base vectors	0x00000000-0xffffff80	0x00000000		std_logic_vector
C_FAULT_TOLERANT	Implement fault tolerance	0, 1	0	yes	integer
C_ECC_USE_CE_EXCEPTION	Generate exception for correctable ECC error	0,1	0		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_LOCKSTEP_SLAVE	Lockstep Slave	0, 1	0		integer
C_AVOID_PRIMITIVES	Disallow FPGA primitives 0 = None 1 = SRL 2 = LUTRAM 3 = Both	0, 1, 2, 3	0		integer
C_PVR	Processor version register mode selection 0 = None 1 = Basic 2 = Full	0, 1, 2	0		integer
C_PVR_USER1	Processor version register USER1 constant	0x00-0xff	0x00		std_logic_vector (0 to 7)
C_PVR_USER2	Processor version register USER2 constant	0x00000000-0xffffffff	0x00000000		std_logic_vector (0 to 31)
C_RESET_MSR	Reset value for MSR register	0x00, 0x20, 0x80, 0xa0	0x00		std_logic_vector
C_INSTANCE	Instance Name	Any instance name	microblaze	yes	string
C_D_AXI	Data side AXI interface	0, 1	0		integer
C_D_LMB	Data side LMB interface	0, 1	1		integer
C_I_AXI	Instruction side AXI interface	0, 1	0		integer
C_I_LMB	Instruction side LMB interface	0, 1	1		integer
C_USE_BARREL	Include barrel shifter	0, 1	0		integer
C_USE_DIV	Include hardware divider	0, 1	0		integer
C_USE_HW_MUL	Include hardware multiplier 0 = None 1 = Mul32 2 = Mul64	0, 1, 2	1		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_USE_FPU	Include hardware floating point unit 0 = None 1 = Basic 2 = Extended	0, 1, 2	0		integer
C_USE_MSR_INSTR	Enable use of instructions: MSRSET and MSRCLR	0, 1	1		integer
C_USE_PCMP_INSTR	Enable use of instructions: CLZ, PCMPBF, PCMPEQ, and PCMPNE	0, 1	1		integer
C_USE_REORDER_INSTR	Enable use of instructions: Reverse load, reverse store, and swap	0, 1	1		integer
C_UNALIGNED_EXCEPTIONS	Enable exception handling for unaligned data accesses	0, 1	0		integer
C_ILL_OPCODE_EXCEPTION	Enable exception handling for illegal op-code	0, 1	0		integer
C_M_AXI_I_BUS_EXCEPTION	Enable exception handling for M_AXI_I bus error	0, 1	0		integer
C_M_AXI_D_BUS_EXCEPTION	Enable exception handling for M_AXI_D bus error	0, 1	0		integer
C_DIV_ZERO_EXCEPTION	Enable exception handling for division by zero or division overflow	0, 1	0		integer
C_FPU_EXCEPTION	Enable exception handling for hardware floating point unit exceptions	0, 1	0		integer
C_OPCODE_0x0_ILLEGAL	Detect opcode 0x0 as an illegal instruction	0,1	0		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_FSL_EXCEPTION	Enable exception handling for Stream Links	0,1	0		integer
C_ECC_USE_CE_EXCEPTION	Generate Bus Error Exceptions for correctable errors	0,1	0		integer
C_USE_STACK_PROTECTION	Generate exception for stack overflow or stack underflow	0,1	0		integer
C_IMPRECISE_EXCEPTIONS	Allow imprecise exceptions for ECC errors in LMB memory	0,1	0		integer
C_DEBUG_ENABLED	MDM Debug interface 0 = None 1 = Basic 2 = Extended	0,1,2	1		integer
C_NUMBER_OF_PC_BRK	Number of hardware breakpoints	0-8	1		integer
C_NUMBER_OF_RD_ADDR_BRK	Number of read address watchpoints	0-4	0		integer
C_NUMBER_OF_WR_ADDR_BRK	Number of write address watchpoints	0-4	0		integer
C_DEBUG_EVENT_COUNTERS	Number of Performance Monitor event counters	0-48	5		integer
C_DEBUG_LATENCY_COUNTERS	Number of Performance Monitor latency counters	0-7	1		integer
C_DEBUG_COUNTER_WIDTH	Performance Monitor counter width	32,48,64	32		integer
C_DEBUG_TRACE_SIZE	Trace Buffer size	0, 8192, 16384, 32768, 65536, 131072	8192		integer
C_DEBUG_PROFILE_SIZE	Profile Buffer size	0, 4096, 8192, 16384, 32768, 65536, 131072	0		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_DEBUG_EXTERNAL_TRACE	External Program Trace	0,1	0	yes	integer
C_INTERRUPT_IS_EDGE	Level/Edge Interrupt	0, 1	0	yes	integer
C_EDGE_IS_POSITIVE	Negative/Positive Edge Interrupt	0, 1	1	yes	integer
C_FSL_LINKS	Number of AXI-Stream interfaces	0-16	0		integer
C_USE_EXTENDED_FSL_INSTR	Enable use of extended stream instructions	0, 1	0		integer
C_ICACHE_BASEADDR	Instruction cache base address	0x00000000 - 0xFFFFFFFF	0x00000000		std_logic_vector
C_ICACHE_HIGHADDR	Instruction cache high address	0x00000000 - 0xFFFFFFFF	0x3FFFFFFF		std_logic_vector
C_USE_ICACHE	Instruction cache	0, 1	0		integer
C_ALLOW_ICACHE_WR	Instruction cache write enable	0, 1	1		integer
C_ICACHE_LINE_LEN	Instruction cache line length	4, 8, 16	4		integer
C_ICACHE_ALWAYS_USED	Instruction cache interface used for all memory accesses in the cacheable range	0, 1	1		integer
C_ICACHE_FORCE_TAG_LUTRAM	Instruction cache tag always implemented with distributed RAM	0, 1	0		integer
C_ICACHE_STREAMS	Instruction cache streams	0, 1	0		integer
C_ICACHE_VICTIMS	Instruction cache victims	0, 2, 4, 8	0		integer
C_ICACHE_DATA_WIDTH	Instruction cache data width 0 = 32 bits 1 = Full cache line 2 = 512 bits	0, 1, 2	0		integer
C_ADDR_TAG_BITS	Instruction cache address tags	0-25	17	yes	integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_CACHE_BYTE_SIZE	Instruction cache size	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536[2]	8192		integer
C_DCACHE_BASEADDR	Data cache base address	0x00000000 - 0xFFFFFFFF	0x00000000		std_logic_vector
C_DCACHE_HIGHADDR	Data cache high address	0x00000000 - 0xFFFFFFFF	0x3FFFFFFF		std_logic_vector
C_USE_DCACHE	Data cache	0, 1	0		integer
C_ALLOW_DCACHE_WR	Data cache write enable	0, 1	1		integer
C_DCACHE_LINE_LEN	Data cache line length	4, 8, 16	4		integer
C_DCACHE_ALWAYS_USED	Data cache interface used for all accesses in the cacheable range	0, 1	1		integer
C_DCACHE_FORCE_TAG_LUTRAM	Data cache tag always implemented with distributed RAM	0, 1	0		integer
C_DCACHE_USE_WRITEBACK	Data cache write-back storage policy used	0, 1	0		integer
C_DCACHE_VICTIMS	Data cache victims	0, 2, 4, 8	0		integer
C_DCACHE_DATA_WIDTH	Data cache data width 0 = 32 bits 1 = Full cache line 2 = 512 bits	0, 1, 2	0		integer
C_DCACHE_ADDR_TAG	Data cache address tags	0-25	17	yes	integer
C_DCACHE_BYTE_SIZE	Data cache size	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536[2]	8192		integer
C_USE_MMU[3]	Memory Management: 0 = None 1 = User Mode 2 = Protection 3 = Virtual	0, 1, 2, 3	0		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_MMU_DTLB_SIZE[3]	Data shadow Translation Look-Aside Buffer size	1, 2, 4, 8	4		integer
C_MMU_ITLB_SIZE[3]	Instruction shadow Translation Look-Aside Buffer size	1, 2, 4, 8	2		integer
C_MMU_TLB_ACCESS[3]	Access to memory management special registers: 0 = Minimal 1 = Read 2 = Write 3 = Full	0, 1, 2, 3	3		integer
C_MMU_ZONES[3]	Number of memory protection zones	0-16	16		integer
C_MMU_PRIVILEGED_INSTR[3]	Privileged instructions 0 = Full protection 1 = Allow stream instrs	0,1	0		integer
C_USE_INTERRUPT	Enable interrupt handling 0 = No interrupt 1 = Standard interrupt 2 = Low-latency interrupt	0, 1, 2	1	yes	integer
C_USE_EXT_BRK	Enable external break handling	0,1	0	yes	integer
C_USE_EXT_NM_BRK	Enable external non-maskable break handling	0,1	0	yes	integer
C_USE_BRANCH_TARGET_CACHE[3]	Enable Branch Target Cache	0,1	0		integer
C_BRANCH_TARGET_CACHE_SIZE[3]	Branch Target Cache size: 0 = Default 1 = 8 entries 2 = 16 entries 3 = 32 entries 4 = 64 entries 5 = 512 entries 6 = 1024 entries 7 = 2048 entries	0-7	0		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_M_AXI_DP_THREAD_ID_WIDTH	Data side AXI thread ID width	1	1		integer
C_M_AXI_DP_DATA_WIDTH	Data side AXI data width	32	32		integer
C_M_AXI_DP_ADDR_WIDTH	Data side AXI address width	32	32		integer
C_M_AXI_DP_SUPPORTS_THREADS	Data side AXI uses threads	0	0		integer
C_M_AXI_DP_SUPPORTS_READ	Data side AXI support for read accesses	1	1		integer
C_M_AXI_DP_SUPPORTS_WRITE	Data side AXI support for write accesses	1	1		integer
C_M_AXI_DP_SUPPORTS_NARROW_BURST	Data side AXI narrow burst support	0	0		integer
C_M_AXI_DP_PROTOCOL	Data side AXI protocol	AXI4, AXI4LITE	AXI4 LITE	yes	string
C_M_AXI_DP_EXCLUSIVE_ACCESS	Data side AXI exclusive access support	0,1	0		integer
C_M_AXI_IP_THREAD_ID_WIDTH	Instruction side AXI thread ID width	1	1		integer
C_M_AXI_IP_DATA_WIDTH	Instruction side AXI data width	32	32		integer
C_M_AXI_IP_ADDR_WIDTH	Instruction side AXI address width	32	32		integer
C_M_AXI_IP_SUPPORTS_THREADS	Instruction side AXI uses threads	0	0		integer
C_M_AXI_IP_SUPPORTS_READ	Instruction side AXI support for read accesses	1	1		integer
C_M_AXI_IP_SUPPORTS_WRITE	Instruction side AXI support for write accesses	0	0		integer
C_M_AXI_IP_SUPPORTS_NARROW_BURST	Instruction side AXI narrow burst support	0	0		integer
C_M_AXI_IP_PROTOCOL	Instruction side AXI protocol	AXI4LITE	AXI4 LITE		string

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_M_AXI_DC_THREAD_ID_WIDTH	Data cache AXI ID width	1	1		integer
C_M_AXI_DC_DATA_WIDTH	Data cache AXI data width	32, 64, 128, 256, 512	32		integer
C_M_AXI_DC_ADDR_WIDTH	Data cache AXI address width	32	32		integer
C_M_AXI_DC_SUPPORTS_THREADS	Data cache AXI uses threads	0	0		integer
C_M_AXI_DC_SUPPORTS_READ	Data cache AXI support for read accesses	1	1		integer
C_M_AXI_DC_SUPPORTS_WRITE	Data cache AXI support for write accesses	1	1		integer
C_M_AXI_DC_SUPPORTS_NARROW_BURST	Data cache AXI narrow burst support	0	0		integer
C_M_AXI_DC_SUPPORTS_USER_SIGNALS	Data cache AXI user signal support	1	1		integer
C_M_AXI_DC_PROTOCOL	Data cache AXI protocol	AXI4	AXI4		string
C_M_AXI_DC_AWUSER_WIDTH	Data cache AXI user width	5	5		integer
C_M_AXI_DC_ARUSER_WIDTH	Data cache AXI user width	5	5		integer
C_M_AXI_DC_WUSER_WIDTH	Data cache AXI user width	1	1		integer
C_M_AXI_DC_RUSER_WIDTH	Data cache AXI user width	1	1		integer
C_M_AXI_DC_BUSER_WIDTH	Data cache AXI user width	1	1		integer
C_M_AXI_DC_EXCLUSIVE_ACCESS	Data cache AXI exclusive access support	0,1	0		integer
C_M_AXI_DC_USER_VALUE	Data cache AXI user value	0-31	31		integer
C_M_AXI_IC_THREAD_ID_WIDTH	Instruction cache AXI ID width	1	1		integer
C_M_AXI_IC_DATA_WIDTH	Instruction cache AXI data width	32, 64, 128, 256, 512	32		integer

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_M_AXI_IC_ADDR_WIDTH	Instruction cache AXI address width	32	32		integer
C_M_AXI_IC_SUPPORTS_THREADS	Instruction cache AXI uses threads	0	0		integer
C_M_AXI_IC_SUPPORTS_READ	Instruction cache AXI support for read accesses	1	1		integer
C_M_AXI_IC_SUPPORTS_WRITE	Instruction cache AXI support for write accesses	0	0		integer
C_M_AXI_IC_SUPPORTS_NARROW_BURST	Instruction cache AXI narrow burst support	0	0		integer
C_M_AXI_IC_SUPPORTS_USER_SIGNALS	Instruction cache AXI user signal support	1	1		integer
C_M_AXI_IC_PROTOCOL	Instruction cache AXI protocol	AXI4	AXI4		string
C_M_AXI_IC_AWUSER_WIDTH	Instruction cache AXI user width	5	5		integer
C_M_AXI_IC_ARUSER_WIDTH	Instruction cache AXI user width	5	5		integer
C_M_AXI_IC_WUSER_WIDTH	Instruction cache AXI user width	1	1		integer
C_M_AXI_IC_RUSER_WIDTH	Instruction cache AXI user width	1	1		integer
C_M_AXI_IC_BUSER_WIDTH	Instruction cache AXI user width	1	1		integer
C_M_AXI_IC_USER_VALUE	Instruction cache AXI user value	0-31	31		integer
C_STREAM_INTERCONNECT	Select AXI4-Stream interconnect	0,1	0		integer
C_Mn_AXIS_PROTOCOL	AXI4-Stream protocol	GENERIC	GENERIC		string
C_Sn_AXIS_PROTOCOL	AXI4-Stream protocol	GENERIC	GENERIC		string

Table 3-17: **MPD Parameters *(Cont'd)***

Parameter Name	Feature/Description	Allowable Values	Default Value	Tool Assigned	VHDL Type
C_Mn_AXIS_DATA_WIDTH	AXI4-Stream master data width	32	32	NA	integer
C_Sn_AXIS_DATA_WIDTH	AXI4-Stream slave data width	32	32	NA	integer

1. The 7 least significant bits must all be 0.
2. Not all sizes are permitted in all architectures. The cache uses between 0 and 32 RAMB primitives (0 if cache size is less than 2048).
3. Not available when C_AREA_OPTIMIZED is set to 1.

Table 3-18: **Parameter C_FAMILY Allowable Values**

	Allowable Values
Artix	aartix7 artix7 artix7l qartix7 qartix7l
Kintex	kintex7 kintex7l qkintex7 qkintex7l kintexu
Virtex	qvirtex7 virtex7 virtexu
Zynq	azynq zynq qzynq